TECHNICAL REPORT NO. 293

# Continuation-Based Control Operators for Logic Programming

by

Richard M. Salter and Christopher T. Haynes

October, 1989

COMPUTER SCIENCE DEPARTMENT
INDIANA UNIVERSITY

Bloomington, Indiana 47405-4101

# Continuation-based Control Operators for Logic Programming*

Richard M. Salter
*Oberlin College*†

Christopher T. Haynes
*Indiana University*‡

## Abstract

Logic programming languages typically combine declarative specification with inference techniques that utilize a depth-first control discipline. Applications that require other forms of control behavior are often difficult to express in these languages. In this paper we present the high-level extra-logical control operators of a Prolog extension called Conlog. These operators allow multi-tasking and variations on breadth-first search to be conveniently expressed. Examples illustrating the use of these operators include Algorithm A* and the samefringe problem.

## 1. Introduction

Logic programming languages exist both for nonprocedural program specification and as programming tools for problems requiring extensive search. Unfortunately this dichotomy weakens most logic programming languages so that it is difficult to express complex search behavior. Prolog, for example, is a poor choice for problems in artificial intelligence requiring intricate heuristic-based search strategies.

Prolog supports declarative program specification via Horn clauses, with an underlying computation procedure founded on the resolution inference rule. In Prolog, as in most logic programming languages, the inference algorithm is implemented with a depth-first backtracking control strategy. This strategy is chosen for efficiency, though it comes at the cost of resolution completeness. Since Prolog's default control strategy is depth-first search, it is well suited for applications involving this type of search. Prolog also provides the extra-logical control operator cut (`!`), which trims the search tree by discarding a portion of the backtrack history. Cut is widely used, but it provides only a limited ability to modify control. Prolog may be used to solve problems requiring alternative control behavior, such as variations on breadth-first search, but only by using techniques such as dynamic program modification (asserting and retracting program rules) and meta-programming (interpretation) that are often inefficient and obscure.

* This research was supported by NSF grant CCR-8702117.

† Computer Science Program, Oberlin, Ohio 44074; rms@occs.oberlin.edu; [216]775-8095.

‡ Computer Science Department, Indiana 47405; chaynes@iuvax.cs.indiana.edu; [812]855-3376.

Much previous effort has gone into extending the control capabilities of Prolog, as in IC-Prolog [4] and Epilog [15]. However, the control mechanisms introduced by these languages require the insertion of special syntax at critical points within the code. This results in programs that are difficult to understand and which may produce more data than necessary due to incorrect backtrack sequencing. [12]

In this paper we introduce the high- and intermediate-level control operators of Conlog, a Prolog extension that allows alternative patterns of control to be expressed in a direct and efficient manner. Our approach differs from previous proposals in that it is based on *first-class continuations.* Continuations are a general abstraction of sequential control. The continuation of a computation records the current control state—information that indicates how the computation is to be completed, which may be informally viewed as the "future" of the computation.

Continuations were first used in denotational semantics to express non-standard control behavior, such as jumps and exits. For example, the effect of a "goto" statement is to substitute the continuation of the goto label for the current continuation. In many languages a subcomputation has a single continuation representing normal flow of control. The control behavior of logic programs can be described by associating each goal evaluation with two continuations: a *success continuation* represents the forward movement of the program should the goal succeed; and a *failure continuation*, encapsulates the backtracking behavior that is appropriate should the goal fail. Success and failure continuations are appropriate semantic tools for describing any backtracking system, and are frequently used in denotational semantics of Prolog. [1,2,6]

In systems employing rigid backtracking disciplines, such as Prolog, there is little opportunity to manipulate the continuation contexts in which computations execute. An exception is the cut, which is used in Prolog to discard choice points that have been created since the current relation was entered. Cut replaces the current failure continuation with the failure continuation that was current when the relation in which the cut appears was called.

Control information is typically stack allocated. In such cases continuations cannot be used outside of their control context, for when control returns from a context the stack is popped and control information is lost. If provision is made for heap allocation of control information when necessary, continuations may have indefinite extent. Such *first-class* continuations allow multiple threads of control to be maintained indefinitely. With first-class continuations, control no longer is restricted to the linear form of a stack, but can branch to form a *control tree.* (In a control tree each branch represents a computation in progress, and not simply a potential computation as in a search tree.) First-class continuations may be used to obtain a variety of control behavior, including coroutines, multi-tasking, and breadth-first search. [8,9,16,19]

A few languages, most notably Scheme [5], make it possible for the programmer to *reify* first class continuations by making them data objects. Such continuations may be stored in data structures and invoked at any future time, even after the computation has left the dynamic context of the continuation.

The first set of control operators proposed here are of a high level, abstracting control paradigms that are useful in the context of logic programming. A single set of intermediate-level operators generalizing cut is also described, which can be used to implement the high-level operators. The intermediate-level operators may in turn be implemented using low-level operators that support reified first-class failure continuations and dynamic database protection. Companion papers define the semantics of the low level operators [10] and the implementation of Conlog's control operators. [11] The low-level operators are more general, but they are also more difficult to use. The control facilities that are most easily used in logic programming require complex operations involving first-class continuations, such as management of queues through database assertion and retraction.

The high-level operators fall into two categories: the *cleanup* operators (`cleanup` and `defer`) and the *multi* operators (`multi_step` and `multi_sweep`). The cleanup operators are used to express variations on breadth-first search, while the multi operators express multi-tasking (coroutine) control behavior. The intermediate-level operators are `mark_region`, `mark`, and `cut_to_mark`, which generalize cut by permitting the user to name arbitrary "cut points."

We have implemented Conlog via a syntactic preprocessor that produces a Scheme program. This embedding technique, described elsewhere [7,16], makes use of a state-space to restore shallow-bound logic variable bindings when continuations are invoked. These state-space techniques may also be used to obtain a direct (native code) implementation of an extended logic programming system supporting these operators.

Section 2 provides detailed descriptions and examples of the cleanup operators, while section 3 does the same for the multi operators. Section 4 describes the mark operators, and the implementation of the cleanup and multi operators in terms of the mark operators is in Section 5. Conclusions are drawn in the last section. The reader is expected to be familiar with Prolog. [3]

## 2. Alternative Search Strategies

In this section we present the "cleanup" operators of Conlog, which allow various combinations of depth-first and breadth-first search to be conveniently expressed. This is not possible in standard Prolog because of the persistence of success; that is, as long as a computation is succeeding along a particular path it will continue to pursue that path.

It is possible to explicitly interrupt the search by deliberately failing, but this results in backtracking that is "blind." It is impossible to later return to the point at which backtracking was initiated, since there is no way to record this point and its control context is not retained (backtracking pops some of the context off the control stack). The cleanup operators enable "non-blind" backtracking [18], in which control points are recorded prior to backtracking in such a way that it is possible to return to them at a future time.

