2
3
NIL

The power of this tool is more in its demonstration of a valid modification from one control structure to another, as a demonstration that this can be validly done. The previous example is a demonstration that reasonable tasks can be accomplished with this. The field of reflection is relatively new and there are not any demonstrations modification of one control structure into a different nonequivalent structure that still has meaning.

No modification of bfs can be made to handle this last example, for with an explicit queue like the one used by bfs, once a node is passed its information is disposed of. Thus the list is disposed of once it is traversed once.

It would be very simple to create an analogous reflective control tool that converts a control structure the opposite way called converttostack.

3 GRAB

Our next control tool allows explicit access to control knowledge. At this point in our research we have created a tool - grab, that access a control stack. In particular by position away from the top of the stack. An argument is given to it that indicates the frame's position from the top that is desired. The different parts of the frame are all accessible by auxiliary functions code, pos and locals. It would be relatively easy to extend this tool to access frames by function name or by function name and local variable value, or other values contained in the control stack. Or to extend the tool to other control structures.

3.1 Operational Semantics

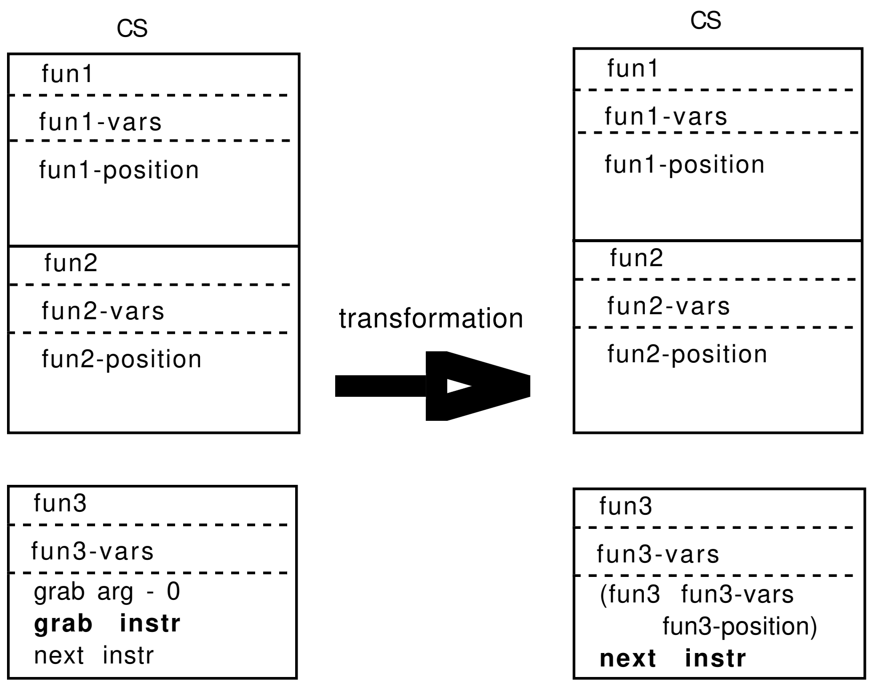
Figure 23 describes the operational semantics of grab, while Figure 24 describes its auxiliary function's: code, locals and pos' operational semantics.

Position is indicated by the most immediate instructions that have been executed or that will be executed plus any arguments that will be used in the execution of those instructions or that are returned by those instructions. Arguments are placed similarly to how a stack machine might stack them. Implementation is highly implementation dependent so only minimal details are given.

Grab

1) Obtain indicated frames and return it as the result value of the grab instruction.

e.g. (grab 0)



The argument to grab - 0 indicates the frame that will be the returned value. 0 indicates the current frame, 1 is the next closest frame to the top of the control stack and so on.

Fig 23. Grab operational semantics.

There are three auxillary functions that return the different parts of a control frame - code, locals and pos. They operate on a frame as indicated below.

```
(code '(fun3 fun3-vars
     fun3-position))              =>        fun3

(locals '(fun3 fun3-vars
     fun3-position))              =>        fun3-vars

(pos '(fun3 fun3-vars
     fun3-position))              =>        fun3-position
```

Fig. 24. Grab auxiliary function operational semantics.

3.2 Recursive Lambda and Other Uses

One use for grab is the construction of recursive lambdas. We saw a function equivalent to a recursive lambda in Section 2.4's function treeintree. A recursive lambda is a function that is unnamed and recursive. The important question becomes how does the function call itself if it has no name to use. As above we use (grab 0) which grabs the current frame and (code ...) which obtains the function form from the frame. In this way (code (grab 0)) returns the function from of the current frame or current function. Thus a function can refer to itself using (code (grab 0)). What follows is a function call using a recursive lambda implementation of the factorial function.

```
qlisp> (funcall (function (lambda (n)
          (if (= n 0)
             1
             (* n (funcall (code (grab 0)) (- n 1))) )))
  7)
5040
```

The above recursive lambda is far more understandable than the more common Y operator rendition given below.

```
> (funcall (let ((Y (function (lambda (f)
                      (funcall (function (lambda (g) (funcall g g)))
                      (function (lambda (x)
                                (funcall f (function (lambda ()
                                                    (funcall x x) ))) )) )) )))

    (funcall Y (function (lambda (fg)
           (function (lambda (n)
                (if (= n 0)
                               1
                               (* n (funcall (funcall fg) (- n 1))) ) ) )))) )
    7)

5040
```

Certainly our recursive lambda has fewer lines of code. But also, there have to be many levels of execution that have to be simulated in order to grasp what the Y operator actually does. It typically takes a not insignificant period of time and a pencil and paper in order to grasp what is happening with the Y operator example. Grab is rather straight forward, the operational model that it depends on is familiar to every programmer who uses function or procedure calling.

