The Control Structure Facilities of ECL

Charles J. Prenner

Center for Research in Computing Technology
Harvard University

1. Introduction

ECL is a programming language system currently under development at Harvard University. A general description of the system is given in a companion paper [1]. This paper will describe some of the control structure facilities available in ECL - a complete description will appear as part of the author's Ph.D. thesis.

Most existing programming languages allow for only one path of control. The programmer is presented with an environment in which he may use conditional statements, iteration statements and nested function calls (perhaps called recursively) to modify the flow of control through his program but it is not usually possible to either suspend execution of the current program (control path) and start up another or to set up another control path to execute in parallel with the current one. EL1, as originally proposed in [2], allows for only one path of control. ECL, however, allows for the creation and manipulation of multiple paths of control. These paths may execute in parallel, or act as coroutines or in any other relation desirable. Since the number of paths which can be created to execute in parallel may be greater than the number of processors available on a given machine, some sort of path scheduling must be done. The path scheduler is written in EL1 and is available for redefinition by the programmer.

The following sections will discuss:

1) what an ECL path is,
2) a distinguished path which controls both path scheduling and the flow of control between paths,
3) primitives for manipulating paths,
4) examples of multi-path control.

2. Paths

A path is the computational environment created by the execution of a (sequential) EL1 procedure. Associated with each path is an environment and a unique activation record. The environment includes the current nesting of function calls and the name-value pairs associating formal parameters with actual parameters and local variables with current values. The activation record (ACTRC) is defined as an EL1 STRUCT. All ACTRCs are allocated in the heap and thus must be referenced by pointers. The mode ARPTR is defined as a PTR(ACTRC) for this purpose.

Paths may share common data structures. The sharing may be accomplished by referencing the same global variable (i.e. assignments made to variables defined in the "top level" environment) or by passing pointers to objects in the heap between paths. Note that it is not the pointers themselves but the objects pointed to which are shared.

A path may be created by calling the primitive function GET. This function takes an integer argument specifying the amount of core (in K) to be allocated initially for the paths environment (stacks). GET returns an ARPTR for the newly created path.

Two functions can be used to initiate a computation in a path: PAP and PAPQ (path-apply). The relation between these two functions is the same as the relation between SET and SETQ in LISP - the former evaluates its first argument while the latter does not, i.e.

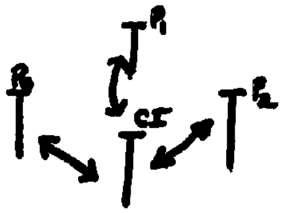
PAP(QUOTE(FOO(X,Y),P)= PAPQ(FOO(X,Y),P)

PAP (or PAPQ) takes two arguments: the first argument is a function call to be applied in the path which is its second argument. The function name and the arguments to the function are evaluated in the current paths environment (e.g. FOO, X, and Y above), the environment of the path which is the second argument to PAP (e.g. P above) is modified so that when (or if) control passes to it, the function (FOO) will be called on its arguments (X, Y). It is not necessary for a path to be newly created in order to PAP a function call into its environment. PAP may be used to apply a procedure in the environment of a path which has already started a computation (see example 1 for a use of PAP in this context).

Once a path has been created and a function call PAPed into its environment it is ready for execution. The mechanism for passing control between paths and executing paths in parallel will be discussed in the next section.

3. Control Interpreter

There exists one distinguished path in ECL - the control interpreter (CI) path. This path contains in its environment a queue of all paths which would be executing in parallel if there existed enough processors - thus it has the ability to act as a path scheduler. In addition, the CI acts as a control switchyard for other paths in the system. No path can pass control directly to any other path - it must communicate the control via the CI path.



If a path cannot proceed (for any reason) it passes control to the CI. The CI will then examine its queues, choose a new path to run and then pass control to that path. Control is passed from a path to the CI path by executing a primitive function CIA - (control-interpreter-apply). CIA takes two arguments - the first is the name of a procedure to be executed in the CI environment, the second is the argument to that procedure. The interpretation is as follows: control is to be passed from the current path to the CI path (as soon as the CI path is free - i.e. no control resides in the CI path) and the function is applied to its argument in the environment of the CI.

The following variables are declared in the C environment:

LASTRUN (an ARPTR) is the path which has passed control to the CI.

WRUNQ (a STRUCT(FIRST:ARPTR,LAST:ARPTR)) is a queue of those paths which would be running if there were enough processors.

When control is passed to the CI for a CIA call, the CI applies the function to its argument and then checks the value of LASTRUN. If it is NIL then the CI chooses a new path to run from the WRUNQ (see example 2), if it is not NIL then the CI passes control to the path specified by LASTRUN - which may be a different path from the one which executed the CIA call (see example 1).