Breadth-first search branch points are created with the operator `cleanup`. Invoking `cleanup`($G$), where $G$ is a goal, results in creation of a new first-in-first-out *cleanup queue* and the invocation of $G$ within the scope of this queue. The scope of the cleanup queue is *dynamic* in the sense that the queue is *active* until control leaves $G$ or the queue is shadowed by by another cleanup operation. Control may leave the dynamic context of $G$ with either success or failure, or via invocation of a first-class continuation representing control outside its dynamic context.

The `defer` operator causes a "pseudo-failure." The immediate control effect of `defer` is the same as `fail`: the current failure continuation is invoked to initiate backtracking. However, before failing, the current success continuation is enqueued on the currently active cleanup queue. If backtracking causes control to return from the goal of a `cleanup` operation, a continuation is dequeued from the associated cleanup queue and invoked. If the queue is empty, then backtracking proceeds past the `cleanup`.

Consider the following program schema.

```
p :- cleanup(q1).

q1 :- defer, (q2 ; q3).
q1 :- q4.

q2 :- defer, q5.
q2 :- q6.

q3 :- defer, q7.
q3 :- q8.
```

Invoking `p` causes `cleanup` to create a cleanup queue and control proceeds into `q1`. The `defer` in the first clause of `q1` causes a continuation that performs the alternation `q2;q3` to be enqueued, and `q4` to be entered. Assuming `q4` fails, the continuation which performs the alternation `q2;q3` is dequeued. When control enters `q2`, evaluation of `q5` is deferred and `q6` is entered. When `q6` fails, `q3` is called, `q7` is deferred and `q8` called. Assuming `q4`, `q6`, and `q8` fail, the final order in which subgoals are visited is `cleanup`, `q1`, `defer`, `q4`, `q2`, `defer`, `q6`, `q3`, `defer`, `q8`, `q5`, `q7`. This order combines elements of both depth-first and breadth-first search. The control tree for this procedure is traversed in an order indicated

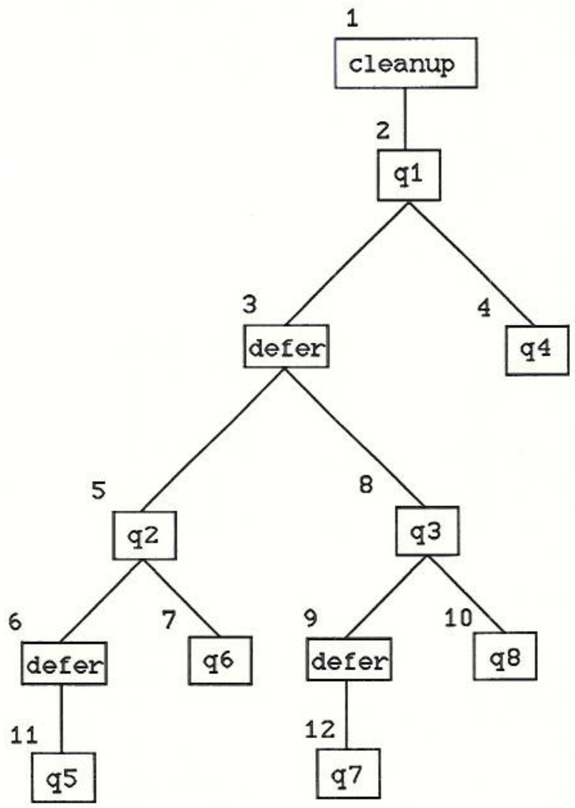


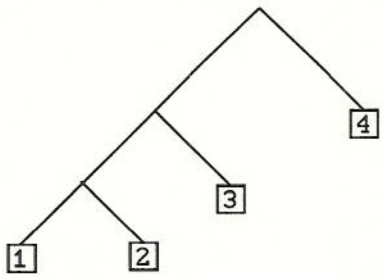
*Figure 1.* Sample control pattern for `cleanup/defer`

by the node numbers in Figure 1.

To achieve completeness with reasonable efficiency, many problems involving extensive search require a combination of depth- and breadth-first search. Such searches are often heuristically driven, and for these we allow `defer` to accept a numeric argument. This argument is used as a priority value for insertion of the deferred continuation into the queue (lower values indicate higher priority), and makes it straightforward to implement a variety of heuristic-based search strategies. If `defer` is called with no arguments, the priority defaults to 1.

## Example: Breadth-first Search

To illustrate an application of `cleanup` and `defer`, we present a procedure `bf` that performs a purely breadth-first traversal of a binary tree. We assume a binary tree is represented as either an atomic list element or a pair of the form [$L$|$R$], where $L$ and $R$ are binary trees. For example, the list structure `[[[1|2]|3]|4]` represents the binary tree depicted by



The relation `bf(T,X)` binds `X` to the leaf values of `T` in a breadth-first fashion. It is obtained by adding `cleanup` and `defer` to a standard depth-first tree traversal program.

```
bf(T, X) :- cleanup(traverse(T, X)).
traverse([L|R], X) :- !, defer, (traverse(L, X); traverse(R, X)).
traverse(X, X) :- .
```

`bf` serves only to create a cleanup queue. By using `defer` to postpone traversal of an interior node, each call to `traverse` will descend just one level. If the node is a leaf, the traversal terminates successfully by binding `X` to the leaf value. Otherwise, traversal of the node's children is deferred. These deferred traversals are processed in turn as the cleanup queue is accessed. Failure in the left child of such an interior node (the left disjunct) causes the traversal to continue in the right child (the right disjunct). Failure in the right child results in backtracking to the `cleanup` point. This dequeues the next continuation, so that the traversal continues with the next pair of sibling nodes. Each time an interior node is visited another disjunction is added to the queue. As nodes are traversed from left to right, continuations that traverse their children are added to the queue in a left to right order.

## Example: Algorithm A*

A straightforward implementation of the well-known heuristic search procedure *Algorithm A** [14] is possible, under certain restrictions, by providing `defer` with a computed priority. Algorithm A* applies to problems that can be represented in terms of a *production system.* To specify the problem we define a space of possible *states* and a set of *transition rules,* which are partial functions over the state space. The solution is a sequence of states indicating a path between given initial and goal states. Usually a minimal path is sought based on a cost assigned to each transition.

A frequently used example for such a problem is the *eight puzzle.* [14] The eight puzzle consists of a $3 \times 3$ matrix representing 8 numbered movable tiles. One cell is always empty, making it possible to move an adjacent tile into the empty cell, thereby moving the empty cell as well. These transitions can be completely described using four functions,

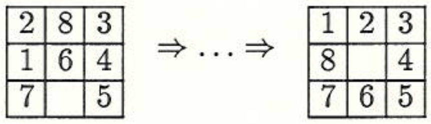


*Figure 2.* Initial and goal configurations for the 8-puzzle

each moving the empty cell in one of four directions. Figure 2 illustrates possible initial and final states for this problem.