Alternatively a "goto" like syntactic recursion example could be used.

```
Lisp> (let ((n 7))
       #1=(if (= n 0) 1
             (* n ((lambda (n) #1#) (- n 1))) ) )
5040
```

However here we are attaching a label #1 to the function. This is exactly what we are trying to avoid with an unnamed function. The purpose behind unnamed function is to have constructs that can be created, modified and used on the fly as often as is desired. Labels have to be generated in a trackable form, what we were trying to avoid.

Additional uses of grab would be whenever information in the control structure such as values in local variables is desired by either the current process or potentially other processes. We feel that the above examples just demonstrate the tip of the iceberg of what can be done with grab. For a programmer of program to have full power it/she/he must be able to access all information that exists in the program and this is what grab provides.

4. ZAP

The next tool is zap which allows explicit manipulation of the control stack. Manipulation of the control stack is by function name. In our implementation each frame's code component has the name of the function, thus matching by name can be done. If the frame's code component is a lambda list then it can't be matched by name but can be matched as a lambda list. We start by showing the operational semantics of zap and then go on to show some uses of zap.

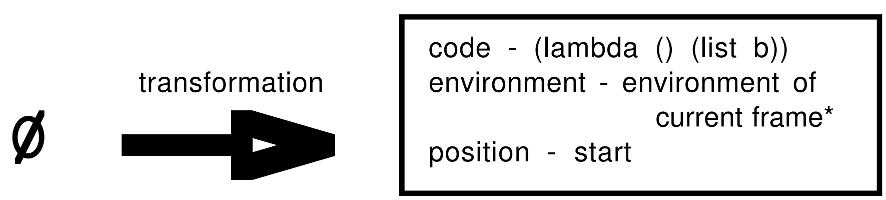
4.1 Operational Semantics

There is an auxiliary function for zap called glue. Glue will create a new frame. Zap can then use the newly created frame and add it to the control structure in any desired location. We describe the semantics of glue in Fig. 25.

Glue

1) Create a frame for the given call including code, position, and local variables.

e.g. (glue (list b))



* - For this implementation we have choosen a style of programming without dynamic or lexical scope. Only local and global variables are permitted. Globals can be referenced and set while locals are only permitted to be referenced.

Fig. 25. Operational semantics of glue.

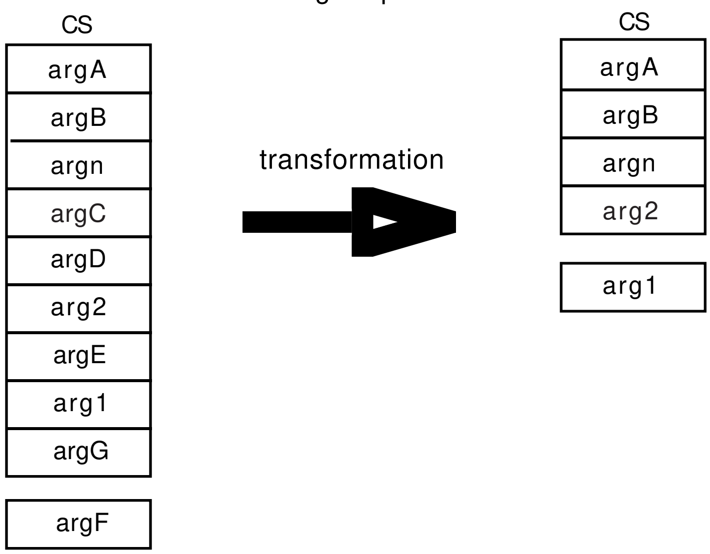
The operational semantics for zap follow in Figure 26.

Zap

1) Remove all frames from the control stack except those that match the arguments and all of those below the last argument.

e.g. (zap arg1 arg2 ... argn)

In the below figures only function names are listed representing the code for each frame. Arg? represents a function name.



e.g. (zap (glue (list b)) arg1 arg2 ... argn)

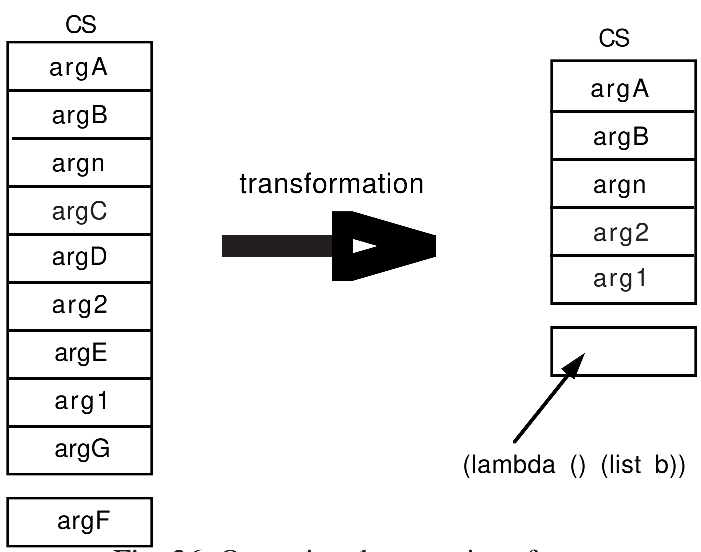


Fig. 26. Operational semantics of zap.

