New Control Structures in Icon*

Stephen B. Wampler

TR 81-1a

July 1981

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported by the National Science Foundation under Grant MCS79-03890.

New Control Structures in Icon

1. Introduction

Expressions in Icon [7] are capable of generating sequences of results during the course of their evaluation. For this reason, expressions in Icon are often referred to as *generators*. The primary control mechanism that exploits generators is goal-directed evaluation [12], which provides control backtracking [8]. This paper describes additional control mechanisms that have been added to Version 4 of Icon [2], an extension of Version 3 [1].

These mechanisms add increased usefulness to generators. Besides increasing the programmer's control over generators, applications include the development of user-defined generators at the expression level, generators of unlimited scope, and coroutine processing.

2. New Control Structures

2.1 Explicit Limitation

Several control structures still implicitly limit generators to their first result. For example, the control clause of if-then-else is limited to producing at most one result. Generators may be explicitly limited with the *limitation control structure*.

The syntax for explicit limitation is

expr1 \ *expr2*

which limits *expr1* to at most *expr2* results. For example,

```
every write(!x \ 3)
```

writes the first three elements of x.

2.2 Repeated Evaluation

The *repeated evaluation* control structure repeatedly evaluates its argument. The syntax for repeated evaluation is

|*expr*

For example,

```
every writes(|(1 to 3))
```

prints

```
1231231231231...
```

Note that the argument of repeated evaluation is not limited.

|*expr* is *not* equivalent to the infinite expression

expr | *expr* | *expr* | ...

In |*expr*, evaluation terminates if any evaluation of *expr* fails. Hence, while the expression

```
|1
```

produces an infinite sequence of results, each consisting of the integer 1, the expression

```
|(1 > 3)
```

fails to produce a result. Likewise

```
|move(1)
```

terminates once the cursor reaches the end of the subject.

Because of this aspect of repeated evaluation,

```
|("text" ~== read())
```

terminates if a line is input consisting of the word text, despite the fact that the argument might subsequently succeed if evaluated again.

Repeated evaluation permits the construction of user-defined generators at the expression level. For example, if afile is a file, the operation of

```
|read(afile)
```

is identical to that of the supplied operation !afile.

A generator that repeatedly cycles through the first three primes is

```
|(2 | 3 | 5)
```

As final example, consider

```
|(s1 | s2)
```

this expression generates the infinite sequence of results s1, s2, s1, s2, s1, ... It may be used, as in

```
s1 := s2 := ""
i := 0
every |(s1 | s2) \ *s ||:= s[i+:=1]
```

to decollate the string s into strings s1 and s2. Note that explicit limitation is used to limit the infinite sequence of results to the size of the string s. An alternative method is to rewrite the every expression as

```
every |((s1 | s2) ||:= s[i+:=1])
```

3. Co-Expressions

As stated earlier, generators in Icon are limited by the syntax of the language. This has the advantage of providing straightforward means of controlling generators, as well as permitting efficient implementation [12]. Further, the "first-in, last-out" activation of nested generators makes generators well suited to combinatorial applications [8].

However, this has the disadvantage of fixing the evaluation of a generator to a single lexical point within a program. For example, if alist is a list, then !alist is a generator for producing the elements in the list. As such, it can be used as in

```
every write(!alist)
```

to print the elements in the list. However, because of the restrictions described above, this generator cannot be used in a straightforward manner to print, for example, only every other element in alist.

Version 4 provides a mechanism for extending the scope of a generator beyond a single syntactically fixed point. This permits generators to be used in a wider variety of applications than is possible in Version 3. In addition, this mechanism provides facilities at the expression level for developing evaluation strategies similar to those provided at the procedure level in languages with co-routines.

3.1 Co-Expression Creation and Activation

The expression

```
create expr
```

creates a *co-expression* for evaluating **expr**. A co-expression is a data object consisting of an expression and an environment in which to evaluate that expression. This environment becomes a separate copy of the environment in which the create is performed, including copies of any dynamic local identifiers. As such, it contains the state information necessary for the evaluation of the expression, independent of surrounding context.

Unlike conventional expressions, which are evaluated in environments that are lexically restricted to fixed points in a program, an expression within a co-expression may be *activated* wherever a result is desired from the evaluation of the expression. If x is a co-expression, then @x activates the expression to obtain a result.

For example, evaluation of the following expression assigns to the identifier x a co-expression for producing the elements of alist.

```
x := create !alist
```

Activations of x produce successive elements from alist. For example

```
while write(@x)
```

writes all the elements of alist. Activation of a co-expression fails whenever the co-expression is unable to produce any results. Hence, in the above example, activation of x fails after all the elements of alist have been generated. Once activation of a co-expression has failed, subsequent attempts to activate the same co-expression also fail.

