With the 0/1 knapsack problem we have a number of items that have to be fit into a knapsack. Each item has a value and a weight. There is a limit to the total weight that the knapsack can hold. The object is to fill the knapsack with as great a value as possible without going over the weight limit. Parts of objects can not be put into the knapsack, only the whole object or nothing, thus the 0/1 reference. A solution to the problem would be a list of 0's and 1's, where zero indicates that the item will not be placed in the knapsack and a 1 indicates that the item will be placed in the knapsack. The items are preordered in a best value to weight ratio order.

In order to create a best-first solution, we will generate a tree of all possible combinations of zero's and one's. The first two branches of the tree will be a zero for the first item and a one for the first item. The best value that can be obtained from any position would be the inclusion of all of the next best value to weight ratio items, filling the knapsack all the way up. It might be that the last item to be placed in the knapsack is to big and will overflow the knapsack in this case as a best possible estimate the percentage weight of that object that will fill the knapsack is multiplied by the value to get the maximum value that the knapsack could hold. As a minimum best value the value is calculated without that last item. Thus with each node we generate we have an upper and lower estimate on what the best value is that the node can generate. We will always choose the position to expand that has the best upper estimate. If we reach a position to expand where zero's and one's have been generated for all of the items then its upper estimate will be equal to its lower estimate, since it is no longer an estimate, all items have been exactly defined on inclusion or not. Thus the lower estimate for this position is equal to or better than the upper estimate for all other positions, and we have reached a best cost position for the 0/1 knapsack problem.

To create a best-first solution to the 0/1 knapsack problem we don't need G the heuristic estimate of the cost from the current position to the goal. Nor do we need to keep track of the variable "goal" nor the variable "path" since the path and position are identical. The cost to this position can be regenerated very easily using the position thus F is also not needed. We are left with only the variables position and hval an upperbound estimate of the current position.

```
(defsc branchbound (position &optional (hval (cost position)))
              hval              ; value to sort
              (function >)      ; sort function
  (if (goal-reached-test position)
     (progn (print position) (print hval) (returncf))
     (generate-children position hval (make-children-list position)) ))

(defun generate-children (position hval children)
  (unless (null children)
          (when (valid-first children)
                (newframe  branchbound  (getfirst children)))
          (generate-children position hval (rest children))))
```

The generic part of the algorithm appears above and is analogous to the A* algorithm. The most significant change to the algorithm is that we cannot reach a position by two different paths thus we are always creating a new frame in generate-children, there is never a possibility that an old frame needs to be modified. Thus generate-children is simplified. The rest of the 0/1 knapsack solution appears in Appendix B.1.

6 COMPARISON WITH OTHER CONTROL STRUCTURE MECHANISMS

In this section we compare some of our control tools to some existing control, some potential other control tools, and explores some of the limitations of our control tools. The first subsection gives one way to classify control tools. The next subsection looks at the subcomponents used as operational semantics to implement different control tools. The final subsection indicates issues we are not addressing in relation to our control tools, and gives potential problems in the usage of control tools.

6.1 Types of Control Structures

One way to compare different kinds of control structures is by the data structure used to implement the control structure. In this way normal function calling is a control stack. We have introduced control queues and control sorted collections. Coroutines would be control dictionaries. Dictionaries are a data structure described in smalltalk [11]. Dictionaries are unsorted and have tags for all values. With a coroutine you have a name for each coroutine. A coroutine is accessed like a dictionary, by tag-name value, there is no order to the different coroutines. We have also introduced control carousels a tool that has control in the form of a circular queue data structure [4]. Conniver's links of control [16] might be considered some form of control graph. A control array would be like a coroutine in that frames can be accessed by an index, but they would have an additional feature of an order that they would default to if the next coroutine to go to is unspecified, this might happen on return from a coroutine.

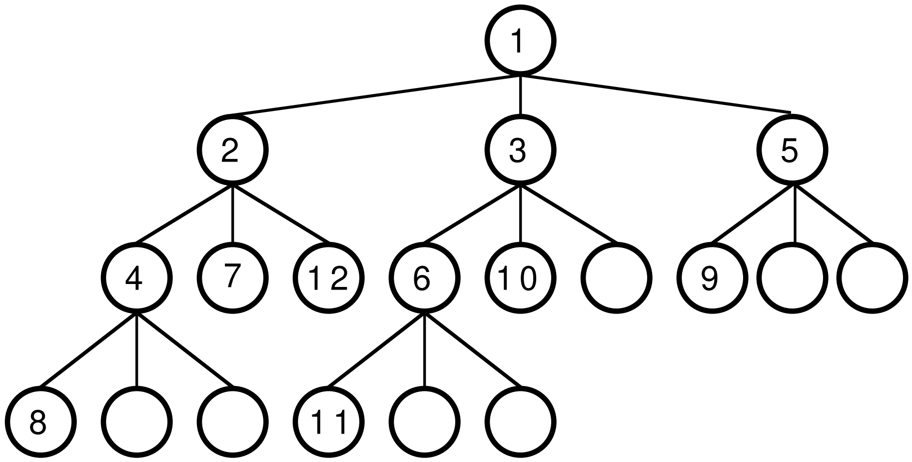
In developing a new control tool based on a specific data structure it is essential to focus on the development of a tool that mimics some form of normal human reasoning, like control queues mimic breadth-first traversal. Conniver's control structure that resembled a graph was perhaps too powerful. It did too many things that did not resemble common reasoning methods. This may be why it is no longer a popular language.