4.2 Comparison with other Exception Handling Mechanisms

When would a programmer want to explicitly manipulate the control structure? Whenever they want to get out of the standard control flow. This is when they have an exception to the standard control flow. We now give examples of exception handling with zap. We will go on to compare zap's exception handling with exception handling mechanisms of Lisp and Pop-11 and demonstrate that zap's exception handling is more modular.

In our first example there are two possible exceptions, the first transfers control back to fun6 the second more extraordinary exception transfers control further back to fun3, but executes frame fun6 along the way. Note that all exception handling is handled in one function. Other types of exception handling mechanisms typical have code strewn all throughout the program instead of modularly situated in one location.

```
qlisp> (defun fun1 (a b)
  (print (cons 'fun1 (fun2 a b))))
FUN1
qlisp> (defun fun2 (a b)
  (cons 'fun2 (fun3 a b)))
FUN2
qlisp> (defun fun3 (a b)
  (cons 'fun3  (fun4 a b)))
FUN3
qlisp> (defun fun4 (a b)
  (cons 'fun4 (fun5 a b)))
FUN4
qlisp> (defun fun5 (a b)
  (cons 'fun5 (fun6 a b)))
FUN5
qlisp> (defun fun6 (a b)
  (cons 'fun6 (fun7 a b)))
FUN6
qlisp> (defun fun7 (a b)
  (cons 'fun7 (fun8 a b)))
FUN7
qlisp> (defun fun8 (a b)
  (if (eq a 'strangevalue)
      (if (eq b 'doubleexception)
          (zap (glue (list b)) fun6 fun3)
          (zap (glue (list b)) fun6))
      '(fun8)))
FUN8

qlisp> (fun1 'strangevalue 'doubleexception)
(FUN1 FUN2 FUN3 FUN6 DOUBLEEXCEPTION)
(FUN1 FUN2 FUN3 FUN6 DOUBLEEXCEPTION)
qlisp> (fun1 'strangevalue 'normal)
(FUN1 FUN2 FUN3 FUN4 FUN5 FUN6 NORMAL)
(FUN1 FUN2 FUN3 FUN4 FUN5 FUN6 NORMAL)
qlisp> (fun1 'normal 'normal)
(FUN1 FUN2 FUN3 FUN4 FUN5 FUN6 FUN7 FUN8)
(FUN1 FUN2 FUN3 FUN4 FUN5 FUN6 FUN7 FUN8)
```

As a nonmodular example of exception handling let's look at catch and throw of Lisp. In many ways catch and throw act like a label and a goto. Catch is the label and throw is the goto. To capture the exception handling of the previous example we have to have exception handling code in four different locations: fun3, fun5, fun6 and fun8.

```
(defun fun1 (a b)
  (print (cons 'fun1 (fun2 a b))))
(defun fun2 (a b)
  (cons 'fun2 (fun3 a b)))
```

```
(defun fun3 (a b)
  (cons 'fun3 (catch 'exception1 (fun4 a b))))
(defun fun4 (a b)
  (cons 'fun4 (fun5 a b)))
(defun fun5 (a b)
  (cons 'fun5 (let ((temp (fun6 a b)))
                 (if (equal (last temp) '(doubleexception))
                (throw 'exception1 temp)
                temp))))
(defun fun6 (a b)
  (cons 'fun6 (catch 'exception2 (fun7 a b))))
(defun fun7 (a b)
  (cons 'fun7 (fun8 a b)))
(defun fun8 (a b)
  (if (eq a 'strangevalue)
     (throw 'exception2 (list b))
     '(fun8)))

> (fun1 'strangevalue 'doubleexception)
(fun1 fun2 fun3 fun6 doubleexception)
> (fun1 'strangevalue 'normal)
(fun1 fun2 fun3 fun4 fun5 fun6 normal)
> (fun1 'normal 'normal)
(fun1 fun2 fun3 fun4 fun5 fun6 fun7 fun8)
```

Pop-11 [2] has the catch and throw exception handling mechanism, but in addition has other control structure manipulation mechanisms. The most modular of which is chainto. With chainto exception handling code need only be located in two locations: fun5 and fun8. However an auxiliary function returnval must be created to grab values off of the stack and pass them along the return chain.

```
define fun8;
  if A = "strangevalue" then
      conspair(B, evallist) -> evallist;
      chainto(fun6, returnval); endif;
  conspair("fun8", evallist) -> evallist;
  enddefine;
define fun7;
  fun8();
  conspair("fun7", evallist) -> evallist;
  enddefine;
define returnval;
  enddefine;
define fun6;
  fun7();
  conspair("fun6", evallist) -> evallist;
  enddefine;
define fun5;
  fun6();
  if B = "doubleexception" then
      chainto(fun3, returnval) endif;
  conspair("fun5", evallist) -> evallist;
  enddefine;
define fun4;
```

```
  fun5();
  conspair("fun4", evallist) -> evallist;
  enddefine;
define fun3;
  fun4();
  conspair("fun3", evallist) -> evallist;
  enddefine;
define fun2;
  fun3();
  conspair("fun2", evallist) -> evallist;
  enddefine;
define fun1;
  fun2();
  pr(conspair("fun1", evallist));
  pr(newline);
  enddefine;

: "strangevalue" -> A; "doubleexception" -> B; [] -> evallist; fun1();
[fun1 fun2 fun3 fun6 doubleexception]
: "strangevalue" -> A; "normal" -> B; [] -> evallist; fun1();
[fun1 fun2 fun3 fun4 fun5 fun6 normal]
: "normal" -> A; "normal" -> B; [] -> evallist; fun1();
[fun1 fun2 fun3 fun4 fun5 fun6 fun7 fun8]
```

Exception handling is a term that really creates a limited view of what is being done here. A more general view would consider it a miracle that we can create a hierarchical structure of function calls to our reasoning process. This is quite opposite from the view that we call fit our reasoning into a hierarchical form with only a few exceptions. The power of zap is that it takes the more general view that valid and useful structures of our reasoning process is more varied. With zap we have more power to reorganize the reasoning structure of our programs, and thus capture more varied reasonings used by the mind.