A naive approach is exhaustive search, using backtracking, of a tree representing all possible non-cyclic paths. Assume `move(State,NewState)` is a nondeterministic procedure relating `State` with a `NewState` reachable from it by one of the transition rules. A Prolog program that performs this search under the assumption of uniform transition cost follows.

```
pathsearch(Initial,Goal,Sol) :-
     retractall(seen_state/1), solve_bt(Initial,Goal,[],Sol).

solve_bt(Goal,Goal,PSol,Sol) :- Sol = reverse([Goal|PSol]).
solve_bt(State,Goal,PSol,Sol) :- seen_state(State), !, fail.
solve_bt(State,Goal,PSol,Sol) :- asserta(seen_state(X)),
     move(State,Newstate), solve_bt(NewState,Goal,[State|PSol],Sol).
```

The relation `seen_state` records states as they are produced. The variable `Sol` holds the solution, a sequence of states along a path from the initial state to the goal. `PSol` holds the partial solution consisting of the sequence of states along the path from the initial to the current state (in reverse order).

For all but the simplest problems, `pathsearch` is unsatisfactory. It may be improved by using a heuristic to order the search. Normally an explicit graph structure is required to enable resumption of the search at any node, requiring a substantial restructuring of the `pathsearch` program. In Conlog we can retain the structure of `pathsearch`, relying on first-class continuations to maintain the context of each search path so that it may be resumed when appropriate. (The Conlog run-time system maintains a control tree with first-class continuations at the leaves, which takes the place of an explicit graph structure.)

The basic search algorithm, known as *graphsearch*, maintains a pool of states that have yet to be explored and uses an evaluation function $f$ to determine which move to make. In *Algorithm A*, $f(n) = g(n) + h(n)$ is an estimate of the minimal cost path from the initial to the goal state constrained to pass through state $n$. $g(n)$ is the current best cost from the initial state to $n$, and a heuristic function $h(n)$ estimates the minimal cost from $n$ to the goal state. If $h$ is bounded from above by the actual minimal cost then we have algorithm A*, which is guaranteed to terminate with an optimal path if a path exists.

```
graphsearch(Initial,Goal,Sol) :- retractall(seen_state/1),
     cleanup(solve_a(Initial,Goal,0,[],Sol)).

solve_a(Goal,Goal,PathCost,PSol,Sol) :- Sol = reverse([Goal|PSol]).

solve_a(State,Goal,PathCost,PSol,Sol) :- seen_state(State), !, fail.

solve_a(State,Goal,PathCost,PSol,Sol) :- asserta(seen_state(State)),
     h(State,H), F is H + PathCost, defer(F),
     move_with_cost(State,Newstate,C), NewPathCost is PathCost + C,
     solve_a(NewState,Goal,NewPathCost,[State|PSol],Sol).
```

*Figure 4. Algorithm A** using `cleanup` and `defer`

(If $h \equiv 0$ and transition costs are uniform this produces a purely breadth-first search).

Since the evaluation function uses the current best cost from the initial state to $n$, this value must be continually revised whenever $n$ is reached by a better path. Under a second restriction (the "monotone restriction") we are guaranteed that each state $n$ will be reached for the first time from a given neighboring state along a minimal cost path from that state. We will assume that this criterion holds for the heuristic function used in our solution.

A `graphsearch` procedure may be obtained in Conlog by modifying `pathsearch`. First we upgrade procedure `move` to `move_with_cost`, which includes a third parameter `C` bound to the cost of the transition from `State` to `NewState` (in the simplest case this is 1 for all transitions). The procedure `h(State,H)` computes the heuristic value associated with `State` and binds it to `H`. Pending moves are associated with continuations, and a priority cleanup queue manages these continuations according to the priorities set forth by the evaluation function. The priority queue is established by a `cleanup` at the beginning of the search. In order to capture the continuation identified with expansion of a state, a `defer` must be placed at a point after the state is determined to be a viable candidate for a move, but before the move is made. In the `pathsearch` program this occurs immediately prior to the call to `move` in the third clause of `solve_bt`. By then it is known that the state is neither the goal state (handled by the first clause), nor one that has been seen before (handled by the second clause). Since `defer` expects a priority value we must evaluate the state and pass this value to `defer` as the priority to be associated with its future expansion. Since this evaluation requires knowledge of the accumulated cost incurred in reaching the state, we keep track of this quantity with the parameter `PathCost`.

The `graphsearch` program retains the overall structure of `pathsearch`, while implementing a considerably more complex algorithm. When `move_with_cost` is called, it binds `NewState` nondeterministically to each immediate descendent of `State`. The call `solve_a(NewState,...)` that follows either terminates with a complete solution, fails because the state has already been seen, or fails after deferring the move to this state. Each call to `move_with_cost` is therefore followed either by failure or termination. Thus in the case of nontermination, backtracking will cause `NewState` to be bound successively to adjacent states. These states will in turn be discarded or added to the queue. Once all adjacent states have been processed, control returns to `cleanup`, which dequeues the continuation that moves to the state of highest priority.

## 3. Multi-tasking

It is well known that many applications involving deep search realize a significant speedup when a coroutine discipline is used to alternate steps of several search processes. [13] A simple example of this is the *samefringe* problem, in which the leaf elements of two trees are compared. [17] When the trees can be searched in parallel, with comparisons made as soon as leaves are encountered, termination is possible as soon as a difference is found. Coroutines are a traditional method for accomplishing this.

Clark [4] and Porto [15] incorporate coroutine mechanisms into Prolog that make use of variable annotations and a novel control regime, respectively. In this section we present mechanisms that implement multi-tasking without special annotations or a nonstandard control regime. Each task (or coroutine) retains control until it explicitly passes control to a scheduler and thereby becomes idle. A first-class continuation is used to retain the state of each task during its idle period. A scheduler chooses an idle task for dispatch, unlike traditional coroutine systems in which the next coroutine to be resumed is explicitly named at the time a coroutine becomes idle.

The two variations on synchronous multi-tasking in Conlog are termed `multi_step` and `multi_sweep`. Both use a simple first-in-first-out scheduling discipline, but variations with more complex scheduling are possible. The description of these operators requires some terminology. We refer to the computation from the invocation of a goal to its success or failure, or its non-termination, as a *phase* of computation. The terms *success phase*, *failure phase,* and *nonterminating phase* will also be used. Since a goal may be reinvoked following a success phase, it may be viewed as yielding a sequence of zero or more success phases, each returning a set of bindings to its caller, followed by a failure phase or nontermination phase.

Both `multi_step` and `multi_sweep` take a list of subgoals. A round-robin multi-tasking discipline is used, with each subgoal evaluated by a task that idles between phases. A task terminates after its failure phase, and the multi operators fail when all of their tasks have terminated.

With multi_step, control returns to the caller after each success phase. The next phase of the next task begins upon backtracking. Any bindings produced by a multi_step task are visible to the caller of multi_step, and are seen by that task when it is next invoked by backtracking. However, these bindings are not seen by the other tasks.