Using the co-expression x, writing the even-numbered elements of alist may be accomplished with

```
while @x do
   write(@x)
```

The activation operator itself is limited to at most one result. Hence, only one result is produced by the expression

```
every write(@x)
```

However, repeated evaluation may be applied to a co-expression to achieve the effect of "unlimiting" activation. For example, the following expression writes all the elements of alist.

```
every write(|@x)
```

As another example, consider the generator find(s1,s2). Creating a co-expression for find(s1,s2) permits the results from evaluating find to be obtained when they are needed and where they are needed. Hence

```
x := create find("ab", "abra cadabra")
write("The first is at ", @x)
write("The second is at ", @x)
```

outputs

```
The first is at 1
The second is at 9
```

Note that without co-expressions find(s1,s2) is re-evaluated each time it occurs. Thus

```
write("The first is at ",find("ab", "abra cadabra"))
write("The second is at ",find("ab", "abra cadabra"))
```

outputs

```
The first is at 1
The second is at 1
```

3.2 Operations on Co-Expressions

If x is a co-expression, then

```
*x
```

is the number of results that have been produced from x. For example,

```
x := create find("text",!file)
while write(@x)
count := *x
```

outputs the column positions of the string text in file, and assigns to count the number of occurrences of text.

The *refresh* operation (^x) returns a copy of the co-expression x with the environment portion being the original environment of x. Hence

```
x := create find("ab", "abra cadabra")
write("The first is at ", @x)
write("The second is at ", @x)
x := ^x
write("The first is still at ", @x)
```

outputs

```
The first is at 1
The second is at 9
The first is still at 1
```

3.3 Uses of Co-Expressions

The following examples show some applications of co-expressions.

3.3.1 Unbounded Selection

One of the simplest uses of co-expressions is in forming *unbounded* selection operations whose scopes are not limited to a single site of evaluation. For example, the following code segment decollates a list, alist, into two lists, odd and even. The elements of odd are the elements of alist with odd indices, and the elements of even are those with even indices.

```
blist := create !alist
odd := list()
even := list()

while put(odd, @blist) do
   put(even, @blist)
```

Note that this process can be expressed more succinctly by replacing the while loop with

```
every |put(odd | even, @blist)
```

Similarly, the string decollation example given earlier may be rewritten

```
s1 := s2 := ""
nextchar := create !s
every |((s1 | s2) ||:= @nextchar)
```

3.3.2 Generative Co-Expressions

Because co-expressions exist indefinitely, many algorithms may be written using generators that could not be written in this way otherwise.

Consider a procedure for supplying successive integer values. It is simple and natural to express this as a generator, as in

```
procedure incr(n)

   repeat {
      suspend n
      n +:= 1
      }

end
```

Creating separate co-expressions for several different invocations permits use of this generator within several independent co-expressions. For example, this procedure can be used as shown below to create a label generator that generates successive labels of the form L*nnn* (starting with L010).

```
genlab := create "L" || right(incr(10), 3, "0")
```

(Note that the precedence of create is identical to the precedence of repeat.)

At the same time, a second co-expression may use the same generator as in

```
nextint := create incr(0) % maxcycle
```

to repeatedly cycle through a sequence of integers.

The nodes of a linked list can be represented with records declared by

```
record lnode(value, link)
```

A generator for sequencing through elements of a linked list is shown below. Creating an co-expression that invokes this procedure results in an unbounded selection operation for generating the elements from a linked list.

```
procedure nextelement(llist)

   repeat {
      suspend .llist.value
      llist := .llist.link | break
      }

   fail
end
```

Co-expressions permit the separation of an algorithm from the situations in which it is to be used. This generally results in clearer, more concise code. For example, there are many applications, such as the "same fringe" problem [11] that require access to the leaves of a binary tree. If the nodes of a binary tree are represented with records declared by

```
record node(data, ltree, rtree)
```

then the procedure

```
procedure leaves(t)

   if /t.ltree & /t.rtree then return t.data
   suspend leaves(\t.ltree | \t.rtree)

end
```

generates the values of the leaves of the tree. (Values are generated because suspend dereferences its argument.) The operation of the procedure depends upon the fact that node fields have no value until one is assigned.

leaves may be used in any application requiring access to the values of the leaves of a tree, as in

```
every write(leaves(tree))
```

By creating a co-expression of an invocation of leaves, this same procedure may be used as an unbounded selection operation. For example, the following code is equivalent to the every expression given above.

```
nextleaf := create leaves(tree)
   .
   .
   .
while write(@nextleaf)
```

In turn, unbounded selection permits the procedure to be used in more complex situations, such as in the procedure compare given in the next section.

3.3.3 Controlling the Evaluation of Multiple Generators

Goal-directed evaluation provides a *cross-product* [8] form of analysis when several generators are present in the same expression. This cross-product analysis is effectively a depth-first search for results among a set of possible results. While goal-directed evaluation is extremely useful in combinatorial applications, it provides little assistance in situations where the results need to be interleaved. By permitting the order of evaluation of generators to be specified by the programmer, co-expressions provide the capability of interleaving the results of generators.

A procedure may activate two or more co-expressions in parallel, providing dot-product analysis to complement the cross-product analysis provided by simple goal-directed evaluation.

For example, in a solution to the same-fringe problem, two or more trees can be walked in parallel to determine if their leaf nodes have the same values in the same order. The following procedure determines if two trees have exactly the same leaves in the same order. If so, the procedure returns the number of leaves in each tree.