6.2 Alternative Control Subcomponents

The main difference between a control stack and control queue is the implementation of the call and return mechanism. With a control stack we have a single return from the top and a single call to the top. With a control queue we have one return to the front and two call steps to the back, the first one returns control to the front of the queue, and the second one sets up a new frame at the back of the queue and transfers control to it. If we focus on the return and call mechanisms as being the way to classify or create control mechanisms we could develop a number of new control mechanisms. We have explored this possibility, but none of these effectively match basic human conceptual paradigms or they can be easily mimicked by some of the paradigms we have already introduced and thus are inappropriate to develop.

The control mechanisms we have explored include: having only one call step for a control queue instead of two; having the two control queue steps combined with a return to the back of the queue instead of the front; having an explicit qcall instruction instead of having a defq; explicit return to front or return to back instructions; and explicit placement of the current frame in an explicit location in the control structure.

Another way we have explored varying the control flow algorithm is if we move the element on the front of the queue to the rear of the queue after it does a call. With this mechanism we get a tree traversal that is a combination of depth-first and breadth-first which we call a hybrid control queue(see Figure 36). This is the closest we have come to finding a mechanism that might be useful by classifying control mechanisms in this way.



Hybrid control queues are like control queues except that they place the calling frame on the rear of the queue. Note that the called frame will be placed behind the calling frame if the called frame makes a recursive call. The order of traversal of the nodes following those indicated in the diagram depends on the depth of the tree.

Fig. 36 Control flow for hybrid control queues.

6.3 Other issues: What is missing from CQ's?

We have avoided a number of issues in the development of our control tools. We have not discussed: scope - lexical or static; parameter passing and return values. Most of this comes from our preference in the type of language mechanisms we prefer to use. Our preference is to use global functions; with parameters passed by value and which can not be modified; and global variables. We feel that the type of tasks that are typically done with scope tools are handled better with our own control tools or a mechanism that more explicitly indicates inheritance of variables and functions such as object-oriented programming. We leave the integration of our control tools and specific scoping mechanisms like object-oriented programming to later research [5]. From preliminary sketches it appears that implementation of numerous types of scoping mechanisms pose no problems for control tools.

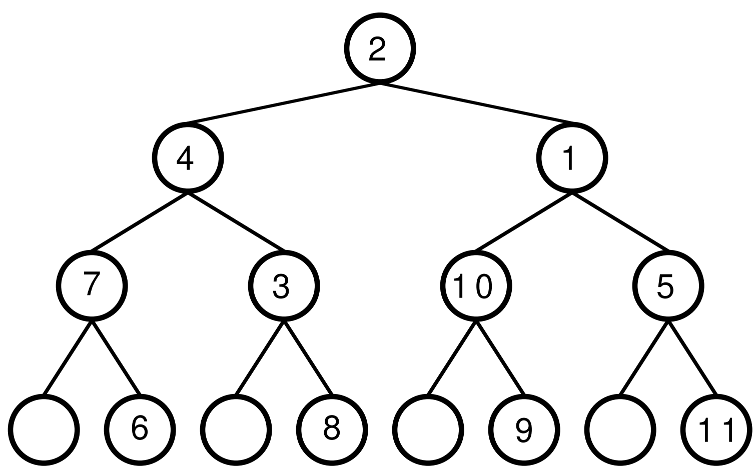
We now ask the question of how often do control queues mimic breadth-first traversal. As we saw earlier in Section 2.4, control queue almost always captured breadth-first traversal. However there is an exception. If an the action for the recursive function is done before or after the recursive function call there is no problem. If there is action both before and after then a problem arises. Below we show the function strange_behavior which traverses a binary tree. Figure 37 shows the unusual traversal order that develops. If operation upon a node is restricted to either side of the recursive control queue call, then normal behavior is maintained. Likewise if there is no double recursion no problem can occur, only in rather unusual double recursion cases with actions occuring on two sides of the double recursion calls can this problem occur.

```
(defun printnode (bitree) (print (first bitree)))
(defun arightbranch (bitree) (eq 'r (second bitree)))
(defun aleftbranch (bitree) (eq 'l (second bitree)))
(defun left (bitree) (third bitree))
(defun right (bitree) (fourth bitree))
(defq strange_behavior (bitree)
        (unless (null bitree) (progn
```

```
        (if (aRightBranch bitree) (printnode bitree))
        (strange_behavior (left bitree))
        (strange_behavior (right bitree))
        (if (aLeftBranch bitree) (printnode bitree)) )))

; In the tree given as input there is a second value for each node this value indicates what
; kind of branch is being traversed. An l indicates a left branch, while an r indicates a right
; branch.

qlisp> (strange_behavior '(a l (b l (d l (h l) (i r)) (e r (j l) (k r)))
                (c r (f l (l l) (m r)) (g r (n l) (o r)))))
C
A
E
B
G
I
D
K
M
F
O
H
J
L
N
NIL
```



Strange_behavior's traversal of a tree to a depth of three. The order of traversal of the next nodes depends on whether the tree has a depth greater than three or not. The top node is assumed not to be aRightBranch.