With multi_sweep, control returns after a complete sweep through its list of tasks, and upon backtracking another complete sweep is made. As with multi_step, any bindings made by a task in one phase are seen by the same task when it is resumed on a subsequent sweep initiated via backtracking. However, unlike multi_step, none of the bindings made by its tasks are visible to the caller of multi_sweep. (Communication is still possible via the database.)

**Example: Scatter Search**

To illustrate the use of multi_step, we present a binary tree search procedure that uses a novel search pattern, which we refer to as a *scatter search.* If $T$ is a (possibly infinite) binary tree with a value associated with each leaf, then $SS(T) \Rightarrow \langle v_1, \ldots \rangle$ indicates that the scatter search of $T$ visits leaves in the order indicated by the (possibly infinite) sequence of leaf values. With this notation, scatter search may be characterized by the following rules:

(a) If $T$ is a leaf with value $v$, then $SS(T) \Rightarrow \langle v \rangle$,

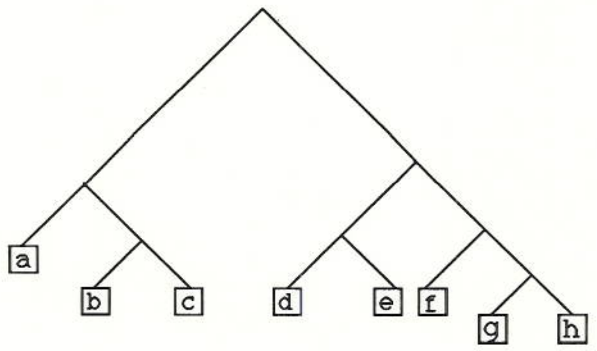
(b) If $T$ is an interior node with sons $L$ and $R$, $SS(L) \Rightarrow \langle l_1, l_2, \ldots \rangle$, and $SS(R) \Rightarrow \langle r_1, r_2, \ldots \rangle$, then $SS(T) \Rightarrow \langle l_1, r_1, l_2, r_2, \ldots \rangle$, provided the sequences $\langle l_1, l_2, \ldots \rangle$ and $\langle r_1, r_2, \ldots \rangle$ are the same length. If one of $SS(L)$ or $SS(R)$ terminates before the other, then $SS(T)$ continues with the remaining sequence. For example, if $\langle l_1, l_2, \ldots, l_n \rangle$ is shorter than $\langle r_1, r_2, \ldots \rangle$, then $SS(T) \Rightarrow \langle l_1, r_1, l_2, r_2, \ldots, l_n, r_n, r_{n+1}, \ldots \rangle$.

In a scatter search each interior node performs an alternating merge of the leaf sequences of its children. That is, it polls its children in sequence from left to right (as in breadth-first search). Interior nodes deliver one leaf node value each time they are polled, unless there are no remaining leaf nodes, in which case the node reports failure and is ignored. The result is a type of breadth-first search in which computation time is distributed over the tree in a fair manner that assesses a unit of time for each leaf and no time for visiting interior nodes.

A simple implementation of scatter search may be obtained using multi_step. Using the same tree representation as before, we have:

```
ss([L|R], X) :- !, multi_step([ss(L, X), ss(R, X)]).
ss(X, X) :- .
```

From the definition of multi_step we see that the right-hand side of the first clause will produce bindings for X that alternate between those produced by ss(L,X) and ss(R,X), as required. For example, consider the tree `[[a|[b|c]]|[[d|e]|[f[g|h]]]`, which has the form



The call `ss([[a|[b|c]]|[[d|e]|[f|[g|h]]]])` is equivalent to

```
1 multi_step([multi_step([ss(a),
2                          multi_step([ss(b),
3                                      ss(c)])]),
4             multi_step([multi_step([ss(d),
5                                     ss(e)])]),
6                         multi_step([ss(f),
7                                     multi_step([ss(g),
8                                                 ss(h)])])])])
```

The order in which leaves are visited is indicated by the following list of line-number:leaf-letter pairs: 1:a, 4:d, 2:b, 6:f, 3:c, 5:e, 7:g, 8:h.

### Example: Parallel Tree Walk

A more elaborate example illustrating `multi-step` is a solution to the branch and bound problem described by Lindstrom [13]. We are given two trees $T_1$ and $T_2$ with positively weighted nodes. The problem is to determine the tree that contains the path with the least total path weight without necessarily computing the path weight itself. The simplest approach is to compute the best path weight in each tree and compare results. The individual path weights can be computed using a branch and bound approach to cut off search on a path that exceeds the optimal computed so far. A better approach is to alternate between trees each time a path that beats the current optimal is found. This way, if one of the trees runs out without finding a better value we can terminate without having to complete the search of the other. The solution is shown in Figure 5 as the relation `optimal(T1,T2,Winner)`. Trees `T1` and `T2` are represented as three element lists, consisting of the node weight followed by the left and right descendants. `Winner` is bound to either `T1` or `T2` depending on the outcome.

Implementing this strategy requires the ability to alternate control between two backtracking contexts. The searches are managed by the procedure best_path. best_path uses the accumulating parameter SoFar to hold the weight accumulated on the way down. The parameter Weight holds the final weight value of a completed path. Search cut-off relies on the relation best/1, which asserts the current optimal value. In the first clause we have reached a tip node, so we bind Weight accordingly and return. The second clause represents a cut-off: the accumulated path weight at the current node exceeds the current optimal. We return, binding Weight to the incomplete path weight (which will be rejected). The final two clauses carry the search into the left and right subtrees, respectively. The weights returned by these searches are compared with the current optimal, and search continues until either a better value is found or the tree is exhausted.

When best_path successfully returns with a new path weight it is guaranteed to beat the current optimal. This value replaces the current asserted value as the new optimal. The relation assert_best(T,TNum) changes the database relation best accordingly whenever a completed search of a path in T returns with a new optimal value. If, on the other hand, best_path fails, assert_best asserts the relation done(TNum), where TNum is a tag (either 1 or 2) identifying the tree.

multi_step is used to alternate between search phases in the two trees. Following a search phase in one of the trees either a new best value has been asserted, or the search has ended in failure. In the first case the new best value becomes the value used by the next phase of the other tree for further comparisons. Otherwise, the program terminates, declaring the other tree as containing the global optimal path weight. Thus following each phase of multi_step it is necessary to check for termination in the tree most recently searched. The relation check_for_completion(T1,T2,Winner) does this by checking the database for the done relation and binding Winner to the tree that has not terminated. If neither tree has terminated, check_for_completion fails and control backtracks into multi_step for the next phase of the next tree.

The top level procedure optimal(T1,T2,Winner) initializes the best relation to "infinity" and sequences the multi_step-controlled search of trees T1 and T2 with a call to check_for_completion.