5 CONTROL SORTED COLLECTIONS

Our final control tool is control sorted collections. We borrow the data structure sorted collections from smalltalk [11]. A sorted collection is similar to a list or stack in that it is a collection of objects. Unlike a list it has an order. However unlike stacks or queues its order is not determined by the order of when elements are added to it. Its order is determined by the values of the elements themselves. An auxiliary function goes along with the sorted collection to indicate how the elements are to be sorted. An example might be the function (lambda (A B) (< A B)) which would sort the collection in an ascending order. The function is used to compare two elements of the collection. The variable A matching with the first element encountered in the collection when starting the sort, B matching with the second element. If the function returns a true value then the value in A is sorted into the first location, otherwise B is sorted into the first location. In this way the lambda function can be used along with any number of different implicitly built-in sort algorithms to sort the collection.

Control sorted collections are similar to sorted collections in that they use a function to sort themselves internally. The elements that are being sorted are the control frames. Like control queues, control sorted collections can be intermixed with normal function calling control stacks. There is an additional argument needed for control sorted collections in comparison to sorted collections. Unlike sorted collections, control sorted collections have elements that are complex. Frames are built up data structures that have several components: position, code and locals. Position and code are not structures that can be compared, while variables can have values that can be compared. But which local variables should be used to do the comparison. The second argument for control sorted collections is a form that indicates the variable or construction of variables that are to be used for comparison.

Control sorted collections can be used similarly to how control queues are used. Control queues are used as a general breadth-first traversal mechanism, control sorted collections can be used as a general best-first traversal mechanism.

5.1 Operational Semantics

In Figure 27 we show the basic structure for control sorted collections. The structure resembles that for control queues with two exceptions. First there is an extra data structure that contains the sort function and variables. Second fragments are not permitted. If there is an unattached frame it must be the top of the stack for the first control sorted collection (CSC) frame. A CSC frame is analogous to a queue frame and is equivalent to a control stack itself because normal functions can be called from within a control sorted collection frame.

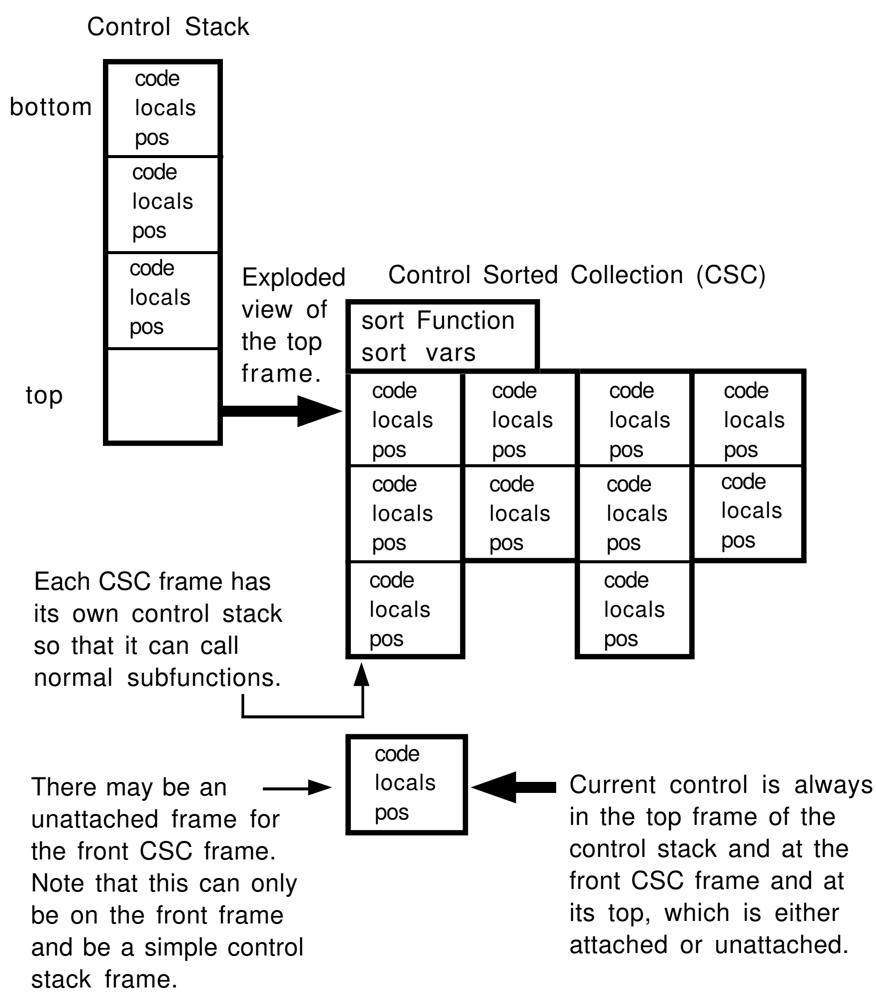
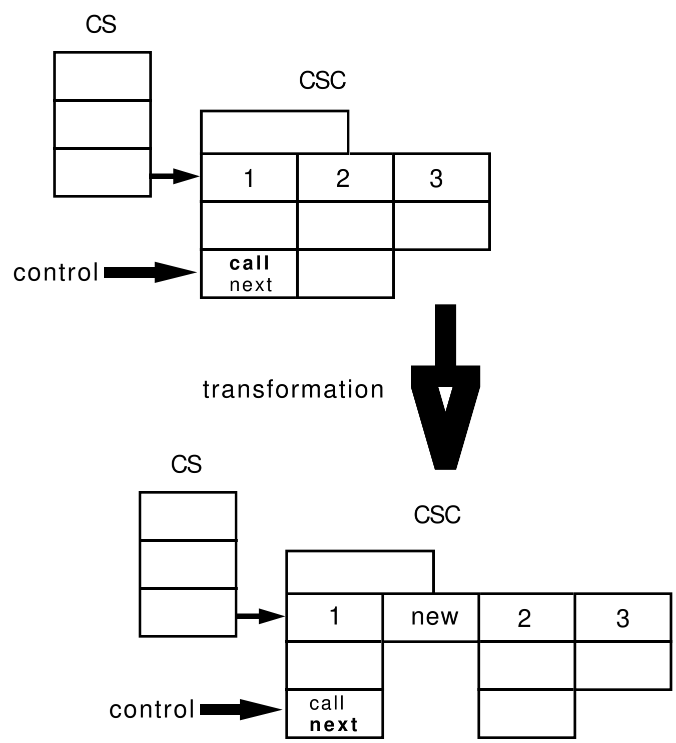