Fig. 37. Order of traversal for strange_behavior.

7 CONTROL QUEUE IMPLEMENTATION

We have added all of the above control tools except control sorted collections to a Lisp evaluator written in Lisp called qlisp. In this section we describe difficult implementation issues that we encountered. In sections 7.1, and sections 7.7-7.9 we outline our implementation algorithm. In sections 7.2-7.7 we describe a number of interesting control structure configurations that can be created. We describe the implementation of control queues from a general perspective. No matter how these tools are implemented the issues we discuss must be addressed. The below issues would be of concern even if control queues were added to a language different from Lisp such as C. The basic implementation of zap and grab is trivial once a basic mechanism has been developed to represent frames containing position, locals and code information, and thus not discussed. These structures are necessarily implementation dependent and part of our control tools implementation. Converttoq is a minor modification to the control queue implementation and is also not described. Control sorted collection's implementation differs from control queue's in two ways. First it is easier since there are no fragments, all of the issues discussed in Sections 7.3-7.6 are not necessary. Second a sort mechanism needs to be implemented. Sort mechanism are well known and thus straight forward and not discussed.

Compiler aspects are not considered here, likewise we leave efficiency and other implementation issues to later research.

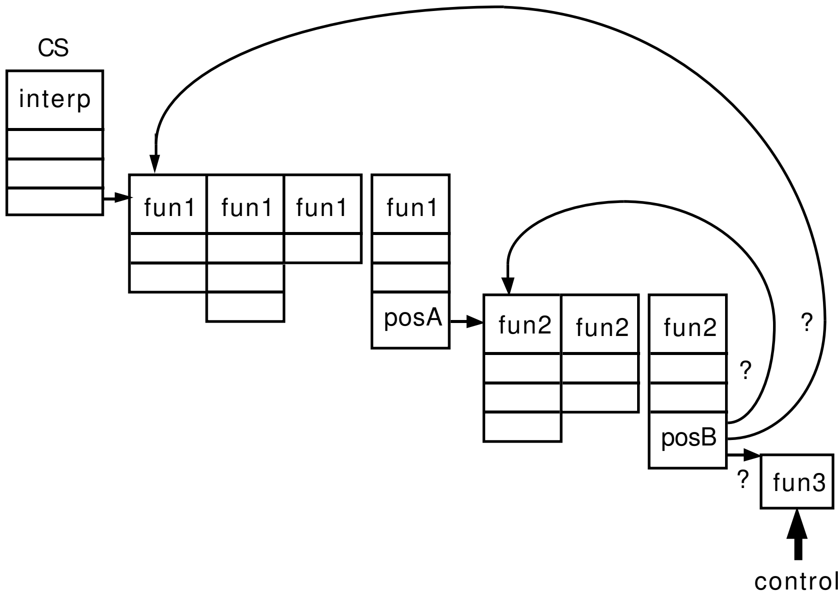
7.1 Basic Control Structure

The control structure we used is as follows. A frame contains a list of function name, local variables and current position. A control stack was a list of: frames as just described; or a queue. A queue was a list of the symbol "queue" followed by any number of control stacks. Notice that each element of a queue is a control stack, this is because within a queue function ordinary functions can be called, and thus a control stack develops within the queue frame created for that queue function. We name each control stack within a queue a queue frame. Thus a queue frame is a control stack.

Function definition, local variables, and position are necessary components of each frame, although specific representation can vary depending upon language represented and implementer's preferences. In our implementation the function could be referenced globally via the name as in [22]. Only: user defined functions including lambdas; glues; and eval calls were represented by frames, system defined functions were treated as positions in code. Local variables were defined as an association list of formal parameter - argument-value pairings. Position was indicated by embedded lists of the arguments evaluated so far and their return values. The position indication mechanism was one of the most difficult components to implement. An extra symbol was created to indicate the queue call2 mechanism. There was also a special mechanism created to allow evaluated arguments to be lists.

7.2 Levels

Different levels of control queues can develop as seen in Figure 38. Since any queue frame is a control stack it can contain a queue itself. This situation is indicated by posA calling a second queue function fun2. Within any queue frame if an ordinary function is called then a new frame is added to the control stack of the queue frame. When the existing queue (fun1) is called then control is returned to the front of the queue. However a completely



If there is a queue call at posB then it must be determined which control queue is being referred to: the current one (fun2); one a level up (fun1), or a completely new one (fun3).

Fig. 38. Levels of control queues.

new queue-function name could be called and thus a second control queue could be created (fun2) (this would be a fun2 call at posA in Figure 38.)

When a queue call is made within several levels of control queues the name of the queue call must be checked against existing control queues in the control structure (fun1 or fun2) or if it is a new control queue call name (fun3). This is the decision posB in Figure 38 and depends on whether the queue call is to fun1, fun2 or fun3.