## Example: Samefringe

To illustrate multi_sweep we provide a solution to the samefringe problem. [17] Given two trees, possibly with different internal structures, we are interested in determining whether the sequence of nodes visited by depth-first traversal of the trees is the same. An efficient solution searches the trees in parallel so that termination is possible as soon as an unequal pair is found. This requires search procedures that are capable of being suspended, as each element is found, while retaining their control state.

```
optimal(T1,T2,Winner) :-
    retractall(done/1),
    retractall(best/1),
    asserta(best(100000)),
    multi_step([assert_best(T1,1), assert_best(T2,2)]),
    check_for_completion(T1,T2,Winner).

assert_best(T,TNum) :-
    best_path(T,0,Weight), retract(best(_)), asserta(best(Weight)).

assert_best(T,TNum) :- asserta(done(TNum)).

check_for_completion(T1,T2,Winner) :- done(1), Winner = T2.

check_for_completion(T1,T2,Winner) :- done(2), Winner = T1.

best_path([R,[],[]],SoFar,Weight) :- !, Weight =  R + SoFar.

best_path([R,LSon,Rson],SoFar,Weight) :- best(Best), Best < SoFar + R,
    !, Weight =  R + SoFar.

best_path([R,LSon,Rson],SoFar,Weight) :- NewSoFar = R + SoFar,
    best_path(LSon,NewSoFar,WLeft), best(Best), WLeft < Best,
    Weight = WLeft.

best_path([R,LSon,Rson],SoFar,Weight) :- NewSoFar = R + SoFar,
    best_path(RSon,NewSoFar,WRight), best(Best), WRight < Best,
    Weight = WRight.
```

*Figure 5.* Parallel Tree Walk

We start with a standard left-right tree traversal program, called `is_fringe_element`. Returning to the tree representation used in scatter search, we have:

```
is_fringe_element([L|R], X) :- is_fringe_element(L, X).
is_fringe_element([L|R], X) :- !, is_fringe_element(R, X).
is_fringe_element(X, X).
```

If we proceed as in the previous example with the call

```
multi_step([is_fringe_element(T1, X), is_fringe_element(T2, X)]).
```

`X` is bound alternately to successive fringe elements of `T1` and `T2`. Unfortunately, this does not allow us to compare fringe elements. Instead we use `multi_sweep` to cycle through a phase of `is_fringe_element` for each tree before returning to the caller for comparison. Since `multi_sweep` does not include the caller in the binding scope of its constituents, we must assert a database entry following each search phase. We use the relation `fringe_element(X, N)`, where `X` is the element, or `exhausted` if the tree is exhausted, and `N` is a tag (either 1 or 2) identifying the tree. The procedure `fringe` performs this task for a given tree and tag.

```
fringe(T, N) :- is_fringe_element(T, X), asserta(fringe_element(X, N)).
fringe(T, N) :- asserta(fringe_element(exhausted, N)).
```

Now each phase of the call

```
multi_sweep([fringe(T1, 1), fringe(T2, 2)])
```

asserts two entries into the database corresponding to the next two fringe values, or the sentinel value `exhausted` once either of the trees runs out. Upon return from each `multi_sweep` phase we retrieve the asserted elements, retract them from the database, and succeed only if they are equal. This check is performed by the procedure `same_fringe_elements`:

```
same_fringe_elements :-
        fringe_element(X,1),
        fringe_element(Y,2),
        retractall(fringe_element/2),
        X = Y.
```

`T1` and `T2` will have identical fringes only if each success phase of the `multi_sweep` combination asserts values that pass the test imposed by `same_fringe_elements`. The relation `for_all(P1,P2)` introduces precisely this control structure: the `for_all` succeeds only if every success of `P1` is followed by a success of `P2`. If `P2` ever fails, then `for_all` also fails.

```
samefringe(T1, T2) :-
        for_all(multi_sweep([fringe(T1, 1), fringe(T2, 2)]),
                same_fringe_elements).

for_all(P1, P2) :- not(call(P1), not(call(P2))).
```

## 4. Operators That Generalize Cut

From the perspective of continuations, a cut replaces the current failure continuation with the failure continuation associated with the point of entry into the current procedure. First-class continuations allow cut to be generalized, giving the programmer the ability to choose the failure continuation that is to be installed. Since the scope of variables is severly limited in Prolog, it is convenient to use an indirect means of referencing these continuations.

In this section we present three operators, `mark_region`, `mark` and `cut_to_mark`, that generalize cut using a *mark queue* to record failure continuations. Invoking `mark_region(`$Tag$`, `$G$`)` calls goal $G$ with a fresh queue identified by $Tag$. The scope of the mark queue is similar to that of a `cleanup` queue; it is active while $G$ is being evaluated, except when shadowed by another `mark_region` call with the same tag. The operations `mark(`$Tag$`)` and `cut_to_mark(`$Tag$`)` both reference the active queue indicated by $Tag$. (It is an error if such a queue does not exist.) The goal `mark(`$Tag$`)`, which always succeeds, reifies the current failure continuation and enqueues it on the currently active queue indicated by $Tag$. As with cut, `cut_to_mark(`$Tag$`)` also succeeds, but the current failure continuation is replaced by one that dequeues a failure continuation from the indicated queue (if the queue is nonempty) and invokes it. We refer to this as *invoking the top mark*. If the queue is empty, then the failure continuation of the `mark_region` call that created the queue is invoked. Note that the queue is not consulted until control backtracks to the `cut_to_mark` point. Cut can be implemented with these operators by preceding each call to a relation containing a cut with a call to `mark_region`, and replacing cut by `cut_to_mark`.

We illustrate the use of the mark operators with the following program schema:

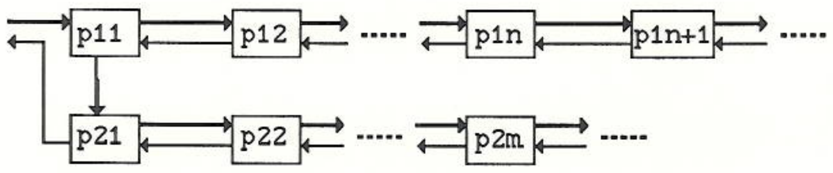
```
p1 :- mark_region(tag, p2).
p2 :- q1, mark(tag), q2, q3, cut_to_mark(tag), q4, fail.
```

The `mark` operation following `q1` enqueues the current failure continuation, which will backtrack into `q1`. Execution proceeds with `q2`, `q3`, `cut_to_mark(tag)`, and `q4`. `fail` initiates backtracking by invoking the current failure continuation. If control backtracks to the `cut_to_mark` point, the continuation enqueued by `mark` is invoked, causing control to backtrack into `q1`. Backtracking through `q2` and `q3` is avoided.