Fig. 27. Basic operational semantics of control sorted collections.

Control sorted collections have an ordinary call and return (Figures 28 and 29 respectively). In addition there are alternative ways of calling and returning from control sorted collections. There is an explicit call function (Figure 30), and an explicit returncf (Figure 31) which terminates all of the frames of control sorted collection at the same time. There is a function similar to glue called newframe (Figure 32) which adds a frame to the control sorted collection. In Figure 33 there is a description of how grab works with control sorted collections. Finally there are two auxiliary functions getframevar and setframe for accessing and setting the local variables of control sorted collection frames respectively (Figures 34 and 35).

Normal function call of a CSC function: the current frame's position is advanced; if the current frame is unattached it is attached; a new frame is created for the new CSC frame and put on the control sorted collection; then the control sorted collection is resorted using the indicated sort variables of each CSC frame and the sort function; control is always maintained at the front of the CSC.



If no control sorted collection exists yet.

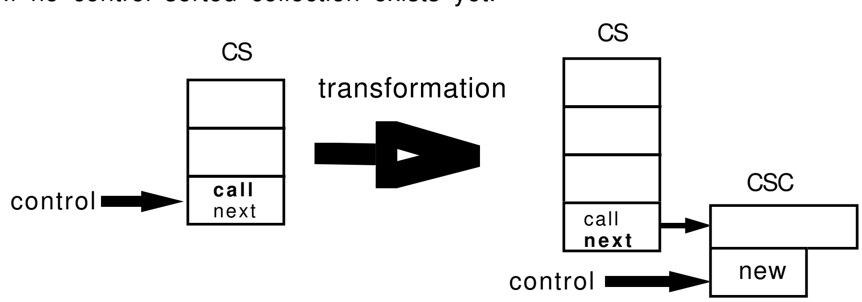


Fig. 28. Operational semantics of control sorted collection's normal call.

Normal function return: the first CSC frame is removed; then the CSC is sorted because more frames may have been added with newframe; if no more CSC frames exist then the entire CSC is removed and control returns to the top of the stack.

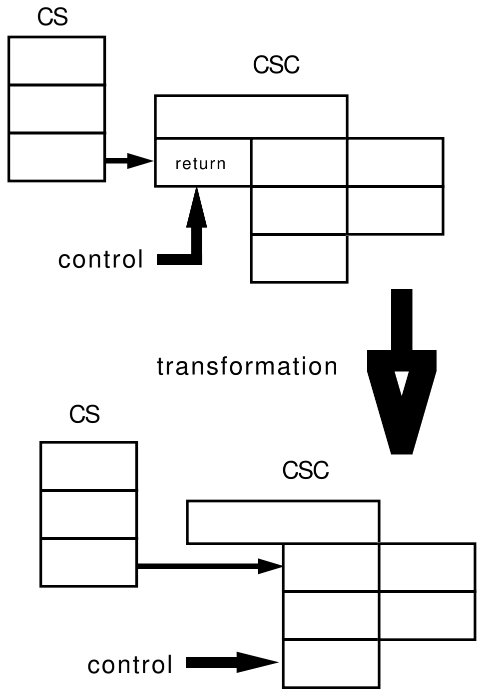


Fig. 29. Operational semantics of control sorted collection's normal return.

The function (call ...): advances local control past the "call"; if the current frame is unattached it is attached; reorders the control sorted collection according to the sort function and sort variables; and places control at the front of the control sorted collection. In the below example non-generic values are used for demonstration purposes.

