Logic Continuations

by

Christopher T. Haynes

Computer Science Department
Indiana University
Bloomington, Indiana 47405

TECHNICAL REPORT NO. 183

Logic Continuations

by

Christopher T. Haynes

Revised: September, 1986

To appear in *The Journal of Logic Programming.*

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

Logic Continuations

Christopher T. Haynes

Computer Science Department
Indiana University
Bloomington, Indiana 47405 USA

Abstract

We develop a 'complete' embedding of logic programming into Scheme—a lexically scoped Lisp dialect with first-class continuations. Logic variables are bound in the Scheme environment and the success and failure continuations are represented as Scheme continuations. To account for the semantics of logic variables and failure continuations, the state-space model of control is modified in a novel way that generalizes the trail mechanism. This ensures that logic variable bindings are properly restored when continuations are invoked to perform 'lateral' control transfers that are not possible in a traditional logic programming context. It is thereby possible to obtain greater control flexibility while allowing much of a program to be expressed with logic programming.

1. Introduction

Much of the attraction of logic programming systems is that the programmer is relieved of responsibility for specifying control behavior. The system automatically performs resolution theorem proving according to a built-in search strategy, such as the depth-first search of Prolog [6]. However, difficulties arise when the built-in strategy does not suit the programmer's needs. Facilities may be provided to modify the default strategy, such as Prolog's cut, but there are still cases in which these facilities are awkward or inefficient.

We are concerned with problems, such as those arising in large artificial intelligence applications, in which control of the search strategy is of central importance for reasons of efficiency, and perhaps even termination. If a suitable strategy can be identified in advance, it may still be possible to use a traditional logic programming language by employing meta-programming. However, there are other applications in which the search strategy is based upon complicated heuristics and cannot be predicted before runtime. In such cases logic programming is usually abandoned in favor of procedural languages such as Lisp. The simpler control behavior of procedural languages provides a more direct approach to implementing control strategies based on runtime heuristics.

Of particular interest in artificial intelligence applications is the control paradigm of ***non-blind backtracking*** [27]. For example, a process that starts in control context A may introduce an (initially unbound) logic variable X, pass a choice point B, bind X to a, and then proceed to some context C (Figure 1[a]). It is then decided that the choice made at B was probably not a good one, for it has taken much time to get to C and little progress has been made toward the goal. Therefore control backtracks to B, X is unbound, and another choice is taken which causes X to be bound to b and leads to a context D (Figure 1[b]). It may be that from D the second choice looks even worse than the first, and it is desired to perform a 'lateral' control transfer back to C (dotted line).

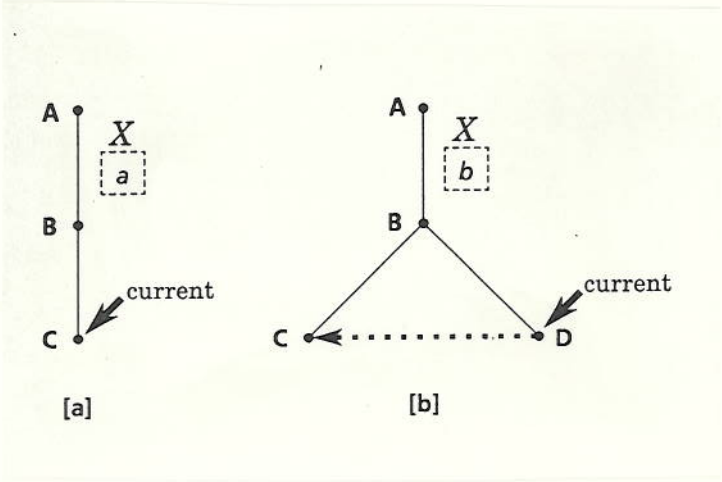

Figure 1. Non-blind backtracking.

Such transfers of control are not possible in most systems because control information is allocated on a stack that must be popped when backtracking from C to B. Such backtracking is *blind*, because it is impossible to return to the previous context. If it is possible to record the control context at C and return to it sometime after backtracking to B, then the backtracking is *non-blind*. Applications of non-blind backtracking include variations on breadth-first search.

Non-blind backtracking allows branches to form in the control information, forming a tree structure. The control tree's root is the initial control context and each node represents a unit of control information, such as a "stack frame". The ancestors of a given node are the nodes that can be reached by a series of returns. Branching of the control tree generally requires heap allocation of control information, though it may still be possible to cache some control information on a stack.

The control context of a computation is known as its *continuation,* because it controls how the computation will continue in the absence of explicit control transfers. In particular, every application has a continuation that may be viewed as a function of one argument. This continuation expects to receive the value returned by the applied function, with which it will continue the computation. Though programming systems must maintain control context information (generally using a stack), this continuation information is generally inaccessible to the programmer. Yet it is possible to provide a means for the programmmer to request that the current continuation be made available as an object of computation. The most convenient form in which to encapsulate the current continuation is as a functional object of one argument, which is sometimes called an "escape procedure". Such encapsulation is necessary to insulate the programmer from the system's representation of control information, which is highly implementation specific.[1]

If continuations are made available as functional objects, they provide an abstraction

[1] In some logic programming implementations the system's success and/or failure continuations may be represented by closures (functional objects with an associated environment). These continuations are part of the implementation and are not accessible to the logic programmer. They should not be confused with the encapsulated continuations discussed here.

of control that may be used to implement a variety of non-standard control behaviors. [11, 12, 14, 31] For maximum generality, continuations must be *first-class* objects. That is, it must be possible for them to be passed to and returned from functions, stored in data structures, and invoked any number of times from any point in a computation. It is then possible to 'mark' the current position in the control tree by saving its continuation. Control may later be transferred to the marked point by simply invoking this continuation.

An approach to obtaining greater control flexibility in logic programs is to 'completely embed' logic programming facilities in a procedural language that provides first-class continuations. Roughly speaking, this means compilation of a logic programming language into a procedural language so that the logic programming and procedural languages share common environment and control contexts. It is then possible to use a multi-paradigm approach to solve complex problems. Most of a program may be written in a logic programming language, but with the ability to escape into the surrounding procedural implementation language to obtain variations on the search strategy. In particular, unrestricted use of non-blind backtracking and lateral control transfers becomes possible when the procedural language supports first-class continuations. However, it is necessary to ensure that logic variable bindings are properly maintained when continuations are invoked. For example, in Figure 1[b] the value of logic variable X should be changed from b to a when control is transferred from D to C. This is accomplished automatically using a mechanism developed in this paper. The resulting 'logic continuations' may then be used to obtain non-standard behavior while allowing much of the program to be expressed using the techniques of logic programming.[2]

The embedding developed here is presented in Scheme—a lexically scoped Lisp dialect with first-class continuations. However, it must be emphasized that the principal technique developed in this paper is applicable to any language with first-class continuations.