7.3 Fragments

After a call2 a new queue frame is created that is separate from the queue until a call1 is made from within that queue frame, at that point the queue frame will be attached to the end of the queue. But in the meantime the unattached queue frame could create a second queue, as in Figure 39. The second queue (q2) could itself have an unattached queue frame which might create a third queue (q3). In this way we could create any number of disjoint fragments of the control structure (as indicated in Figure 39).

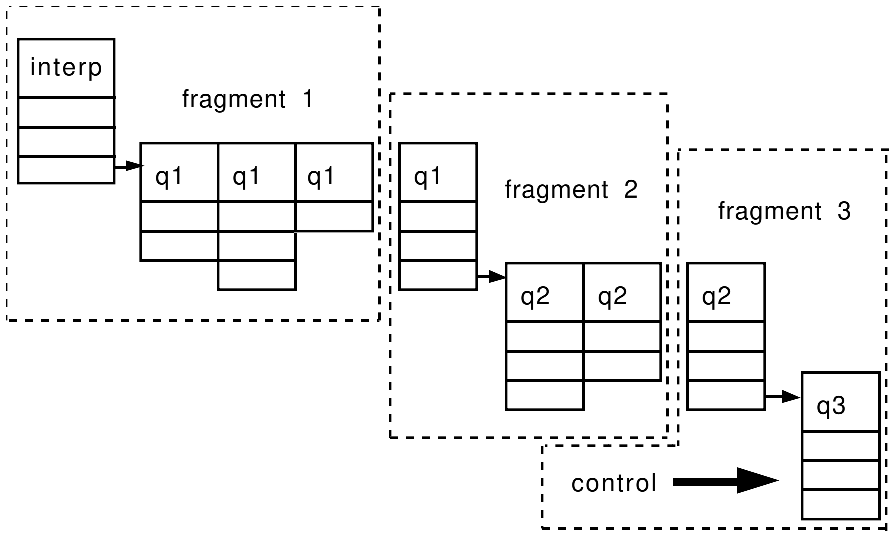


Fig. 39. Fragments.

Using fragments we also have a nice aesthetic for control. Control is always maintained in the last fragment.

7.4 Internal queues

A second queue can be created when control is at the front of a queue, as indicated in Figure 40. This queue could also create other queues. Current control will always be in the last fragment and in the first queue frame's last frame. If the last frame is also a queue, control will be within its first queue frame. This is a recursive definition of where control should reside.

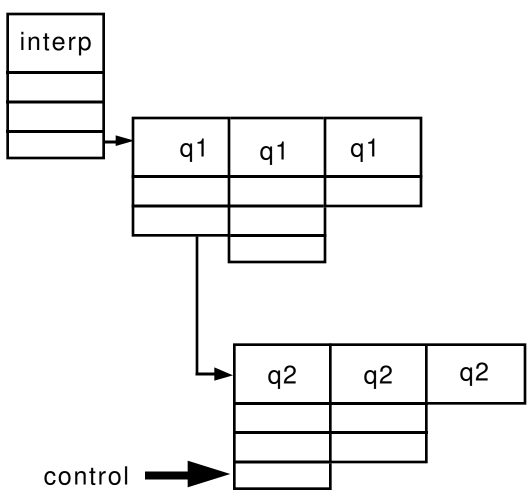


Fig. 40. Internal queues.

7.5 No Mutually Recursive Control Queues

If one queue (q3) creates a second queue (q4) which does a call1 on the original queue, a third fragment will be created as in Figure 41. If this unattached q3 queue frame does a call1 on q4 what should happen? Should control be passed to the first queue frame of q4, with fragment 3 being placed on the back of the q3 queue? If this were to happen then q4 of fragment 2 would terminate before the call2 of the q4 of fragment 4 was ever reached, thus the two q4's would never be connected within the same queue. Thus when call1 for q4 is executed in fragment 3 a completely new queue should be created. This q4 has the same name as the q4 of fragment 2 but is a different queue as indicated in Figure 41.