4. Primitive Functions

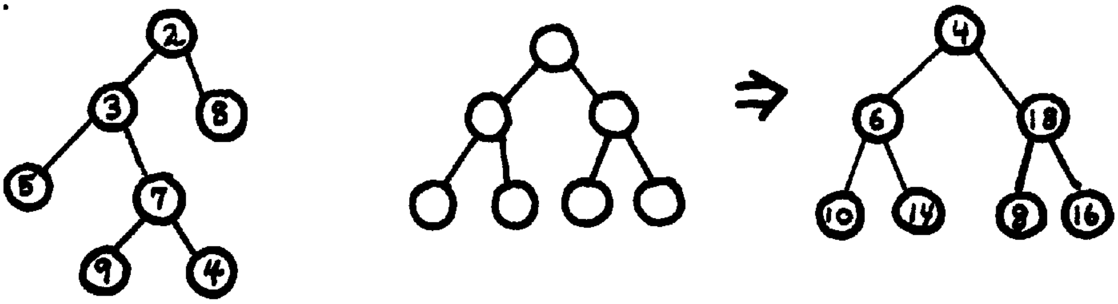
Four primitive control functions have been described in the previous sections: GET, PAP, PAPQ and CIA. There exist three more functions which are primitive in the system:

1) RETFROM (FNAME,VALUE) - return from the most recent call to the function FNAME with VALUE as result,
2) DELETE (P) - indicate that the path P is no longer eligible for running (i.e. an error occurs if a path attempts to pass control to P),
3) MYPATH () - returns an ARPTR to the activation record of this path.

5. Example 1: Tree Walks Using Coroutines

Consider the following problem: given two binary trees x and y, where x and y have the same number of nodes but not necessarily the same structure, walk each tree in prefix order and assign to each node of y two times the node value of the corresponding node of x.

e.g.



x y new y

The date structure definitions are[1]:

```
TREE←TREE::PTR("NODE");
NODE←NODE::STRUCT(LS:"TREE",RS:"TREE",NODE\VAL:INT);
```

To solve this problem we will define a procedure TREE\DOUBLE which will create two new paths px and py - making a total of 3 paths, including the path in which TREE\DOUBLE is called (which we will refer to as p_0) p_0 will call upon px and py as coroutines. When called, px (py) will return a pointer to the next node in the prefix walk of the tree x (tree y). Note that since px and py are separate paths they retain their internal state upon returning the next node to p_0.

TREE\DOUBLE is defined as follows:

```
TREE\DOUBLE←EXPR(X:TREE,Y:TREE;TREE)
          BEGIN
          DECL PX,PY:ARPTR;
          DECL NX,NY:TREE;

     [1] PX←COCALL(WALK(X));
     [2] PY←COCALL(WALK(Y));

     [3] LOOP: NX←RESUME(PX,NIL);
     [4] NY←RESUME(PY,NIL);

     [5] NX=NIL =>[)    DELETE(PX);
                        DELETE(PY);
                        Y (];

     [6] VAL (NY).NODE VAL←2*VAL(NX).NODE VAL;

     [7] GOTO     LOOP

         END;
```

[1] COCALL creates a new path to be called as a coroutine. The new path (when started) will apply the procedure WALK on tree X.
[2] same as [1] for tree Y.
[3] Resume coroutine PX. Control leaves this path and PX is restarted. When PX resumes this path it will pass back (as the "result" of the procedure RESUME) a pointer to the next node of tree X which will be assigned to NX.
[4] same as [3] for tree Y.
[5] If PX returns NIL then the entire tree has been walked. Delete paths PX and PY and return from TREE\DOUBLE with Y as result.
[6] make the node of Y be two times the node of X.
[7] Loop.

[1] Terminal nodes are represented by NIL LS and RS links

COCALL is defined as follows:

```
            COCALL←EXPR(Ƶ:FORM UNEVAL;ARPTR)
                    BEGIN
                    DECL P:ARPTR;
                [1] P←GET(1);
                [2] P.ANC←MYPATH();
                [3] PAP(Ƶ,P);
                [4] P
                    END;
```

[1] Create a path P.
[2] The STRUCT definition of ACTRC contains a field ANC (ancestor) which is an ARPTR. Store in P the fact thay my path created it. Note that using ANC avoids passing an extra argument to WALK.
[3] PAP into path P a call to the procedure (i.e. WALK(X) or WALK(Y)).
[4] Return P as result.

RESUME is defined as follows[1]:

```
            RESUME←EXPR(PATH:ARPTR,VAL:ANY;ANY)
                    BEGIN
                [1] PAPQ(RETFROM("RESUME",VAL),PATH);
                [2] CIA("SWITCH\PATHS",PATH)
                    END;
```

[1] Apply the procedure RETFROM in the path to be resumed. The procedure to be returned from is RESUME and the result that the call on RESUME should return is VAL.