Some familiarity with Lisp is assumed, though an overview of the dialect used in this paper is provided in the next section. We then present an embedding taxonomy, which is followed by an embedding of logic programming into Scheme. We are then prepared to present a variant of the state-space model that allows the full power of continuations to be used in the context of logic programming. Finally, we address some efficiency issues and discuss the value of complete embeddings in the general context of non-procedural languages.

2. An overview of Scheme

Scheme is a dialect of Lisp that is applicative order, lexically scoped, and properly tail-recursive [7, 23, 29]. Most importantly, Scheme treats functions and continuations as first-class objects.

See Figure 2 for the syntax of a Scheme subset sufficient for the purposes of this paper. The superscript * denotes zero or more, and $^+$ denotes one or more occurrences of the preceding form. Square brackets are interchangeable with parentheses, and are used in

[2] **The term 'logic continuation' is a contradiction in terms from a logician's perspective, for pure logic is non-procedural and thus divorced from control matters. However, we take the computer scientist's perspective of logic programs as a computational model [26] with an implicit control regime.**

⟨expression⟩ ::=
| ⟨identifier⟩
| (**quote** ⟨object⟩)
| (**lambda** ⟨formals⟩ ⟨expression⟩$^+$)
| (**let** ([⟨identifier⟩ ⟨value⟩]*) ⟨expression⟩$^+$)
| (**letrec** ([⟨identifier⟩ ⟨value⟩]*) ⟨expression⟩$^+$)
| (**rec** ⟨identifier⟩ ⟨expression⟩)
| (**do** ([⟨identifier⟩ ⟨init⟩ ⟨next⟩]*) (⟨predicate⟩ ⟨expression⟩) ⟨body⟩*)
| (**cond** [⟨predicate⟩ ⟨expression⟩*]*)
| (**and** ⟨expression⟩$^+$)
| (**case** ⟨tag⟩ [(⟨symbol⟩$^+$) ⟨expression⟩]$^+$)
| (**define** ⟨identifier⟩ ⟨expression⟩)
| (**set!** ⟨identifier⟩ ⟨expression⟩)
| ⟨application⟩

⟨init⟩, ⟨next⟩, ⟨value⟩, ⟨predicate⟩, ⟨body⟩, ⟨tag⟩, ⟨function⟩ ::= ⟨expression⟩

⟨application⟩ ::= (⟨function⟩ ⟨expression⟩*)

⟨formals⟩ ::= ⟨identifier⟩ | (⟨identifier⟩* . ⟨identifier⟩) | (⟨identifier⟩*)

Figure 2. Syntax of a Scheme subset.

the indicated contexts for readability. **Quote** expressions return the indicated literal object, and '⟨object⟩ is equivalent to (**quote** ⟨object⟩). **Lambda** expressions evaluate to first-class functional objects that statically bind their formal identifiers when invoked. The last formal identifier is bound to a list of all remaining arguments if preceded by a dot. If the formals part consists of a single identifier, the identifier is bound to a list of all the arguments. In every case where a list of expressions is indicated in Figure 2, as in the body of **lambda**, the expressions are evaluated in sequence and the value of the last expression is returned. **Let** makes lexical bindings, while **letrec** makes mutually recursive lexical bindings. **Rec** evaluates its expression in an environment that binds its identifier to the value of the expression itself. (**Rec** is similar to the **label** form of Lisp.) **Do** is the traditional Lisp iteration construct. First the ⟨init⟩ expressions are evaluated and bound to the identifiers. Then if the predicate is false, the ⟨body⟩ expressions (if there are any) are evaluated in sequence, the identifiers are rebound to the values of the ⟨next⟩ expressions, and the process is repeated. When the predicate is true, the value of the expression following it is returned. **Cond** evaluates its predicates in sequence until one is true, and then returns the value of the expression paired with the true predicate. The predicate **else** is always true. **And** is sequential conjunction. **Case** evaluates the tag expression, and then returns the value of the first expression with a corresponding symbol that matches the tag. **Define** assigns to a global identifier. **Set!** modifies an existing lexical identifier. (A bang (**!**) is used to flag operations that involve side-effects.) An application evaluates its expressions (in an unspecified order) and applies the functional value of the first expression to the values of the remaining expressions.

We require a few primitive functions. **Apply** invokes its first argument with the list of

arguments passed as its second argument. `Cons` is the traditional Lisp binary construction operation, with associated selectors `car` and `cdr`, and mutators `set-car!` (Lisp's rplaca) and `set-cdr!` (rplacd). Lists are constructed of cons cells. `List` constructs lists and `reverse!` reverses the list pointers in place. `For-each` (mapc) applies its first argument to each element in the list passed as its second argument. `Eq?`, `equal?`, `pair?`, `null?` and `not` are the usual identity, structural equality, pair (cons cell), empty list (nil) and negation predicates.

The function `call-with-current-continuation`, abbreviated `call/cc`, must be passed a function of one argument. This argument is in turn passed the current continuation, which is the control context of the `call/cc` application, represented as a functional object of one argument.[3] Informally, this continuation represents the remainder of the computation from the `call/cc` application point. At any future time this continuation may be invoked with any value, with the effect that this value is taken as the value of the `call/cc` application [9, 10, 11, 12].

The simplest use of continuations is to 'throw' a value out of an expression directly, without completing its evaluation. For example, the evaluation of

```
(cons 'a
   (call/cc
      (lambda (k)
         (cons 'b (k 'c)))))
```

returns `(a . c)`, for the application of `c` to `k` causes the `call/cc` application to return `c` immediately, without ever invoking the inner `cons`.

The control context of a continuation application is discarded unless it has been saved with another `call/cc`. The storage space of discarded control information, as well as continuations and other objects of computation to which there are no longer any references, can be reclaimed by a garbage collector.

3. A taxonomy for embeddings

A number of logic programming embeddings, of widely differing character, have been reported in the literature. The following taxonomy of embeddings attempts to clarify the varying degrees of embedding found in these systems. It may also be of use in classifying other embeddings.

In general, the term *embedding* refers to an implementation in which the embedded language benefits from the programming environment of the embedding, or implementation, language. These benefits may simply be such facilities as structure editors and memory management [25]. *Functional embeddings* also allow functions in the embedded and embedding language to call one another conveniently, as in QLOG [18] and POPLOG [22].