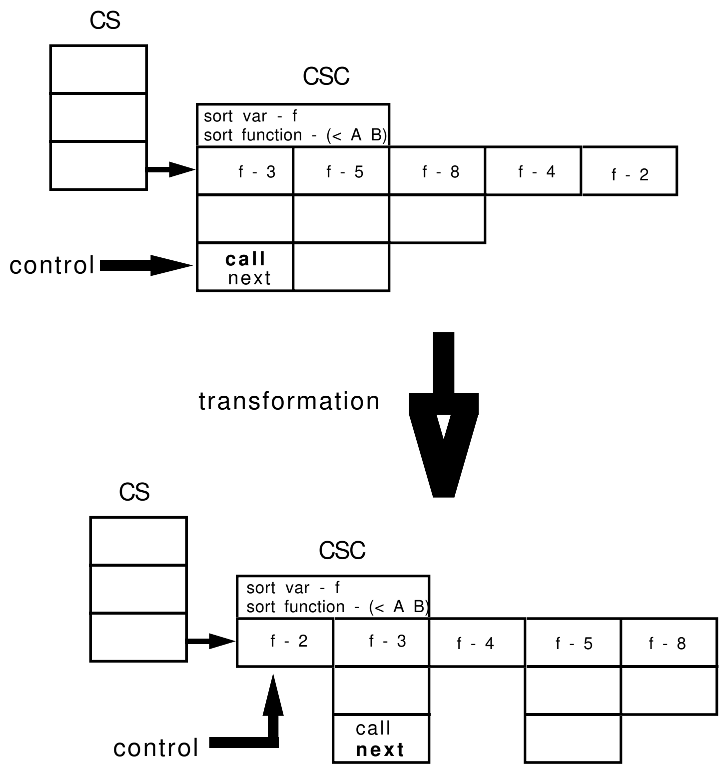


Fig. 30. Operational semantics of control sorted collection's explicit call.

The function (returncf): will return the frame that called the CSC even though there may be additional CSC frames left; all CSC frames will be removed.

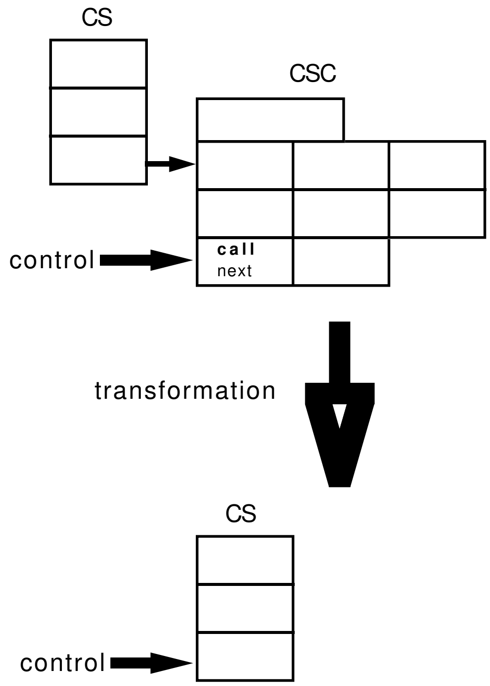


Fig. 31. Operational semantics of control sorted collection's explicit returncf.

The function (newframe ...) adds a new frame to the end of the Control Sorted Collection, Control remains with the front frame. The frames will only be reorganized when there is some form of call on the Control Sorted Collection.

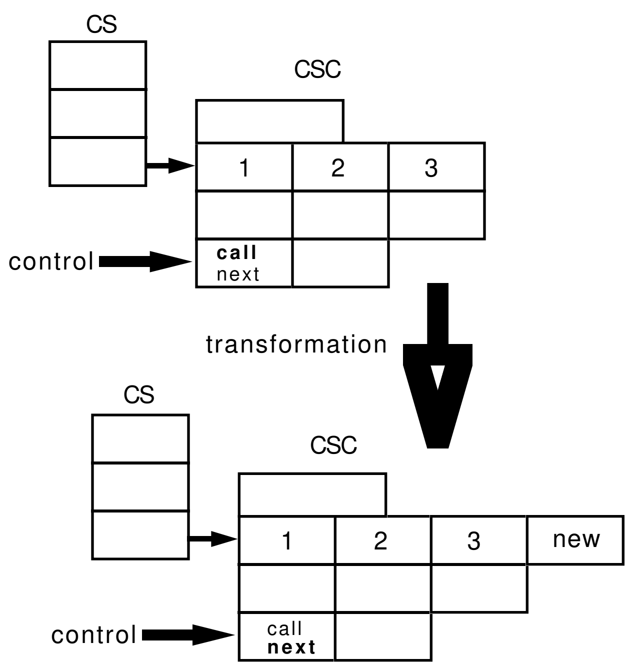


Fig. 32. Operational semantics of control sorted collection's function newframe.

The function (grab CSCfunctionName (var val)) behaves very much the same as grab on an ordinary function. However with a CSC function there is an extra parameter that differentiates CSC frames by a variable - value pair. The first CSC frame with this variable-value pair in the CSC is the value returned just as in a normal grab. Remember however that a CSC frame has the form of an ordinary control stack.