*Control diagrams* may be used to graphically illustrate the behavior of control operators. [3] Each box represents a goal. Boxes are connected by thick and thin arrows, representing success and failure continuations, respectively. For example, the program schema

```
p :- p11, p12, ... p1n, p1n+1 ...
p :- p21, p22, ....p2m ...
```

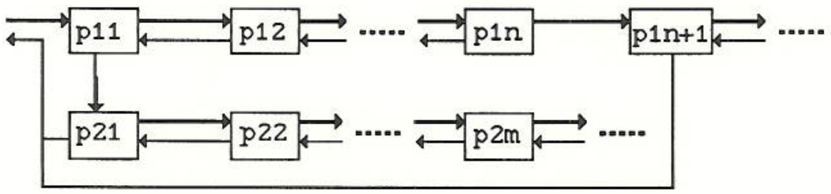
may be represented by the following control diagram.



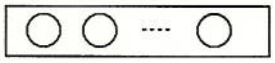
Notice that the failure continuation from `p11` turns into a success continuation on entering `p21`. The effect of cut in the modified schema

```
p :- p11, p12, ... p1n, !, p1n+1 ...
p :- p21, p22, ....p2m ...
```

is shown by the following control diagram:



To illustrate the behavior of the mark operators, we augment the control diagrams with representations of mark queues. Mark queues are depicted as follows (the front of the queue is on the left):



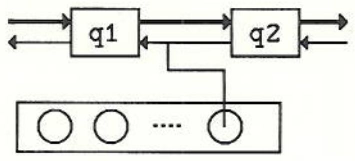
We assume that operations are being performed in the context of a single mark queue. Figure 6 illustrates several sequences containing `mark` and `cut_to_mark`, including the previous example that defined procedures `p1` and `p2`. The effect of `mark` is to enqueue a continuation on the mark queue, while `cut_to_mark` redirects a failure continuation arrow so that it points directly to the mark queue. It points to the queue rather than to a specific queue element, because the continuation to be invoked on failure is not selected until the failure occurs.

It is natural to ask whether there is need for a mechanism that enqueues success continuations rather than failure continuations. One would probably not want to queue the current success continuation because it is about to be executed. Instead, suppose one wants to continue at this time by executing goal `q1`, but enqueue a continuation that will allow the computation to continue by executing goal `q2` in the current control context. This is effectively enqueueing a success continuation for `q2`.

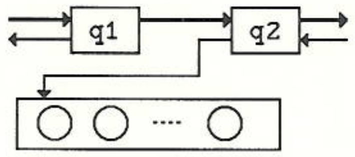
This may be accomplished by invoking a relation `q` defined as follows:

```
q :- mark(tag), !, q1.
q :- q2.
```

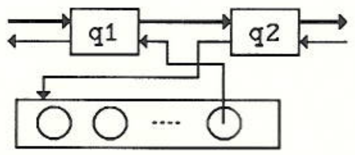
The failure continuation enqueued by `mark(tag)` will backtrack into `q2` as desired. This alternate path is deleted from the current control context by the following cut, and then control proceeds to `q1`. For the enqueued continuation to be invoked, it must first be installed as a failure continuation using `cut_to_mark`. To model success continuations,



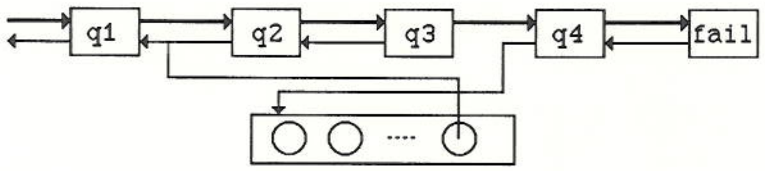
a) q1, mark(tag), q2.



b) q1, cut_to_mark(tag), q2.



c) q1, mark(tag), cut_to_mark(tag), q2.



d) q1, mark(tag), q2, q3, cut_to_mark(tag), q4, fail.

*Figure 6.*

which take control as soon as they are invoked, it is only necessary to fail immediately after installing the enqueued failure continuation. The idiom for invoking a simulated success continuation is thus

```
... , cut_to_mark(tag), fail.
```

Figure 7 illustrates the behavior of the conjunction `p,q,r`, with `p` and `r` arbitrary proce-

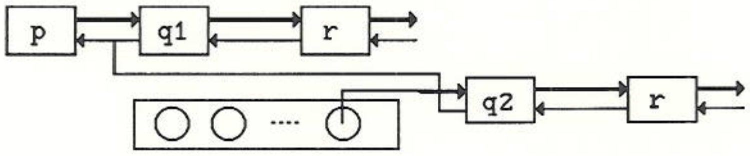


*Figure 7.*

dures and q defined as above. When either q1 or q2 succeeds, control passes to r, while if either fails, control returns to p, which is the desired behavior.

The above example demonstrates that with the ability to manipulate failure continuations one also effectively has the ability to manipulate success continuations. Thus a similar queue mechanism for success continuations is unnecessary. Since failure continuations express possible future computation paths that will occur only upon failure, enqueuing them (with `mark`) or installing them (with `cut_to_mark`) does not immediately affect the current path (though they may be forced to take effect immediately, as with `fail` above). Success continuations do not have this delayed behavior, and thus it does not seem possible to simulate failure continuations given success continuations.

## 5. Implementation

The operators described in the previous section have all been implemented directly as part of our Scheme embedding. However, a variety of more direct and efficient implementation methods are possible. The main requirement is that provision be made for heap allocating control and binding information when necessary, and for the restoration of variable bindings when continuations are invoked [7, 16]. In particular, it should be possible to extend the Warren Abstract Machine [20] to support these operations.

In this section we demonstrate that it suffices for an implementation to support the mark operators, for these may be used to implement `cleanup`, `defer`, `multi_step`, and `multi_sweep`. The mark operators may in turn be implemented directly, or in terms of low-level operators (see the appendix).

The set of low level operators consists of (1) an operator for binding the current failure continuation to a logic variable as a reified first-class object; (2) an operator for invoking reified failure continuations; and (3) an operator for dynamic control of database bindings. The latter operation allows queues of failure continuations to be maintained in the database with dynamic scope. Variations on the operations described in this paper, such as the use of priority rather than fifo queues, may be obtained using these operations. The ability to bind continuations to logic variables has a wide ranging effect on the language semantics, since continuations are functions and it is unreasonable to endow them with an external representation. (If the mark operations of this paper are implemented as primitives, rather than using the low-level operations, this complication does not arise.) As mentioned earlier, the low level operations are also more difficult to use. These factors place further discussion of them beyond the scope of this paper.

## 5.1 Implementing the `cleanup` Operators

In the following discussion of the implementation of the `cleanup` operators we limit ourselves to the version that employs a first-in-first-out queue. An analogous implemention with priority queues can be made using a corresponding priority queue version of the `mark` operators. In the implementation of `cleanup` and `defer`, each cleanup queue corresponds to a new mark queue. Since cleanup queues are not tagged, we use a unique tag, `cleanup_queue`, to identify them.