Further embedding is achieved when the embedded and embedding languages share a common environment, so that identifier references in the embedded language refer directly

[3] Using this primitive we can define `catch`, a version of Landin's *J* operator [20, 24, 29]: `(catch` *id exp*`)` ≡ `(call/cc (lambda (`*id*`)` *exp*`))`.

to identifier bindings in the embedding language. Such an *environment embedding* may be obtained only when the embedded language is compiled into the embedding language, and the embedding language supports first-class functions (or *closures*).[4] It may be advantageous to implement some segments of a large program in the embedded language and others in the embedding language, so that the facilities of each language may be used where most appropriate. An environment embedding provides convenient and efficient transfer of information between these segments. Of course care must be taken to ensure that each segment respects the semantics of the other; in particular, assignment to logic variables would risk violation of logic programming semantics.

Finally, a *complete embedding* is obtained with an environment embedding in which the control context may be obtained at any stage in the computation, and then invoked at any future time in order to return to that context. This is possible with an embedding language, such as Scheme, that supports first-class continuations. However, special care is required to ensure that when a continuation is invoked the values of logic variables are properly restored to their values at the time the continuation was created. A complete embedding provides a semantically consonant union of both the environment and control contexts of the embedding and embedded languages.

The environment embeddings of logic programming into Scheme by Felleisen [8] and Srivastava, Oxley and Srivastava [28] fail to be complete embeddings. Though first class continuations are shared by the embedded and embedding language, they do not restore logic variable bindings when invoked. Thus there are continuation invocations which, if performed in these embeddings, will violate the semantics of logic programming.

The problem of restoring logic variable values is related to the problem of restoring the values of dynamic (or fluid) bindings. The *state-space* model of control was originally developed to solve the dynamic binding problem and to implement a generalization of unwind-protect in the presence of first-class continuations [2, 11, 13]. The central result of the present paper is a new form of state-space model for the maintenance of logic variable bindings in the event of any meaningful continuation invocation.[5]

4. An environment embedding of logic programming

In this section we develop an environment embedding of logic programming in Scheme. Though the embedded and embedding languages share a common control environment, this fails to be a complete embedding because no attempt is made to restore logic variable bindings upon continuation invocation. The primary purpose of this embedding is to provide a framework within which to present a complete embedding that avoids this problem; however, it is hoped that the simple structure of this embedding will be of some interest in its own right.

---

[4] **Environments contained in closures must be heap allocated to allow indefinite extent of environment bindings. Komorowski states that Prolog's variable binding and control mechanisms require stack structures distinct from those of the embedding language [18]. This is true only when the embedding language lacks first-class functions and continuations.**

[5] **We shall see that certain continuation invocations are not meaningful, for they are inconsistent with logic programming semantics.**

	⟨pred⟩	::= (⟨relation⟩ ⟨term⟩*)	predication form
	⟨clause⟩	::= `(logic-lambda (⟨id⟩*) (⟨term⟩*) ⟨pred⟩*)`	clause form
	⟨term⟩	::= *Scheme expressions with value in* T	term form
	⟨relation⟩	::= *Scheme expressions with value in* R	relation form
	D	= *Scheme values (except references)*	denotable values
`lvar` $\in$	V	= *ref*(unbound) | *ref*(D) | *ref*(V)	logic variables
`term` $\in$	T	$= D \mid V \mid T \times T$	terms
`fk` $\in$	K_f	= *cont*()	failure continuations
`sk` $\in$	K_s	= *cont*(K_f)	success continuations
`rel` $\in$	R	$= T^* \rightarrow P$	relations
`pred` $\in$	P	$= [K_f] \rightarrow K_f$	predications
`clause` $\in$	C	$= [T, K_f] \rightarrow K_f$	clauses
	alt	$: C^* \rightarrow R$	alternation function
	seq	$: P^* \rightarrow P$	sequencing function

Figure 3. Syntax, types and functionality of a logic embedding.

We use non-structure sharing [15, 16, 22]: a logic variable is represented as a reference (pointer), which may refer either to a value, to another logic variable with which it has been unified, or to a unique value denoting 'unbound'. See Figure 3 for a logic variable domain equation, as well as syntax and type definitions for other elements of this embedding. (The symbols on the left indicate the standard identifier names that will be used for objects of their type in the program segments that follow.) For efficiency, invisible pointers may be used instead of references [15, 28]. We name the logic variable constructor, selector, binder, type predicate and bound predicate functions `lvar`, `lval`, `bind-lvar!`, `lvar?` and `bound-lvar?`, respectively.

We take a *substitution* to be simply a set of logic variables. `Extend-subst` is passed a substitution, an unbound logic variable and a value. It binds the logic variable to the value, and returns a substitution extended with this variable. `Unbind!` takes a substitution and unbinds each of its logic variables (by assigning their reference the unbound value).

A ⟨term⟩ may be any Scheme expression; however its value is interpreted as a structure built of pairs, logic variables and literals. The function `unify` (Figure 4) unifies two terms. If unification succeeds, a substitution of all logic variable bindings created by the unification is returned. If it fails, any bindings created up to the point of failure are undone, and then the failure continuation `fk` passed to `unify` is invoked. (The occurs check has been omitted for simplicity.)

Failure continuations are represented as functions of no arguments, since no information need be passed when failing. They may be obtained with the function `call-with-current-failure-continuation`, abbreviated `call/cfc`:

```
(define unify
   (lambda (term1 term2 fk)
      (letrec
         ([bind (lambda (var term subst)
                  (cond [(bound-lvar? var) (unify1 (lval var) term subst)]
                        [else (extend-subst subst var term)]))]
          [unify1 (lambda (term1 term2 subst)
                    (cond [(eq? term1 term2) subst]
                          [(lvar? term1) (bind term1 term2 subst)]
                          [(lvar? term2) (bind term2 term1 subst)]
                          [(and (pair? term1) (pair? term2))
                           (unify1 (car term1) (car term2)
                              (unify1 (cdr term1) (cdr term2) subst))]
                          [else (unbind! subst) (fk)]))])
         (unify1 term1 term2 null-subst))))
```

Figure 4. Unification procedure.

```
(define call/cfc
   (lambda (f)
      (call/cc
         (lambda (k)
            (f (lambda () (k any)))))))).
```

`Any` indicates an irrelevant value that will be ignored; failure continuations are essentially command continuations. Initially, we represent success continuations simply as continuations provided by the primitive `call/cc` function.