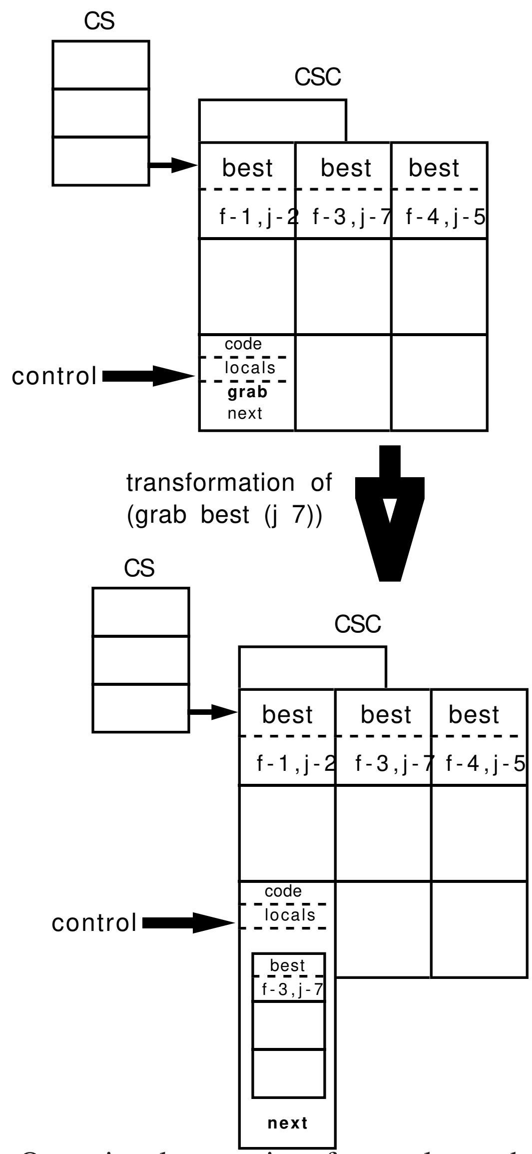


Fig. 33. Operational semantics of control sorted collection's function grab.

The function (getframevar ...) does not work with the control structure, instead it operates like a function taking arguments and returning a value. However it does take a piece of a control structure as its second argument. The second argument control structure will be searched for a frame with the function name or definition of the getframevar third argument, within the found frame the getframevar first argument variable will be searched for in the locals and its value will be returned.

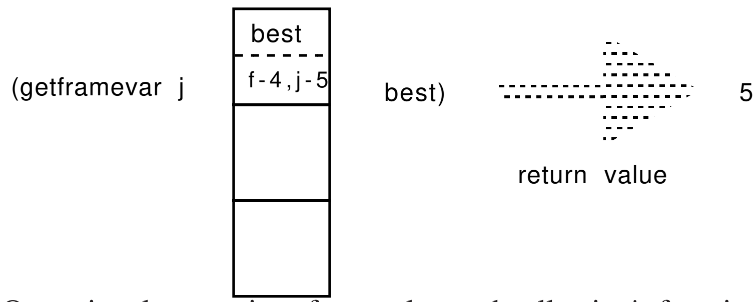


Fig. 34. Operational semantics of control sorted collection's function getframevar.

The function (setframe frameDescription varName varValue ...) will search for the described frame in the control structure and then set the local variable that matches varName to varValue. This function can take any number of optional varName and varValue argument pairs. In the below diagram the arguments to setframe have been stacked in the position structure of a frame as has been done in previous diagrams.

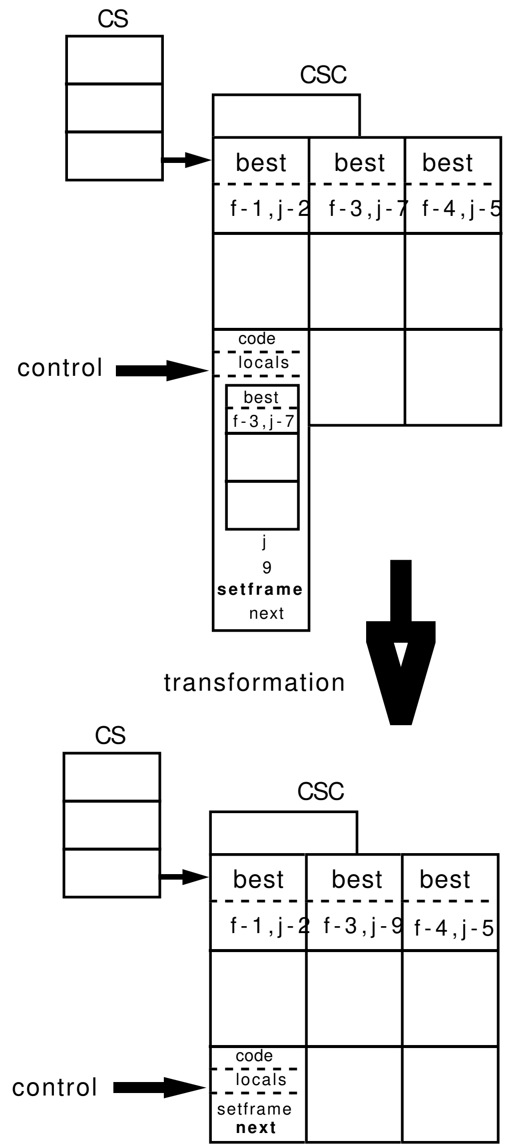


Fig. 35. Operational semantics of control sorted collection's function setframe.

The auxiliary functions setframe and getframevar are really further renditions of zap and grab in that they manipulate and access control structures.

5.2 Best-first Traversal

Best-First traversal examples are by their nature complex. Because of this complexity we will only provide two examples. However we will make these two examples as divergent as possible in order to demonstrate the generality of control sorted collections.

We start with an implementation of the A* algorithm [15] the most famous of best-first search examples. Any A* algorithm will use the same basic structure. Thus we focus on the code that will be generic for numerous different problems, and leave specific code for specific problems for the appendices. As an example we describe the 15 puzzle problem. In this problem we have fifteen numbered tiles on a sixteen tile grid. There is one empty square on the grid. The only movement that is possible is moving a tile adjacent to the empty square (above, below, right, or left of it) into the empty square creating a new empty square where the adjacent tile used to be. The goal of the game is to put the tiles in order from 1 to 15 starting from an arbitrary organization. A heuristic is used indicating how close the current position is from the goal position. This heuristic can be anything as long as the heuristic value of a position is always less than the actual cost in number of moves from the current position to the goal.