[1] The definition of RESUME which was actually used when these functions were run in ECL differs slightly from the definition of RESUME given above. Currently PAP just sets up an evaluation of the form which is its first argument in the environment of the path which is its second argument. Thus it is necessary to bind VAL to VALUE (a global variable) to correctly pass the next node of the tree to the path to be resumed:

```
            RESUME←EXPR(PATH:ARPTR,VAL:ANY;ANY)
                    BEGIN
                    VALUE←VAL;
                    PAPQ(RETFROM("RESUME",VALUE),PATH);
                    CIA("SWITCH\PATHS",PATH)
                    END;
```

[2] Call upon the CI to pass control from this path to the path to be resumed. When control passes to the resumed path it will execute the RETFROM from the call to RESUME in its environment and return VAL as result of the call. Note that this path is left in a state such that when another path tries to resume it the current call to RESUME is the one which will be returned from. Also note that the first time PX and PY are resumed there are no calls to RESUME in their environments to return from. This presents no problem since RETFROM has no effect (returns NOTHING) if no call to the function is found in the function call environment. In this case, control simply "falls through" to the call on the procedure WALK.

```
SWITCH\PATHS←EXPR(Q:ARPTR;NONE)
      BEGIN
      DECL LASTRUN:ARPTR BYREF LASTRUN;
      LASTRUN←Q
      END;
```

SWITCH\PATHS merely modifies LASTRUN to be the path to be resumed.

```
WALK←EXPR(T:TREE;NONE)
      BEGIN
   [1] WALK1(T);
   [2] RESUME(MYPATH().ANC,NIL)

      END;
```

```
WALK1←EXPR(T:TREE;NONE)
      BEGIN
   [1] T=NIL => NOTHING;
   [2] RESUME(MYPATH().ANC,T);
   [3] WALK1(T.LS);
   [4] WALK1(T.RS)
      END;
```

WALK [1] Call upon the auxilliary procedure WALK1 to perform the actual prefix walk.
[2] When WALK1 returns, the tree walk is complete. Resume p_0 with NIL to indicate completion (see TREE\DOUBLE [5]).

WALK1 [1] T=NIL implies that we have tried to walk from a terminal node - thus return NOTHING.
[2] Resume p_0 passing it a pointer to the node.
[3] Call WALK1 recursively on the LS link.
[4] Call WALK1 recursively on the RS link.

6. EXAMPLE 2 - SEMAPHORES

Semaphores [3] and their associated operations P and V are useful for mutual synchronization of processes. P and V can be defined easily in ECL as shown below.

The data structures are:

```
ARQPTR←ARQPTR::STRUCT(FIRST:ARTPR,LAST:ARPTR);

SEM\ELT←SEM\ELT::STRUCT(COUNT:INT,WLIST:ARQPTR);

SEM←PTR(SEM\ELT);
```

The P operation is defined as follows:

```
P←EXPR(X:SEM;NONE)
        BEGIN
        DECL Y:SEM\ELT BYREF VAL(X);
    [1] MYPATH() # PCIAR => CIA("P",X);
    [2] Y.COUNT←Y.COUNT-1;
    [3] Y.COUNT GE Ø => NIL;
    [4] ENTERL(LASTRUN,Y.WLIST);
    [5] LASTRUN←NIL
        END;
```

[1] If my path is not the CI path (PCIAR is a global variable which points to the CI's ACTRC) then call upon the CI to execute P(X).
[2] Subtract one from the semaphore's count.
[3] If the count is greater than or equal to zero then no work has to be done.
[4] If the count is less than zero then enter the path which executed the CIA onto the queue of paths associated with the semaphore.
[5] Set LASTRUN to NIL to indicate to the CI that this path can no longer run. The CI will choose some other path to run.

The V operation is defined as follows:

```
V←EXPR(X:SEM;NONE)
        BEGIN
        DECL Y:SEM\ELT BYREF VAL(X);
        DECL Ƶ:ARPTR;
    [1] MYPATH() # PCIAR => CIA("V",X);
    [2] Y.COUNT←Y.COUNT+1;
    [3] Y.COUNT GT Ø => NIL;
    [4] Ƶ←Y.WLIST.FIRST;
    [5] Y.WLIST.FIRST←Y.WLIST.FIRST.NEXT;
    [6] Y.WLIST.FIRST=NIL→Y.WLIST.LAST←NIL;
    [7] ENTERL(Ƶ,WRUNQ)
        END;
```

[1] If my path is not the CI then call the CI to execute V(X).
[2] Increment the semaphore count by 1.
[3] If the count is greater than zero than there is no work to be done.
[4] Z is bound to the first path waiting upon the semaphore
[5] [6] Remove Z from the WLIST (ACTRCs are linked through a a field NEXT: ARPTR).
[7] Put Z on the queue of paths which may be run.

7. OTHER FEATURES

The above sections give a brief description of the control structure facilities available in ECL. The system also contains a number of facilities which are beyond the scope of this paper. These include: handling of external interrupts, monitoring of variables, and the ability of a path to gain an explicit handle on its environment.

REFERENCES

[1] Wegbreit, B. "An Overview of the ECL System," Proc. of the International Symposium on Extensible Languages, SIGPLAN Notices, Volume 6, Number 12, (December, 1971).

[2] Wegbreit, B., "Studies in Extensible Programming Languages," ESD-TR-70-297, Harvard University, Cambridge, Massachusetts, May 1970.

[3] Dijkstra, E.W., "Cooperating Sequential Processes," in Programming Languages, ed. by Genuys, Academic Press, New York, 1968.