The convention for the use of success and failure continuations is critical to the structure of a logic programming embedding. We opt for 'upward failure continuations' [15], in the manner of Felleisen [8]: in the event of success, a failure continuation is passed upward by either returning it from a function or invoking a success continuation with it. Subsequent invocation of this failure continuation causes backtracking to the point of success. Other alternatives include passing success continuations to the theorem prover and returning in the event of failure [5, 22], passing a continuation which is always invoked with true or false, indicating success or failure [28], passing separate success and failure continuations [17, 32], and representing the failure continuation as a stream of frames [1, 5]. Continuations may be represented either as data structures [1, 5], closures [22, 32], or encapsulations of the system control context [8, 17, 28], as we do here.

Predications (or *atoms*) are represented as Scheme applications in which the function position evaluates to a *relation* and the arguments evaluate to terms. When the relation is applied to the terms, a function is returned that takes a failure continuation. If the terms fail to satisfy the relation, this failure continuation is invoked. Otherwise, the predication returns a new failure continuation that, when invoked, attempts to satisfy the relation in a new way (and invokes the original failure continuation if there are no more ways).

```
(define alt
   (lambda clause-values-list
      (lambda term
         (lambda (fk)
            (call/cc
               (lambda (sk)
                  (do ([cvl clause-values-list (cdr cvl)])
                      [(null? cvl) (fk)]
                      (call/cfc
                         (lambda (fk)
                            (sk ((car cvl) term fk))))))))))))

(define seq
   (lambda preds
      (lambda (fk)
         (do ([preds preds (cdr preds)]
              [pred-fk fk ((car preds) pred-fk)])
             [(null? preds) pred-fk]))))
```

Figure 5. Alternation and sequencing functions.

For example, we express the Prolog predication `member(A,[1|B])` as `(member A (list 1 B))`. Assume the value of identifier `A` is an unbound logic variable and the value of `B` is a logic variable bound to 2. (By convention, Scheme identifiers that are bound to logic variables begin with capital letters. Also, when no ambiguity results we refer to logic variables by the name of the associated Scheme identifier; for example, "`A` is unbound".) Upon receiving the `A` and `(list 1 B)` terms, the member relation returns a predication which is passed a failure continuation, κ_{f1}. A new failure continuation, κ_{f2}, is then returned with `A` bound to `1`. When κ_{f2} is invoked, the application will return (to its original continuation) a third failure continuation κ_{f3}, leaving `A` bound to (the logic variable) `B`. Invoking κ_{f3} results in `A` being unbound and κ_{f1} being invoked.

A relation is formed by passing *clause values* to the `alt` function (Figure 5).[6] A new failure continuation is obtained for each clause invocation using `call/cfc`. The clause either returns a failure continuation (which is passed to the success continuation, `sk`, of the predication that invoked the relation) or it invokes the failure continuation. The failure continuation of the last clause is the failure continuation of the predication.[7]

Because clauses may introduce new logic variables whose scope is local to the clause, the operation for creating clauses must be a special form (it cannot be a function). By analogy with `lambda`, the standard special form that evaluates to a function, we call the

[6] A mechanism for maintaining a data base of relations has been added to this embedding, but in this paper we are not concerned with such issues.

[7] `Alt` may be refined somewhat by replacing the `do` termination clause with `[(null? (cdr clauses)) ((car clauses) term fk)]`. This is similar to evlis tail recursion [30].

form for creating clauses `logic-lambda`. It is implemented as a syntactic extension (macro) that transforms an expression of the form

```
(logic-lambda (id_1 ... id_k) (term_1 ... term_m) pred_1 ... pred_n)
```

into an expression of the form

```
(lambda (term fk)
   (let ([id_1 (lvar unbound)] ... [id_k (lvar unbound)])
      (let ([subst (unify term (list term_1 ... term_m) fk)])
         (logic-bind (seq pred_1 ... pred_n) subst fk)))).
```

This expression evaluates to a clause that takes a term and a failure continuation. The *term* and *pred* expressions are evaluated in an extended environment which associates $id_1, \ldots, id_k$ with new logic variables that are initially unbound. A list of the clause terms is unified with the argument term, returning a new substitution if successful. The predications of the clause are then called sequentially under control of the `seq` function (Figure 5). Each predication receives a failure continuation. If successful, it returns a new failure continuation. This failure continuation is then passed to the next predication or, in the case of the last predication, returned as the result of the clause. When a failure continuation returned as the result of the clause is invoked, it is necessary to unbind the logic variables introduced by the clause. This is managed by the `logic-bind` function:

```
(define logic-bind
   (lambda (pred subst fk)
      (pred (lambda () (unbind! subst) (fk))))).
```

This unbinding could be performed more efficiently using a trail [4, 16]. However, in the next section we extend logic-bind to manage the *rebinding* of logic variables in the event control is transferred back into clauses via *success* continuation invocation. The trail mechanism does not suffice in this more general context.

The Prolog relation

```
append([],Y,Y).
append([H|T],Y,[H|Z]) :- append(T,Y,Z).
```

is expressed in this embedding as

```
(define append
   (relation
      [('() Y Y)]
      [((cons H T) Y (cons H Z)) (append T Y Z)]))
```

where `relation` is a simple syntactic extension that, in the above case, expands into

```
(define append
   (alt (logic-lambda (Y)
          ('() Y Y))
        (logic-lambda (H T Y Z)
          ((cons H T) Y (cons H Z)) (append T Y Z)))))
```

Each rule is expanded into a `logic-lambda` expression that contains a list of the logic identifiers used in the rule, the pattern list, and finally the predications of the rule. Relations in this embedding are simply Scheme functions returned by `alt`.

We define the traditional Prolog *fail* and *is* operations as simple examples. *Fail* is simply a predication that always fails. This is accomplished by immediately invoking the failure continuation passed to the predication. Thus we define `fail` to be the function `(lambda (pred-fk) (pred-fk))`.

`Is` predications are of the form `(is` *var function argument* ...`)`, where *var* evaluates to a logic variable, and *function* is a Scheme function that is to be passed the given arguments. First, any logic variables in the arguments are replaced by their values. If any of these logic variables are unbound, then the `is` predication fails. Otherwise, the function is applied to the arguments, obtaining some answer *ans*. If *var* is an unbound logic variable, then it is bound to *ans* and the predication succeeds; otherwise, if *var*'s value is the same as *ans*, the predication succeeds. In all other cases, it fails. This is accomplished by the Scheme function

```
(define is
   (lambda (var function . args)
      (lambda (fk)
         (let ([args (lval* args)])
            (for-each (lambda (arg) (cond [(lvar? arg) (fk)])) args)
            (let ([ans (apply function args)])
               (cond
                  [(not (bound-lvar? var)) (bind-lvar! var ans) fk]
                  [(equal? (lval* var) ans) fk]
                  [else (fk)])))))))
```

where `lval*` recursively copies a list structure replacing logic variables by their values.