The idea of the A* algorithm is to make a best-first search of all possible sets of moves. Starting from the start position make all possible one step moves (expand the position). Calculate a heuristic value for that new position and add the cost of getting to that position. This sum will be the total estimate or value of the position. The position with the best value will be the next position to be expanded. If a position is arrived at that has been met before but the cost of getting to this position is now cheaper than it was before, recalculate the value of this position, if it has already been expanded put it on the list of unexpanded nodes. All unexpanded positions will always compete with each other to be expanded next. In this way whenever the best value position is also a goal position then a best cost solution has been reached.

We now turn this algorithm into code using control sorted collections. A control sorted collection is defined using defsc, which is analogous to a defq. The following two functions are the generic part of the A* algorithm. For demonstration purposes we have made a number simplifications. We eliminate a concern for space thus a path indicating how we got to a position needs to be wholly stored for each node. This allows us to eliminate a closed list which is usually used by the A* algorithm to hold partial paths from one node to another node, in that way space can be dramatically saved in the storage of paths. Partial paths that appear in many different nodes don't have to be repeated. We also assume that the goal will be reached by some path.

```
(defsc bestfirst (Position Goal &key (G (getEstToGoal Position Goal))
                                        Path (F  0) )
                (+  F  G)               ; value sorting is done on
                (function <)            ; sort function
  (if    (eq position goal)
         (progn (print path) (print F) (returncf))
         (generate-children position goal path F  (make-children-list position))  ) )

(defun generate-children (position goal path F children)
 (unless (null children)
         (setf temp (grab bestfirst (list (quote position) (get-first children))))
         (if (null temp)
             (newframe bestfirst  (get-first children)              ; newly created position
                                  Goal
                                 :path (append path (list (get-first children)))
                                 :F (+ F (cost-first children)) )
                                                                    ; already on open list
             (when (<  (+ F (cost-first children)) (getframevar F temp))
                    (setframe  temp path  (append path  (list (get-first children)))
                                      F  (+ F (cost-first children)) ) )))
         (generate-children position goal path F (rest children))))
```

Bestfirst has five variables, a sort structure, and a sort function. The first variable "position" holds the current position such as the tile organization on the 16 square board (this could be a list of 15 numbers and a 0 for the empty square). "Goal" holds the goal position, this might be the list (1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0). The next three variables are key variables and if not passed by name will take on the value in their associated list or nil if there is no associated list. "G" is the estimate to the goal from the current position. This value is computed by a function for the specific problem at hand, not a generic function, getEstToGoal appears in two examples in Appendix A. "Path" is a list of positions passed through to get to this location. "F" is the cost to get to this position from the starting position. Thus the value of this position is the sum of "F" and "G".

The value that is used to sort the positions or frames is (+ F G). Thus the sort structure contains a list of these variables (+ F G). The positions will be sorted in ascending order. The best-cost position will be the position or frame that will be expanded next.

Next we come to the body of bestfirst. If the position and goal are the same then the best-cost position is a goal position, and we have found a best-cost solution and are done. We need only print out the path used to get to this position and terminate the search using returncf which closes the control sorted collection. Otherwise we need to generate the children of this position (all the possible moves from this position). We then terminate this node by doing an ordinary return. by returning the next best-cost position-frame can take control.

The generate-children function receives a list of all of the child positions from the example specific make-children-list, again two examples of this function appear in Appendix A. Generate-children is a recursive function. If there are no more children in the children variable it terminates execution. Otherwise the first child position is compared with existing frame to see if this position has been reached before and is unexpanded. If it is found it is put in a temporary variable called temp. Notice that the control sorted collection function grab is used. If the grab puts NIL into temp then no previous frame matching this position has been found and the first cond test (null temp) will succeed. In this case a new position is generated with newframe. All the appropriate variable values are passed along to newframe. If an identical position is found in the control sorted collection then there needs to be a check to see if the new path to this position is a better cost than the previous path, if so then the positions cost values need to be updated. The function getframevar is used to check the old cost values, and setframe is used to set the new cost value.

The fifteen puzzle example in Appendix A.2. A shortest path example appears in Appendix A.1.

If we didn't have control sorted collections then control flow would have to be explicitly managed either with a sorted collection, or with a sort algorithms provided by the programmer him/herself. The idea behind control sorted collections is to relieve the programmer of this drudgery. With this tool we are not concerned with efficiency of a running program but rather efficiency of the programmer. Alternatively we could have created a specific mechanism for the A* algorithm with specific tags into the algorithm such as the functions: getEstToGoal and make-children-list which we have mentioned before as well as cost-first and get-first which are additional example specific functions that appear in the bestfirst function. However the problem with that type of structure is similar to the problems we had with bfs. There are numerous situations where the specific structure of a generic set of defined functions simply will not work. First the form of the variables may be different. Second the input functions may have to vary, in our experiments we found that an additional example specific functions would need to be created for printing the path, since calls to the example specific function getfirst appears in two places - one for path and one for position we would need two getfirst functions instead of one. Finally there might need to be a completely different set of variables and algorithm altogether. We next provide the 0/1 knapsack [14] problem as a divergent problem requiring a simpler algorithm and set of variables.