The call `cleanup(`$G$`)` creates a new active cleanup queue. The queue is consulted only when backtracking returns control to the `cleanup` point. The `cleanup` procedure simply creates a new mark queue and executes a sequence that first cuts to this queue and then calls G:

```
cleanup(G) :- mark_region(cleanup_queue,
                          (cut_to_mark(cleanup_queue), call(G))).
```

The invocation of `cleanup(p)` is illustrated in Figure 8(a).

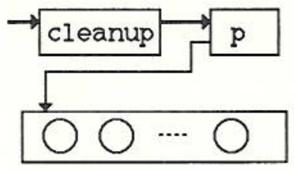
`defer` must fail immediately, while enqueueing a success continuation that returns to its caller. At the end of the last section we developed a programming idiom that can be used to queue an alternate success continuation. We abstract this idiom with the procedure `now/if_cut_to`. The call `now/if_cut_to(tag, p1, p2)` results in an immediate call to `p1`, while enqueueing a call to `p2` on the mark queue identified by `tag`. The latter call is invoked through a future `cut_to_mark(tag)`. Both calls are followed, upon success or failure, by a return to the caller of `now/if_cut_to`.

```
now/if_cut_to(Tag, P1, P2) :- mark(Tag), !, call(P1).
now/if_cut_to(Tag, P1, P2) :- call(P2).
```

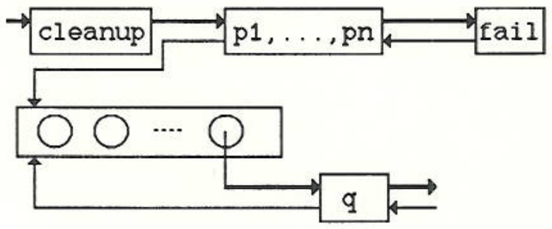
To implement `defer` we let `P1` be `fail`. The enqueued continuation will be invoked either by `cleanup`, in the case of the first `defer`, or by other deferral points that control has backed into. Thus upon return to the defer point a `cut_to_mark` is used to ensure that if control subsequently backs into the `defer`, it will pass to the next deferral point. `P2`, then, is `cut_to_mark(cleanup_queue)`.

```
defer :- now/if_cut_to(cleanup_queue, fail, cut_to_mark(cleanup_queue)).
```

Figure 8(b) illustrates the sequence `cleanup(p)`, where `p :- p1, ..., pn, defer, q`.



a) cleanup(p)



b) cleanup(p), where p :- p1, ..., pn, defer, q

*Figure 8.* Implementation of cleanup and defer

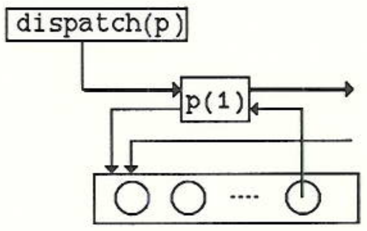
## 5.2 Implementing the multi Operators

The implementation of multi_step is more complicated. Once again, a mark queue with a unique tag, multi_queue, is created as the first step in creating the multiprocessing kernel. Into this queue are placed a sequence of success continuations containing code that runs the phases of each task. The latter will be implemented as the procedure dispatch. Queue initialization is performed by the procedure multi_scheduler, which iterates down the list of task goals [p1, ..., pn], using now/if_cut_to to enqueue calls of the form dispatch(P) for each task. When this iteration terminates, processing of the tasks begins by invoking the top mark of the queue. This is accomplished with the following rules:

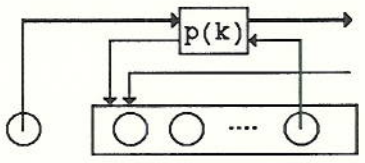
```
multi_step(Plist) :- mark_region(multi_queue, multi_scheduler(Plist)).

multi_scheduler([]) :- cut_to_mark(multi_queue), fail.
multi_scheduler([P\|Plist]) :-
    now/if_cut_to(multi_queue, multi_scheduler(Plist), dispatch(P)).
```

The purpose of dispatch is to process one phase of its argument each time it is resumed, save a continuation for the next phase, and use cut_to_mark to cause the next phase of the next task to be invoked if backtracking returns to this point. Once the final phase of the task is reached, a cut_to_mark causes the next phase of the next task to be invoked immediately. These requirements are achieved by the following steps:

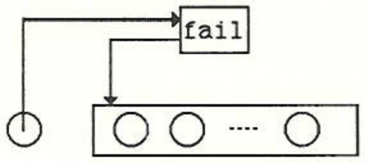
(1) perform a cut_to_mark to dispatch the next task when P fails ,



a) first phase



b) $k$th phase



c) failure phase

*Figure 9.* `dispatch` control diagrams for `multi_step` implementation

(2) invoke `P`,

(3) mark the failure continuation that resumes `P`, and

(4) perform a `cut_to_mark` to dispatch the next task when backtracking returns to this point.

This may be coded as follows:

```
dispatch(P) :- cut_to_mark(multi_queue), call(P),
               mark(multi_queue), cut_to_mark(multi_queue).
```

Figure 9 illustrates `dispatch(P)` at its first phase, at an intermediate phase, and at its failure phase.

If `P` succeeds in its first phase, processing continues into the caller of `multi_step`, but with a failure continuation that will invoke the top mark rather than backtrack into

P. The failure continuation, which is about to backtrack into P, is put at the end of the queue (Figure 9(a)). When this item reaches the front of the queue and is invoked, it processes the next phase of P and continues in the same way, marking the next failure continuation and proceeding into the sequel with a failure continuation that will invoke the top mark (Figure 9(b)). When P finally fails, the initial `cut_to_mark` invokes the top mark without enqueueing any further reference to P, effectively deleting P from contention (Figure 9(c)). When all tasks have completed, invoking the top mark on an empty queue causes backtracking to proceed from the point of definition of the queue, which was upon entering `multi_step`.

`multi_sweep` is implemented analogously to `multi_step`, with two major differences. First, `dispatch` becomes `dispatch_sweep`, which differs by performing a `fail` at the end of its sequence, causing the computation to proceed immediately into the next task. Secondly, `multi_scheduler` becomes `multi_sweep_scheduler`, which creates the analogous queue of `dispatch_sweep` calls, but enqueues a sentinel process `dispatch_sentinel` following the final `dispatch_sweep` process. This process runs only after a complete cycle of tasks has executed. When it does, it first checks the queue to see if it is empty. If it is, indicating that all of the processes have failed, then `dispatch_sentinel` fails, terminating the call to `multi_sweep`. Otherwise, it returns control to the caller. When control returns to `dispatch_sentinel` upon backtracking, it enqueues a failure continuation for another call to `dispatch_sentinel` following the next sweep through the remaining processes, and then invokes the top mark.

Figure 10 contains the implementation of `multi_sweep`. We are assuming the existence of the relation `queue_empty(tag)`, which succeeds if the queue with tag `tag` is empty.