Prolog's *cut* may be incorporated into this embedding by extending `alt` so that it fluidly (or dynamically) binds `fk`. Each time `alt` is called, a new binding—accessible as `(fluid fk)`—is established which records the failure continuation at the time of the `alt` call. This binding remains in force until the `alt` call returns, except during other calls to `alt`. Cut may then be defined as `(lambda (pred-fk) (fluid fk))`; that is, the cut predication receives the current failure continuation `pred-fk` and immediately succeeds by returning the failure continuation of the entire alternation. Since no reference to the current failure continuation is maintained, its storage space may be reclaimed by a garbage collector. Relations might be defined with different versions of `alt`, where each version binds `fk` to a distinct fluid identifier. This would allow a predication in one relation to 'cut' another relation. The state-space model presented in the next section can be extended in a straightforward way to adjust fluid bindings when continuations are invoked [11, 13].

5. A state-space model for logic continuations

In this section we deal with the special difficulty presented by first-class continuations in the context of logic programming: when invoking a continuation it may be necessary to modify the bindings of logic variables. Of course when a failure continuation is invoked it

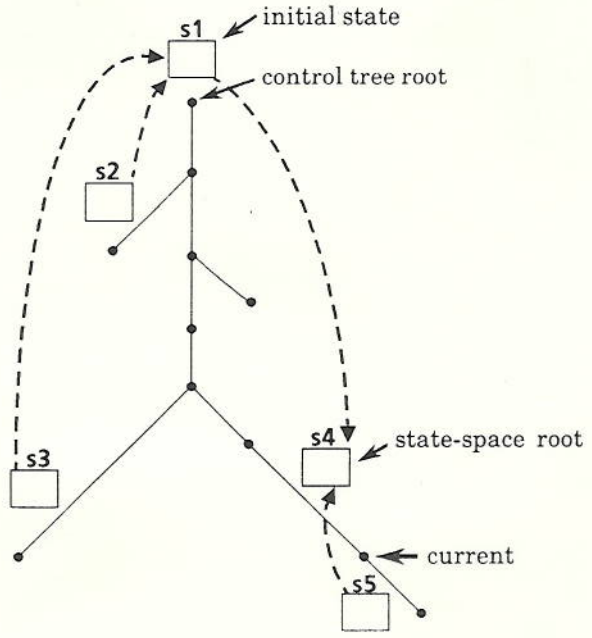

Figure 6. Control tree with state-space.

may be necessary to unbind certain logic variables. But upon invoking a 'lateral' success continuation, as when jumping from one leaf to another of the control tree, it is also necessary to restore the values of logic variables accessible at the destination point to the values they had at the time that point was marked. To manage this unbinding and rebinding of logic variables we modify the state-space model of control.

A *state-space* is a tree with one node (or *state*) for each logic-bind (unification). Each time the state-space is extended, the new state becomes the root of the state-space. Though the control and state-space trees are distinct, for purposes of understanding it is helpful to associate each control tree node with the state which was the root at the time the node was created. See Figure 6, in which boxes represent states and dots represent control tree nodes. The state associated with each node is the nearest one above it. State-space edges point toward the state-space root. (The direction of control tree edges is not indicated, for they always point up.) The ancestors of any state are those states corresponding to nodes that would be reached by an (undirected) traversal of the control tree from a node corresponding to the given state to the current node. If control is transferred between control tree nodes via a return or continuation invocation, and the destination node is associated with a state that is not the current root, the state-space is modified so that this state becomes the new root. This is accomplished simply by reversing the direction of the edges leaving the destination state and its ancestors. We say a state is *active* if it is associated with the current control tree node or with an ancestor of the current node. In Figure 6, states s1 and s4 are active and the remaining states are inactive.

Each state has an associated status, substitution and control link. The status is either *in*, *out* or *returned*: *in* indicates that the state is active; *returned* indicates the state is

inactive because control returned via a successful resolution; and *out* indicates the state is inactive because control passed out of the descendent subtree by invoking a success or failure continuation. Since new states are immediately active, their status is initially *in*.

The substitution is provided by the unification performed just prior to the logic-bind operation that created the state. When a state becomes inactive via a continuation invocation, it is necessary to save the values of the substitution's logic variables so that these values may be restored whenever the state becomes active again. This is accomplished by extending the substitution with local storage for each of its logic variables. The `swap!` operation exchanges the current value and the locally saved value of each logic variable in the substitution. The saved value is initially unbound.

The control link points to the state which was the state-space root at the time the present state was created. This indicates the direction of the control tree root, which is required for a technical reason discussed below.

See Figure 7 for examples of state-space transitions. The current node is indicated by the disembodied arrowhead. Dashed boxes indicate logic variables. Each solid box indicating a state contains the state's status and a smaller box for its substitution variable (if it has one). Saved values are indicated in the variable boxes. Arrows point from these boxes to their associated logic variables. Empty saved value or logic variable boxes indicate unbound values.

In [a] the computation has introduced logic variables X and Y while proceeding from context A to B, and has then bound Y to c with a unification while proceeding from B to C. The logic-bind associated with the unification has extended the state-space with a state s2 whose substitution records the binding of Y and a saved value which is unbound. In [b] control has returned to B via a successful resolution, but context C has been retained by a failure continuation `fk`. The status of s2 is now *returned* and s1 is the new state-space root.

From B the computation next proceeds to D, binding X to a along the way. This binding is recorded in the substitution of a new state, s3, which becomes the root; see Figure 7[c]. The context D is then recorded by a success continuation `sk`, and then a continuation (not indicated) that is associated with choice point B is invoked, resulting in configuration [d]. X is now unbound and its old value, a, has been saved in s3. Computation then proceeds to E, binding X to b along the way, to obtain configuration [e].

A lateral control transfer from E to D is then performed by invoking `sk`. s3 then becomes the root, requiring the reversal of edges s1-s3 and s1-s4. The state-space mechanism then traverses the path from s4 to s3, visiting s4, s1 and s3. When state s4 is visited, its status is changed to *out* and the value b of X is recorded in s4. Nothing happens when state s1 is visited, since it remains active. When state s3 is visited, its status is changed to *in* and X is bound to the value saved in s3's substitution; see Figure 7[f]. Finally, invoking `fk` results in configuration [g].