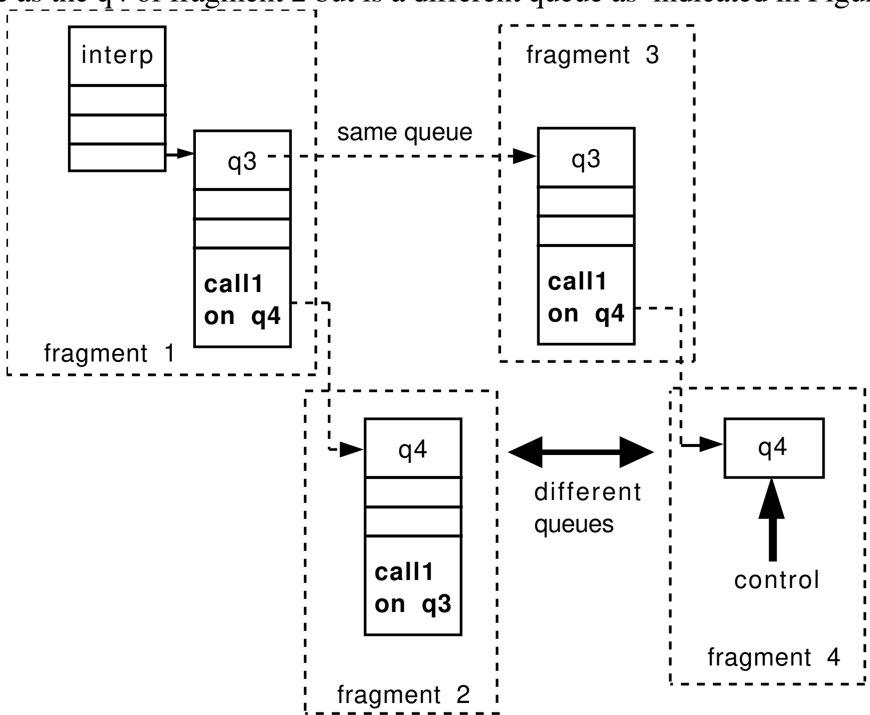
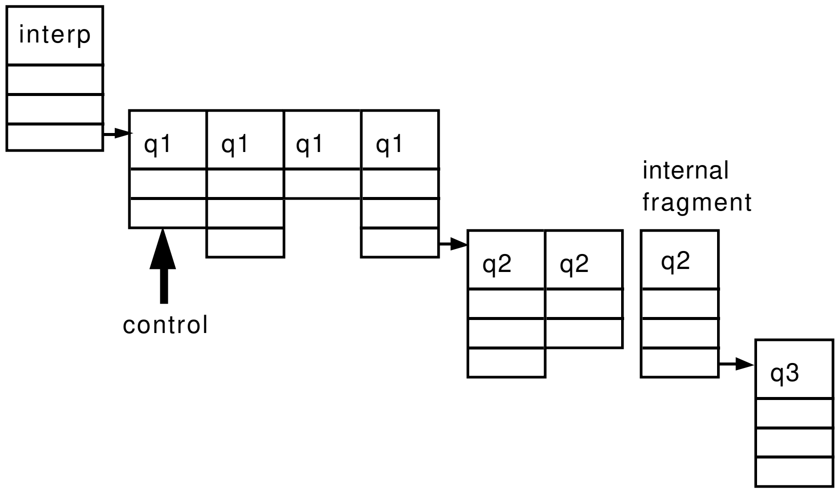


Figure 41. No mutually recursive control queues.

7.6 Internal Fragments

It is possible that a call1 is made upon a queue (q1 of Figure 39) that is the head of a fragment (fragment 2 of Figure 39) a number of fragments above the current fragment. If there is a queue (q1 of fragment 2 of Figure 39) that this fragment (fragment 2 of Figure 39) belongs to, then the fragment (fragment 2) must be grouped with all of the fragments below it, and must be added to the end of the queue (q1 of fragment 1 of Figure 39). Control will then return to the front of the queue (q1 of Figure 39). The resulting control structure would be Figure 42. In this way internal fragments can be created. Whenever the queue frame that contains these fragments is reached the fragments must be again added to the end of the list of fragments of the control structure, thus taking control. In our implementation internal fragments are recognized by the symbol "fragment" being placed as the first element in the queue frame list. The parts of an implementation that must deal with this are the call1 step 2), which must create the internal fragment; and return step 1) which must unwrap the internal fragment and make it external again.



When control returns to q2 control will be in the internal fragment. Note that more q1 frames can be created and added to the end of the q1 control queue so the interal fragment is really internal to one of the q1 frames.

Fig. 42. Internal fragment.

7.7 Call and Call2

Standard function calling and the call2 instruction turn out to be implemented identically except for two things. The type of the new frames are different and call2 advances its position while call does not advance its position, the return will do the advancing of position. Both call and call2 store the previous frame if the previous frame was a queue frame then a new fragment is created for it, otherwise that previous frame is stored upon the last fragment's current control stack whether that is a simple control stack or an embedded queue frame.

7.8 Call1

There are three different action situations for a call1. The first action situation occurs in two different control structure configurations: 1) a completely new queue could be called, and 2) the queue being called is the queue that control is currently in - with the the specification that control is within the front queue frame of that queue. Configuration numbers 1) and 2) are both indicated when the queue name being called does not match the top level queue frame of any fragment besides the first fragment. Control is always at the front of every queue except when a fragment has been started for that specific queue, in which case the queue would be at the top level of the fragment. Thus, if the queue name being called does not match the the top level of any fragments but the first, then we have either configuration 1) or 2) above. Configuration 1) occurs if the queue name does not match any queue level in the first fragment, configuration 2) occurs if it does match the a queue level contained in the first fragment. In both of these configurations the action that occurs is that no fragments are added to any queues.

The second action situation occurs if two conditions are met: a fragment can be found with the queue name being called at its top level; and there is no queue level in a fragment that occurs before that point of the of the same name. In this situation the matched fragment must be attached to the fragment immediately above it. In this action situation we are connecting a new queue to the fragment before it. Note that when searching for the fragment and queue level above it any mutually recursive queue must be excluded from being matched as mentioned above in Subsection 7.5 (this also must be done for the third action situation).

If a fragment top level queue matches the call name and a queue at any level is matched in some fragment above the fragment that is matched, then the third action situation is indicated. In this case the matched fragment through the last fragment are placed upon the matched queue. This is the point that internal fragments could be created. Internal fragments are created if more than one fragment is placed on the matched queue. Control will be returned to the front of the queue matched.