## 6. Conclusion

Languages that are based on a depth-first control strategy, such as Prolog, are valuable in solving problems for which depth-first search is suitable. However, for problems requiring other forms of control a built-in depth-first control strategy becomes an obstacle to program design. This often forces the choice of a traditional imperative or functional language for such problems. Though the control strategy of these languages is less powerful, it is more malleable.

To retain the depth-first control behavior of Prolog, while making other forms of control behavior readily accessible, we have developed Conlog—an extension of Prolog with additional control operators. These operators use first-class continuations to record the current control state at appropriate times so that control may easily return to previous states. In this paper we have presented the high-level extra-logical operators of Conlog, which may be used to obtain non-blind backtracking and multi-tasking control behavior. We have also shown how these may be implemented using a set of intermediate-level

```
multi_sweep(Plist) :-
    mark_region(multi_sweep_queue, multi_sweep_scheduler(Plist))).

multi_sweep_scheduler([]) :-
    !, now/if_cut_to(multi_sweep_queue,
                     (cut_to_mark, fail), dispatch_sentinel).
multi_sweep_scheduler([P\|Plist]) :-
    now/if_cut_to(multi_sweep_queue,
                  multi_sweep_scheduler(Plist), dispatch_sweep(P)).

dispatch_sweep(P) :-
    cut_to_mark(multi_sweep_queue), call(P),
    mark(multi_sweep_queue), cut_to_mark(multi_sweep_queue), fail.

dispatch_sentinel :- queue_empty(multi_sweep_queue), fail.
dispatch_sentinel :-
    now/if_cut_to(multi_sweep_queue, cut_to_mark, dispatch_sentinel).
```

*Figure 10.* Implementation of `multi_sweep`

operators based directly on first-class continuations.

These operators cause minimal disruption of accepted Prolog programming techniques. Insertion of cleanup (non-blind backtracking) operators into ordinary depth-first search programs converts them to breadth-first search. The user need not be concerned with details of control management other than assigning priorities to alternatives if they are required. A valuable characteristic of Conlog's multi-tasking operators is that their tasks are represented as ordinary Prolog-style procedures. Furthermore, synchronization is based on phase completion—a notion that is natural to logic programming. Conlog introduces a minimum of new mechanism of which the programmer must be aware, but allows concise expression of control behavior that is awkward to express in Prolog.

In conjunction with standard Prolog, these operators are powerful tools for specifying complex search strategies. Subproblems for which depth-first search is appropriate may take advantage of the Prolog's default control mechanism, while other forms of control may be obtained using operators employing first-class continuations.

## References

[1] Arbib, B., and Berry, D. M., "Operational and denotational semantics of Prolog," *Journal of Logic Programming,* Vol. 4 (1987), 309–329.

[2] de Bruin, A., and de Vink, E.P., "Continuation Semantics for PROLOG with Cut," Technical Report IR-160, Vrije Universiteit Amsterdam, August 1988.

[3] Clocksin, W. F., and Mellish, C. S., *Programming in Prolog,* Springer-Verlag (1981).

[4] Clark, K. L., McCabe, F. G., and Gregory, S., "I-C PROLOG language features," in K. L. Clark and S.-A. Tärnlund, eds., *Logic Programming,* Academic Press (1982), 253-266.

[5] Rees, J., and Clinger, W., "The revised$^3$ report on the algorithmic language Scheme," *SIGPLAN Notices,* Vol. 21, No. 12 (1986), 37–79.

[6] Debray, S. K., and Prateek, M., "Denotational and operational semantics for Prolog," *J. Logic Programming,* Vol. 5 (1988), 61–91.

[7] Haynes, C., "Logic continuations," *The Journal of Logic Programming,* Vol. 4 (1987), 157–176.

[8] Haynes, C., and Friedman, D. P., "Abstracting timed preemption with engines," *Computer Languages,* Vol. 12, No. 2 (1987), 109–121.

[9] Haynes, C., Friedman, D. P., and Wand, M., "Obtaining coroutines with continuations," *Computer Languages,* Vol. 11, No. 3/4 (1986), 143–153.

[10] Haynes, C. T., and Salter, R. M., "A Prolog semantics with first-class continuations and dynamic database," Indiana University Technical Report No. 292, 1989.

[11] Haynes, C. T., Likes, K. T., and Salter, R. M., "The extended control operations of Conlog and their implementation", in preparation.

[12] Kluźniak, F., and S. Szpakowics, S., "Prolog—a panacea?,"in J. A. Campbell, ed., *Implementations of Prolog,* Ellis Horwood, Ltd. (1984), 71–84.

[13] Lindstrom, G., "Backtracking in a generalized control setting," *ACM Trans. on Prog. Lang. and Systems,* Vol. 1., No. 1 (1979), 8–26.

[14] Nilsson, N., *Principles of Artificial Intelligence,* Tioga Publishing Co. (1980).

[15] Porto, A., "Epilog: a language for extended programming in logic," in J. A. Campbell, ed., *Implementations of Prolog,* Ellis Horwood, Ltd. (1984), 268–278.

[16] Salter, R., and Ashley, J., "A revised state space model for a logic programming embedding in Scheme," BIGRE Bulletin, to appear.

[17] Hewitt, C., *et al.*, "Behavioral semantics of nonrecursive control structures," *Programming Symposium Proceedings*, LNCS 19 (1975), 385–407.

[18] Sussman, G., and McDermott, D. "From PLANNER to CONNIVER—a genetic approach," *Joint Computer Conference Proceedings 41, part II*, AFIPS Press 1171-1179.

[19] Wand, M., "Continuation-based multiprocessing," *Conference Record of the 1980 Lisp Conference,* 1980, 19–28.

[20] Warren, D. H. D., *Applied Logic—Its Use and Implementation as a Programming Tool,* Ph.D. thesis, University of Edinburgh, Scotland, 1977.

## Appendix

An implementation of the `mark` operators using the lower level operators `cfc`, `cut_to`, and `protect` defined in [10] follows:

```
mark_region(Tag, G) :-
    G1 =.. [create_mark_queue, Tag, G], name(Tag, Tagstr),
    append(Tagstr, "/1", Tagstr1), append(Tagstr, "/2" Tagstr2),
    name(Tag1, Tagstr1), name(Tag2, Tagstr2),
    protect(Tag1, protect(Tag2, G1)).

create_mark_queue(Tag, G) :-
    cfc(FC), F1 =.. [Tag, _], F2 =.. [Tag, _, _],
    retractall(F1), retractall(F2),
    F3 =.. [Tag, FC, final], asserta(F3), call(G).

mark(Tag) :- cfc(FC), F =.. [Tag, FC], assertz(F).

cut_to_mark(Tag) :- .
cut_to_mark(Tag) :- F =.. [Tag, FC], call(F),
                    retract(F), cut_to(F), fail.
cut_to_mark(Tag) :- F =.. [Tag, FC, final], call(F), cut_to(F), fail.
```