We represent the state-space of a computation by a globally bound object that responds to messages. The current root is returned in response to the message `root`. The other four possible messages, `extend!`, `return!`, `fail!` and `reroot!`, cause state-space transitions. (See Figure 8.) `Extend!` is used when control enters a logic-bind; it adds a new

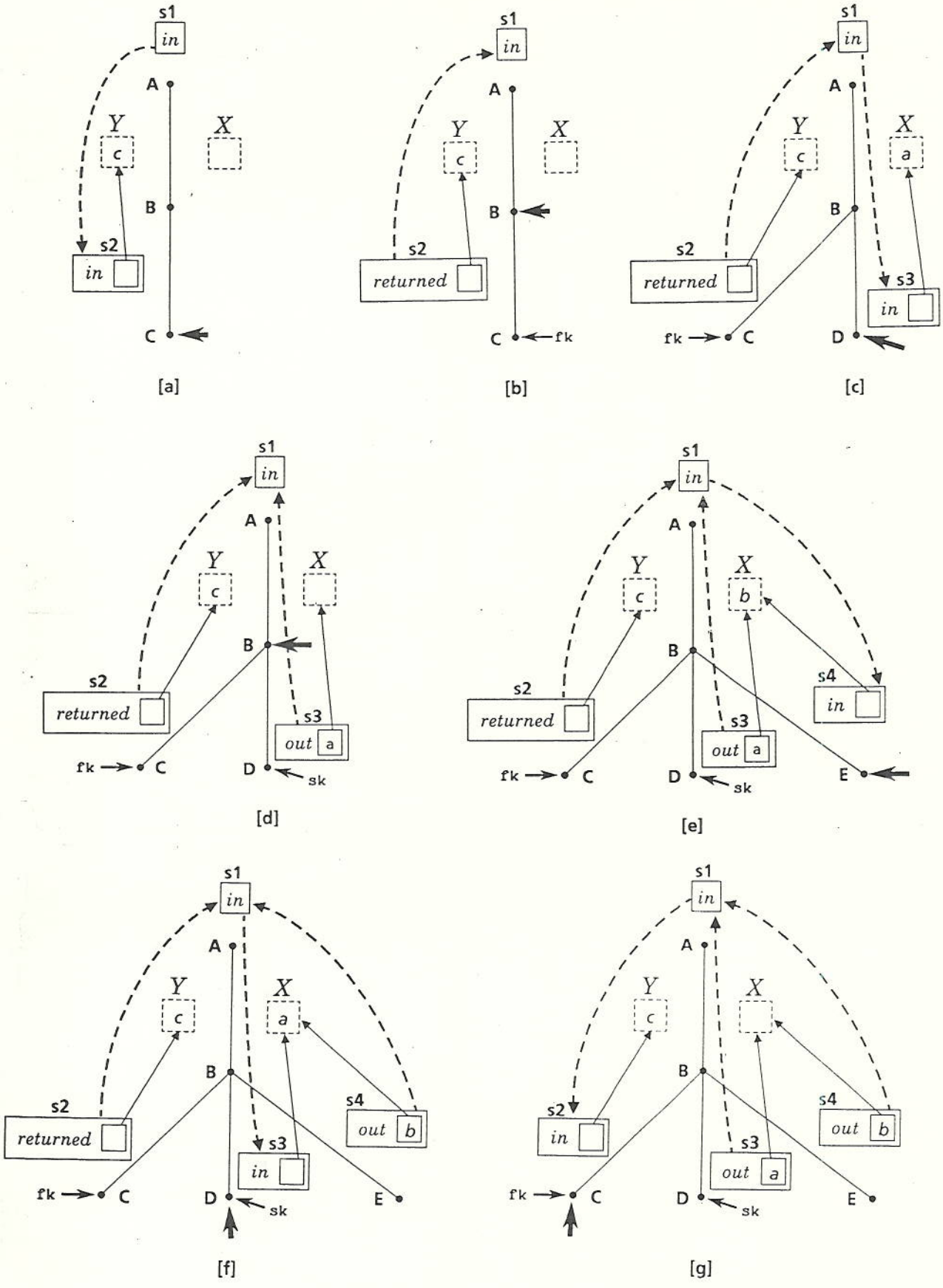

Figure 7. Examples of state-space transformations.

```
(define state-space
   (let ([root (list (lambda (msg) do-nothing))])
      (lambda (msg)
         (case msg
            [(root) root]
            [(return! fail! extend! reroot!)
             (lambda (new-state)
                (case msg
                   [(extend!) (set-cdr! root new-state)]
                   [(return! fail! reroot!)
                    (reverse! new-state)
                    (for-each (lambda (x) (x msg)) root)])
                    (set! root new-state))])))))
```

Figure 8. Initial logic state-space.

state to the space that maintains the substitution being bound, and makes it the root. `Return!` is used when control returns successfully from a logic-bind; it restores the root to the state that was the root when the logic-bind was entered. The new version of logic-bind is

```
(define logic-bind
   (lambda (pred subst fk)
      (let ([state (state-space 'root)])
         ((state-space 'extend!) (make-state subst))
         (let ([ans (pred (lambda () (unbind! subst) (fk)))])
            ((state-space 'return!) state)
            ans)))).
```

The `fail!` and `reroot!` state-space messages are used when failure or success continuations, respectively, are invoked; they make the destination continuation the root and may adjust some logic variable values, as will be described presently. The new versions of `call/cc` and `call/cfc` are

```
(define call/cc
   (lambda (f)
      (prim-call/cc
         (lambda (k)
            (let ([state (state-space 'root)])
               (f (lambda (v)
                     ((state-space 'reroot!) state)
                     (k v))))))))

(define call/cfc
   (lambda (f)
      (prim-call/cc
         (lambda (k)
            (let ([state (state-space 'root)])
               (f (lambda ()
                     ((state-space 'fail!) state)
                     (k any))))))))
```

where `prim-call/cc` is the original `call/cc` function. These are the only operations on the state-space; the user may not manipulate it directly.[8]

States are represented internally as cons cells; see Figure 9. The car of a state cell is an object that responds to the state transition messages, while the cdr is a link that points to the nearest state in the direction of the root, or nil if the state is the root. Hence each state cell may be viewed as a list whose tail is a list of its ancestors and whose last element is the root. With each of the state transition messages a new state must be passed, which becomes the root. In the `return!`, `fail!` and `reroot!` cases, the state links are adjusted to point to the new root by simply reversing the links in the list headed by the new state. Think of picking up the tree by the new state and giving it a good shake so that all paths lead to the new root. Each of the objects associated with states on the reversed list (which now begins with the old root) is then passed the state-space transition message and will modify its local state as required.