7.9 Return

If return is from the initial frame created by the interpreter, print the return value. Otherwise advance the position of the next frame by adding the return value to its' position indicator, also remove the current frame and make the next frame the current frame. Return for a queue frame is identical except for the following: do not advance the position of the next frame because call2 does this; reset internal fragments if necessary; and remove the entire queue if necessary.

8. CONCLUSIONS

We have introduced a number of axioms for the development of programming languages. The axioms include issues on reflection, control abstractions, and computational models. Using these combined axioms we have developed a set of language features for working with control structures. These control tools can be added to any programming language that has procedures or functions. These tools have a number of useful software engineering qualities and principles. Control sorted collections and control queues are general best-first and breadth-first traversal mechanisms respectively. Zap and grab provide explicit (visibility) access and manipulation of control. Zap does modular exception handling. Grab allows understandable recursive lambdas. Underlying our results is a challenge to the programming language development community to liberate itself from the current assumed operational semantics of functions and procedures.

GLOSSARY

Arguments - The values that are passed to a function call to bind with parameters. In our operational semantics arguments are stored temporarily in the list of instruction in a stack fashion immediately before the function call instruction that they are attached to.

Call - Call is a control tool instruction used to sort a control sorted collection and place control at the front of the control sorted collection. Call is also used to indicate the internal control stack operational semantics executed when a normal function is called.

Call1 - The first part of the operational semantics executed when a control queue function is called.

Call2 - The second part of the operational semantics executed when a control queue function is called.

Code - This is a control instruction that returns the code or function component of a frame argument given to it. This is also used to indicate the function definition attached to any control frame.

Control Queue - A control tool that uses a queue of control stacks instead of an ordinary control stack.

Control Sorted Collection - A control tool that uses a sorted collection of control stacks instead of an ordinary control stack.

Control Stack - The ordinary form of control structure associated with a function call. Frames associated with different embedded function calls are organized into a stack.

Converttoq - A control tool instruction that converts a control stack into control queue as a program is running.

Defq - A control tool instruction that is used to define control queues.

Defq Function Call - A term used to indicate when a function that is defined as a control queue is called.

Defsc - A control tool instruction that is used to define control sorted collections.

Fragment - The part of a control queue that is unattached from the rest of the control queue because no control queue function call of the same control queue has been executed from with in it yet.

Frame - The part of the control structure associated with a single function call. A frame maintains: local variable values; the current position within the function; and a copy or pointer to a copy of the code for the function.

Getframevar - A control tool instruction that will return values that are associated with specific variables of a frame given as an argument to this tool. The name of the variable is also given as an argument to this tool.

Glue - A control tool that will create a new frame that has a position indicator that points to the beginning of a function given to glue as an argument. It has the local variables and values of the frame that it was called from.

Grab - A control tool instruction that will return a control frame that matches either a specific position in the control stack given as an argument to this tool, or a frame that matches a specific function name and local variable values given as arguments to this tool.

Instructions - The lines of code that are executed to run the program. In the operational semantics we provide, only important instructions are listed in the order that they are to be executed. Usually they are listed at an assembly like level and may have arguments that are passed to them interspersed among them. The current instruction to be executed is given in bold.

Internal Fragment - A control queue fragment that has to be grouped with but unattached to a control queue because some higher level control queue has taken control.

Internal Queue - The type of data structure that evolves when a new control queue is created within another control queue of a different name.

Levels - Several differently named control queues are embedded within each other. Each lower level of control queue is a different level.

Locals - A control tool instruction that returns an assoc list of the local variables of a frame given to it as an argument.

Mutually Recursive - When two functions recursively call each other.

Newframe - A control tool instruction that will add a new frame to the end of a control sorted collection, given as arguments are: a control sorted collection function name; and a list of argument values that are given to the new frame as parameter values.

Pos - A control tool instruction that returns the position component of a frame given to it as an argument. This is also used to signify the part of a frame that indicates the current location in code execution.

Queue - The part of a control structure associated with a control queue. This is a queue of control stacks.

Queue Frame - A single element in a control queue, which is a control stack itself.

Queue Function - A function that has been defined as a control queue.

Queue Function Call - A call to a function that has been defined as a control queue.

Return - The operational semantics associated with the return of a function call, whether it be a control stack, control queue or control sorted collection function.

Return Value - The value that is returned from a function call. This value is passed to the position in code where the function was called. In our operational semantics this value is sometimes interspersed in the code, so that it can latter be used by an instruction as an argument.

Returncf - A control tool instruction that destroys a control sorted collection and returns control to the function that initially called the control sorted collection.

Setframe - A control tool that will set specific variables within a control structure, within any frame of the control structure not just the current frame. This tool takes a frame and variable names and values as arguments, which it will use to match the appropriate frame and set local variables.

Sort Function - This is the function that is associated with a control sorted collection that is used to sort the different CSC frames.

Sort Variables - This a structure that indicates the combination of variables to be used from each frame to do the sort of CSC frames on.

Sorted Collection Frame (CSC Frame) - A frame of a control sorted collection, which itself will consist of a control stack.