When a state object receives a transition message, its response will depend on both its current status and the type of message. There are three statuses and three types of messages, so there are nine possibilities to consider.

`Return!` is used only by logic-bind to restore the state-space root to the state s_i in which the logic-bind was entered. Thus the `return!` message is received by only two states: s_i and the state s_j created by the logic-bind. The status of both s_i and s_j will be *in*, and the only effect of the `return!` is to change the status of s_j to *returned*. (At the time the `return!` message is sent, s_i is already the root. s_i detects this by noticing its cdr link is nil, and thereby avoids changing its status.)

The status of a state is *in* if and only if it is active. When a state is created it becomes active and its state is initially *in*. Control can only leave the control subtree associated with

[8] For convenience we define the state-space globally in this paper, but in a production system its scope should be restricted to the `call/cc`, `call/cfc` and `logic-bind` functions.

```
(define make-state
   (lambda (subst)
      (let ([status 'in]
            [control-link (state-space 'root)])
         (rec local-state
            (list
               (lambda (msg)
                  (case msg
                     [(return!)
                      (cond [(not (null? (cdr local-state)))
                             (set! status 'returned)])]
                     [(fail!)
                      (case status
                         [(returned) (set! status 'in)]
                         [(in) (cond [(eq? control-link (cdr local-state))
                                      (set! status 'out)
                                      (swap! subst)])]
                         [(out) (error "can't fail when out")])]
                     [(reroot!)
                      (case status
                         [(returned) (error "can't reroot when returned")]
                         [(in) (cond [(eq? control-link (cdr local-state))
                                      (set! status 'out)
                                      (swap! subst)])]
                         [(out) (set! status 'in)
                                (swap! subst)])])))))))
```

Figure 9. `Make-state` for a logic state-space.

a state in one of two ways: a logic-bind return, which sets the state's status to *returned*, or invocation of a success or failure continuation, in which case the `reroot!` or `fail!` message is sent to the state. Since control is being transferred out of the node (the state-space link already points toward the new root), the control link of the state will be equal to the state-space link (cdr of `local-state`), and the status is set to *out*.

For control to reenter its control subtree, a state must receive a `reroot!` message while its status is *out* or a `fail!` message while its status is *returned*. In both cases the status is set to *in*. It is not permissible for a state with status *out* to be reentered by invoking a failure continuation, or for a state with status *returned* to be reentered by invoking a success continuation.

The only case in which a state can be visited without control leaving its subtree is when control is passing, via a success or failure continuation invocation, from one of its subtrees to another. In this case the control and state-space links will be unequal and the status will not be altered.

No values are saved or restored in transitions to and from the *returned* status. However, when the status is changed from *in* to *out* the substitution variable values are saved, and they are restored when the status changes from *out* back to *in*. The state-space mechanism exists solely to implement this operation and to ensure that its integrity is maintained. This integrity would be violated if a failure continuation were used to reenter a state with status *out*, or if a state with status *returned* were reentered via a success continuation.

There are several ways in which a complete embedding may be used to mix logic and imperative programming in the solution of a problem. For example, alternatives to a depth-first search search strategy may be obtained by modifying the `alt` function so that a queue of success continuations is maintained representing partially explored alternatives. The new version of `alt` could be installed in place of the original version, making the new search strategy pervasive, or it could be used only in the definition of selected relations. These relations might then employ breadth-first search for exploration of their alternate clauses, while depth-first search was used at other times. Alternatively, selected relations (or clauses of relations) could be coded as imperative procedures, making unrestricted use of continuations. These custom relations could then be invoked in the usual fashion by logic programs.

6. Efficiency considerations

It is presumed that a garbage collection mechanism reclaims heap allocated storage when it is no longer accessible, and that the control tree is heap allocated (at least when it branches). This is required for first-class continuations. If the state-space states are also heap allocated, their storage will be reclaimed automatically when they are no longer accessible. This follows because the only references to states outside of the state-space itself are in control frames associated with `logic-binds` and success and failure continuations. As a result, if an operation such as Prolog's cut causes the sole reference to a continuation to be abandoned, the storage associated with both the continuation and any states associated with the continuation will be reclaimed. Maneuvers, such as tail recursion optimization, for further reducing storage requirements should also be applicable (though they are not used in the simple embedding of this paper).

The code presented here was designed for clarity, not speed, and many improvements are possible. For example, the `alt`, `seq` and `unify` function applications could be compiled in-line [8], the state-space could be built up of data structures instead of procedures, and the `case` dispatchs of the `state-space` and `make-state` functions could be avoided by in-line coding of the state-space operations in `logic-bind`, `call/cc` and `call/cfc`. The initial state is not essential, since it has no associated substitution. It could be eliminated by adding a special case to the `extend` operation.

Our main efficiency concern is the price paid for the state-space mechanism in an optimized implementation. Though a definitive answer awaits the development of such an implementation, a few observations are appropriate at this time. The logic state-space may be viewed as a generalization of the trail mechanism: the state-space is capable of rebinding as well as unbinding logic variables, and may take on a tree structure, rather than being strictly linear. This requires additional run time tag checking and heap, rather

than stack, allocation. However, these additional costs may be avoided much of the time. For example, if it can be proved that no `call/cc` or `call/cfc` operations will be performed during a particular phase of execution, then the system is free to revert to the traditional stack allocated trail. It even seems possible for a system to routinely use a trail, and only convert the trail information into an extension of the state-space when a continuation is actually obtained. The state-space overhead is then incurred only when the user requires its generality. A related 'pay as you go' approach is used by some Scheme implementations that stack allocate control information until `call/cc` is invoked, at which time the stack is copied to the heap. [3, 19, 21]

In many cases where the generality of a logic state-space would be used, the alternatives are also expensive and likely to be less efficient and more cumbersome than employing a well implemented logic state-space. For example, when non-blind backtracking is needed, the alternatives are repeating part of a computation or explicitly saving and restoring necessary information. (This is analogous to the explicit stack management required to simulate recursion in a non-recursive language.) Other approaches to increasing control flexibility, such as LOGLISP's breadth-first search parameters [25], also have overhead and are less general.

7. Conclusion

A principal advantage of non-procedural programming languages, such as Prolog, is that they avoid the necessity of repeatedly specifying commonly occurring patterns of control. This is done by providing a complex default control mechanism, such as Prolog's depth-first search. Problems arise when variations on the default control mechanism are required. Some variations may be accommodated by auxiliary control mechanisms, such as Prolog's cut, but other variations may be difficult or impossible to achieve with such specialized mechanisms.