Zap - A control tool instruction used to reorganize a control stack. This instruction takes the function names of the frames that are to remain in the control stack. All frames below the frame of the last function name given as an argument also remain on the control stack.

APPENDIX A. CONTROL SORTED COLLECTION A* ALGORITHM

A.1 Shortest Path

```
; In this example we are trying to find the shortest road path form one specific city to
; another specific city.  There are numerous combinations of roads that can be taken going
; through different intermediate cities.  Estimates from the current position to the goal city
; are given by air distances, which are always a low estimate on the driving distance.

(defun getEstToGoal (city dest)
  (second (assoc city (second (assoc dest (airdistances))))))

(defun make-children-list (position)
  (second (assoc position (citylist))) )

(defun get-first (children)
  (first (first children)) )

(defun cost-first (children)
  (second (first children)) )

; Numbers in the following data have been tweaked to fully test the A*
; algorithm.

(defun citylist ()
  (quote          ((San-Francisco      ((Portland 638)
                                         (Salt-Lake-City 730)
                                         (Phoenix 806)))
                   (Portland           ((San-Francisco 638)
                                         (Salt-Lake-City 768)))
                   (Phoenix            ((San-Francisco 806)
                                         (Salt-Lake-City 646)
                                         (Denver 904)
                                         (Houston 1206)))
                   (Salt-Lake-City     ((Portland 768)
                                         (Phoenix 646)
                                         (Denver 497)
                                         (San-Francisco 730)
                                         (Chicago 1411)))
                   (Denver             ((Salt-Lake-City 497)
                                         (Phoenix 904)
                                         (Houston 1038)
                                         (Atlanta 1519)))
                   (Chicago            ((Salt-Lake-City 1411)
                                         (Atlanta 715)
                                         (Boston 976)))
                   (Houston            ((Phoenix 1206)
                                         (Denver 1038)
                                         (Miami 1306)
                                         (Atlanta 875)))
                   (Atlanta            ((Denver 1519)
                                         (Houston 875)
                                         (Miami 662)
```

```
                                  (Chicago 715)
                                  (Boston 1084)))
            (Miami               ((Houston 1306)
                                  (Atlanta 662)
                                  (Boston 1547)))
            (Boston              ((Chicago 976)
                                  (Atlanta 1084)
                                  (Miami 1547) ))))

(defun airdistances ()
  (quote              ; distances to Boston
            (Boston        ((Atlanta 953)
                            (Chicago 860)
                            (Denver 1755)
                            (Houston 1604)
                            (Miami 1290)
                            (Phoenix 2290)
                            (Portland 2570)
                            (Salt-Lake-City 2092)
                            (San-Francisco 2708)) )))

; example call

(bestfirst (quote san-francisco) (quote boston))
```

A.2. Fifteen Puzzle

```
(defun getEstToGoal (pos goal)
  ; see how many positions agree in two lists
  (cond ((null pos) 0)
           ((= (first pos) (first goal))
            (getEstToGoal (rest pos) (rest goal)))
       (t (+ 1 (getEstToGoal (rest pos) (rest goal)))) ))

(defun make-children-list (position) ; make a list with u, r, d, l generated
  (gen-u-r-d-l (find-zero position) position))

(defun find-zero (pos)
  (if (= (first pos) 0) 1 (+ 1 (find-zero (rest pos)))) )

(defun gen-u-r-d-l (zero pos)
  (remove-nils (list (up zero pos)
                     (right zero pos)
                     (down zero pos)
                     (left zero pos))) )

(defun up (zero pos)
  (when (not (eq 0 (floor (- zero 1) 4)))
       (exchange (- zero 4) zero pos)))

(defun down (zero pos)
  (when (not (eq 3 (floor (- zero 1) 4)))
       (exchange zero (+ zero 4) pos)))
```

```
(defun right (zero pos)
  (when (not (eq 0 (mod zero 4)))
        (exchange zero (+ zero 1) pos)))

(defun left (zero pos)
  (when (not (eq 1 (mod zero 4)))
        (exchange (- zero 1) zero pos)))

(remove-nils (list)
  (cond ((null list) ())
        ((null (first list))
        (remove-nils (rest list)) )
        (t (cons (first list)
                (remove-nils (rest list)) )) ))

(defun exchange (pos1 pos2 list)
  (cond ((= pos1 1)
         (puttogether (first list) (- pos2 1) (rest list)))
        (t
        (cons (first list)
                (exchange (- pos1 1) (- pos2 1) (rest list)))) ))

(defun puttogether (moved pos list)
  (cond ((= pos 1)
         (append (list (first list)) (list moved) (rest list)))
        (t (recons      (first list)
                        (puttogether moved (- pos 1) (rest list)) )) ))

(defun recons (first list)
  (append (list (first list)) (list first) (rest list)))

(defun get-first (children) (first children))

(defun cost-first (children) 1)

; example call

(bestfirst (quote (1 2 3 4 5 6 0 8 9 10 7 11 13 14 15 12))
      (quote (1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0)))
```

APPENDIX B. CONTROL SORTED COLLECTIONS - NO HUERISTICS

B.1 0/1 Knapsack Problem

```
(defun getfirst (children) (first children))

(defun make-children-list (position)
  (list (append position (list 0))
        (append position (list 1)) ))

(defun goal-reached-test (position)
  (when (= (length position) (length profits))))
```

```
(defun cost (pos)
  (cost-aux pos 0 0 profits weights))

(defun cost-aux (position w p profit weight)
  (cond ((null position) (estimate w p profit weight))
        ((= 1 (first position))
         (cost-aux (rest position)
                   (+ w (first weight))
                   (+ p (first profit))
                   (rest profit)
                   (rest weight) ))
         (t (cost-aux (rest position) w p (rest profit) (rest weight) )) ))

(defun estimate (w p profit weight)
  (cond ((null profit) p)
        ((> m (+ w (first weight)))
         (estimate (+ w (first weight))
                   (+ p (first profit))
                   (rest profit)
                   (rest weight)))
        (t (+ (* (/ (- m w) (first weight))
                 (first profit))
              p)) ))

(defun valid-first (pos) (valid-aux (first pos) 0 weights))

(defun valid-aux (pos w wlist)
  (if (null pos) (> m w)
      (if (= (first wlist) 0)
          (valid-aux (rest pos) w (rest wlist))
          (valid-aux (rest pos) (+ w (first wlist)) (rest wlist)) )))

; sample call

(setf profits (quote (10 10 12 18)))
(setf weights (quote (2 4 6 9)))
(setf m 15)

(branchbound ())
```

ACKNOWLEDGEMENTS

An earlier version of this work appeared in [4]. The adviser for this earlier work was Ed Ashcroft.

REFERENCES

1. Ashcroft, E. A. and Wadge, W. W. Rx for Semantics, *ACM Trans. Program. Lang. Syst.* 4, 2 (April, 1982), 283-294.
2. Barrett, R., Ramsay, A. and Sloman, A. *POP-11: A Practical Language for Artificial Intelligence.* Ellis Horwood, New York, 1985.
3. Bawden, A. Reification without Evaluation, *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming,* (1988), 342-351.

4. Coven, H. J. A Descriptive-Operational Semantics for Prescribing Programming Languages with "Reflective" Capabilities. Ph.D. dissertation, Computer Science Dept. Arizona State Univ., Tempe, AZ, May 1991.
5. Coven, H. J. Position Paper: Interconnection Reflection, 1993 OOPSLA Workshop on Reflection and Metalevel Architectures, (Oct. 1993).
6. Danvy, O. and Malmkjaer, K. Intensions and Extensions in a Reflective Tower, *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming,* (1984), 327-341.
7. Dijkstra, E. W. On the Cruelty of Really Teaching Computer Science, *Communications of the ACM,* 32, 12 (Dec. 1989), 1398-1404.
8. Friedman, D. P. and Wand M. Reification without Metaphysics, *Proceedings of the 1984 ACM Conference on Lisp and Functional Programming,* (1984), 348-355.
9. Frost, R. A. Using Semantic Concepts to Characterise Various Knowledge Representation Formalisms: A Method Facilitating the Interface of Knowledge Base System Components,*The Computer Journal* , 28, 2 (1985), 112-116.
10. Ghezzi, C., Jazyeri, M. and Mandrioli, D. *Fundamentals of Software Engineering*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.
11. Goldberg, A. and Robson, D. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, MA, 1983.
12. Hall, M. Vnews reply to Terrence Brannon. Message-ID: 4227@aplcen.apl.jhu.edu reply to 32334@auc.UUCP (Dec. 4, 1989).
13. Hoare, C. A. R. An Axiomatic Basis for Computer Programming, *Communications of the ACM,* 12, 10 (Oct. 1969), 576-583.
14. Horowitz, E. and Sahni, S. *Fundamentals of Computer Algorithms.* Computer Science Press, Rockville, Maryland, 1978.
15. Luger, G.F., and Stubblefield, W.A., *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*. Benjamin/Cummings Pub. Co., Redwood City, California, 1993.
16. McDermott, D. V. and Sussman, G. J. The Conniver Reference Manual, MIT Artificial Intelligence Laboratory Memo No. 259a, (May 1972, Updated Jan. 1974), 1-45.
17. des Rivieres, J. and Smith, B. C. Implementation of Procedurally Reflective Languages, *Proceedings of the 1984 ACM Conference on Lisp and Functional Programming,* (1984), 331-347.
18. Smith, B. C. Reflection and Semantics in a Procedural Language. Ph.D. thesis, Computer Science Dept., MIT. Available as Laboratory of Computer Science Technical Report 272, MIT, (1982).
19. Smith, B. C. Reflection and Semantics in Lisp,*Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages,* (1984), 23-35.
20. Tennent, R. D. *Principles of Programming Languages.* Prentice/Hall International, Engelwood Cliffs, NJ, 1981.
21. Wand, M. and Friedman, D.P. The Mystery of the Tower Revealed: A Non-Reflective Description of the Reflective Tower,*Proceedings of the 1986 ACM Conference on Lisp and Functional Programming,* (1984), 298-307.
22. Winston, P.H. and Horn, B. K. P. *Lisp, 3rd Edition*. Addison-Wesley, Reading, MA, 1989.