Much of the power of non-procedural program specification may be provided along with the ability to obtain non-standard control behavior on occasion. This is accomplished by embedding the non-procedural mechanism in a traditional procedural language whose control mechanism (principally procedure call) is simple and well understood. Greatest flexibility is obtained when the embedding language makes continuations available as first-class objects of computation.

Special precautions must be taken in contexts, such as logic programming, in which changes may be made to the environment of a control context that must be accounted for when control is returned to the control context via a continuation invocation. We have shown how the state-space model of control may be extended to provide first-class continuations that account for changes in logic variable bindings. Such 'logic continuations' may then be used to obtain greater control flexibility while allowing much of a program to be expressed with logic programming.

Acknowledgements: We thank Matthias Felleisen, Dan Friedman, Peter Williams and anonymous referees for their detailed comments on this paper. This work was supported by the National Science Foundation under grant number DCR 85-01277.

References

[1] Abelson, H., and Sussman, G.J., with Sussman, J., *Structure and Interpretation of Computer Programs,* MIT Press, 1985.

[2] Baker, H.G., Jr., Shallow Binding in Lisp 1.5, *C. ACM,* **21**:565–569 (1978).

[3] Bartley, D.H., and Jensen, J.C., The Implementation of PC Scheme, *Proc. of the 1986 ACM Conference on LISP and Functional Programming,* pp. 86–93.

[4] Bruynooghe, M., The memory management of PROLOG implementations, in: K.L. Clark and S.-A. Tärnlund (Eds.), *Logic Programming,* Academic Press, New York, pp. 83–98 (1982).

[5] Carlsson, M., On implementing Prolog in Functional Programming, *New Generation Computing,* **2**:347–359 (1984).

[6] Clocksin, W.F. and Mellish, C.S., *Programming in Prolog, Second Edition,* Springer-Verlag, New York, 1984.

[7] Dybvig, R.K., *The Scheme Programming Language,* Prentice-Hall, 1987.

[8] Felleisen, M., Transliterating Prolog into Scheme, Computer Science Department Technical Report No. 182, Indiana University, Bloomington, Indiana, 1985.

[9] Felleisen, M., and Friedman, D.P., Control operators, the SECD-machine, and the λ-calculus, *Formal description of programming concepts III,* North-Holland, Amsterdam, 1986, to appear.

[10] Felleisen, M., Friedman, D.P., Kohlbecker, E., and Duba, B., Reasoning with continuations, *Symp. on Logic in Computer Science,* Cambridge, Mass., pp. 131–141 (1986).

[11] Friedman, D.P., and Haynes, C.T., Constraining control, *Conf. Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages,* pp. 245–254 (1985), revised in Computer Science Department Technical Report No. 183, Indiana University, Bloomington, Indiana, 1985.

[12] Friedman, D.P., Haynes, C.T. and Kohlbecker, E., Programming with continuations, in: P. Pepper (ed.), *Program Transformation and Programming Environments,* Springer-Verlag, New York, pages 263–274 (1984).

[13] Hanson, C., and Lamping, J., Dynamic Binding in Scheme, unpublished manuscript, 1984.

[14] Haynes, C.T., and Friedman, D.P., Engines build process abstractions, *Conf. Record of the 1984 ACM Symp. on Lisp and Functional Programming,* pp. 18–24 (1984).

[15] Kahn, K.M., and Carlsson, M., How to implement Prolog on a LISP Machine, in: J.A. Campbell (ed.), *Implementations of PROLOG,* Halstead Press, New York, pp. 117–134 (1984).

[16] Kluźniak, F., and Szpakowicz, S., *Prolog for Programmers,* Academic Press, 1985.

[17] Kohlbecker, E., eu-Prolog, Computer Science Department Technical Report No. 155, Indiana University, Bloomington, Indiana, 1984.

[18] Komorowski, H.J., QLOG—the programming environment for Prolog, in: K.L. Clark and S.-A. Tärnlund (Eds.), *Logic Programming,* Academic Press, New York, pp. 315–324 (1982).

[19] Kranz, D., *et al.,* ORBIT: An optimizing compiler for Scheme, *Proc. SIGPLAN '86 Symp. on Compiler Construction,* in *SIGPLAN Notices,* **21**:234–241 (1986).

[20] Landin, P. A correspondence between ALGOL 60 and Church's lambda notation, *C. ACM,* **8**:89–101 and 158–165 (1965).

[21] McDermot, D., An efficient environment allocation scheme in an interpreter for a lexically-scoped LISP, *Conf. Record of the 1980 LISP Conference,* ACM Order No. 552800, pp. 154–162.

[22] Mellish, C., and Hardy, S., Integrating Prolog in the POPLOG environment, in: J.A. Campbell (ed.), *Implementations of PROLOG,* Halstead Press, New York, pp. 147–162 (1984).

[23] Rees, J., and Clinger, W. (Eds.), Revised^3 Report on the Algorithmic Language Scheme, *SIGPLAN Notices,* October, 1986).

[24] Reynolds, J.C., Definitional interpreters for higher-order programming languages, *Proceedings of the 25th ACM National Conference,* pp. 717–740 (1972).

[25] Robinson, J.A., and Sibert, E.E., LOGLISP: motivation, design and implementation, in: K.L. Clark and S.-A. Tärnlund (Eds.), *Logic Programming,* Academic Press, New York, pp. 299–314 (1982).

[26] Shapiro, E., review of *Foundations of logic programming* by J.W. LLoyd, *Computing Reviews,* **27**:384–386 (1986).

[27] Sussman, Gerald Jay, and Drew Vincent McDermott, "From PLANNER to CONNIVER—A genetic approach", *Proceedings of Joint Computer Conference 41, part II,* AFIPS Press, NJ, (1973) pages 1171–1179.

[28] Srivastava, A., Oxley, D., and Srivastava, D., An(other) integration of logic and functional programming, *Proceedings of The IEEE Symposium on Logic Programming,* pp. 254–260 (1985).

[29] Sussman, G.J., and Steele, G.L., Jr., Scheme: an interpreter for extended lambda calculus", Artificial Intelligence Memo No. 349, MIT, Cambridge, Massachusetts, 1975.

[30] Wand, M., Continuation-based program transformation strategies, *J. ACM,* **27**:164–180 (1980).

[31] Wand, M., Continuation-based multiprocessing, *Conf. Record of the 1980 LISP Conference,* ACM Order No. 552800, pp. 154–162.

[32] Wand, M., A semantic algebra for logic programming, Computer Science Department Technical Report No. 134, Indiana University, Bloomington, Indiana, 1983.