Note that when the repeat loop terminates, the same number of activations have been attempted on the two co-expressions. If the number of results supplied by the co-expressions differs, then the trees have an unequal number of leaves.

```
procedure compare(tree1, tree2)
local leaf1, leaf2, nextleaf1, nextleaf2

   nextleaf1 := create leaves(tree1)
   nextleaf2 := create leaves(tree2)

   repeat {
      leaf1 := @nextleaf1 | {@nextleaf2; break}
      leaf2 := @nextleaf2 | break
      if leaf1 ~=== leaf2 then fail
      }

   return *nextleaf1 = *nextleaf2

end
```

If e1 and e2 are two co-expressions

```
every |(@e1 | @e2)
```

alternates activations of the two co-expressions. The results generated by

```
|(@e1 | @e2)
```

consist of alternating results from e1 and e2. If activation of either co-expression fails, the remaining co-expression continues to produce results until its activation also fails.

For illustrative purposes, (there are more efficient methods using character sets and string mapping techniques [4]), consider a procedure merge that interleaves the characters from two strings. If the strings are of equal length, then merge collates the two strings. If the strings differ in length, then the extra characters in the longer string are appended to the resulting string.

The procedure merge may be written as shown below

```
procedure merge(s1, s2)
local e1, e2, s

   e1 := create !s1
   e2 := create !s2
   s := ""

   every s ||:= |(@e1 | @e2)

   return s
end
```

The technique of using repeated evaluation to interleave activations of co-expressions naturally generalizes for any number of co-expressions. For example, code for interleaving four strings is obtained by the replacement of the repeated evaluation expression given above with

```
|(@e1 | @e2 | @e3 | @e4)
```

3.4 Co-Expressions as Coroutines

There is a close correspondence between semi-coroutine systems [3] and the capabilities of co-expressions that have been described in the previous sections. In a semi-coroutine system, program control can be passed between some master process and a number of subordinate coroutine processes. The subordinate processes may not pass control among themselves, however.

Activation of a co-expression *interrupts* evaluation of the activating expression and *continues* evaluation of the co-expression. Suspension from a co-expression interrupts evaluation of the co-expression and continues evaluation of the activating expression. Thus co-expressions and generators represent primitives from which semi-coroutines can be constructed.

A co-expression can also activate other co-expressions, producing a general coroutine style of evaluation. The effect of one co-expression activating another is simply that evaluation is interrupted in the first, and continued in the second, thus providing at the expression level capabilities similar to the capabilities provided by coroutines at the procedure level in languages such as SL5 [10], and ACL [15].

3.4.1 Additional Language Features for Co-Expressions

The Icon programming provides some additional facilities intended to aid in the use of co-expressions in a general coroutine style. These are the keywords &main and &source, and the ability to pass results between co-expressions.

Program execution in Icon is initiated by an implicit call to the procedure main. The keyword &main is a co-expression for this call. Activation of &main from any co-expression returns control to the point of interruption in the evaluation of the call to main. A typical use of &main is as a means of exiting a cycle of co-expression evaluations.

For example, compare

```
global line, in, out

procedure main()

   in := create |{line := read() | @&main; @out}
   out := create |{@in & write(line)}
   @out

end
```

with

```
global line, in, out

procedure main()

   in := create |(line := read() & @out)
   out := create |(@in & write(line))
   @out

end
```

both of which copy standard input to standard output.

When the read in the first version fails, control is explicitly returned to &main, and processing terminates normally. In the second version, failure of the read causes failure of the activation of in in out. This in turn results in failure of the co-expression out, causing control to return to the expression that last activated out. If there was any input at all, this activating expression is in, but in cannot produce any results, so control returns to out, etc.!

&source is a co-expression for the activating expression of the currently active co-expression. Control may be explicitly transferred from a co-expression to its activating expression by activating &source.

A result can be supplied to the activation of a co-expression by

expr1 @ *expr2*

which supplies the result of *expr1* to the activation of the co-expression that is the result of *expr2*. (This result is ignored if the co-expression is being activated for the first time.)

3.4.2 Examples

To illustrate a number of coroutine facillities, Grune [9] posed the following problem:

Let A be a process that copies characters from some input to some output, replacing all occurrences of aa with b, and a similar process, B, that converts bb into c. Connect these processes in series by feeding the output of A into B.

The following program is a solution to this problem using co-expressions.

```
global A, B

procedure main()

   A := create compress("a", "b", create |reads(), B)
   B := create compress("b", "c", A, &main)

   while writes(@B)

end

procedure compress(c1, c2, in, out)
local ch

   repeat {
      ch := @in
      if ch == c1 then {
         ch := @in
         if ch == c1 then
            ch := c2
         else
            c1 @ out
         }
      ch @ out
      }

end
```

This solution is similar to a solution originally presented in Simula by Lynning [13], and translated into ACL by Marlin [15]. Like their solutions and those proposed by Grune, it assumes an infinite stream of input. Like these solutions, the one above creates two instances of the same procedure for the operation of both A and B. The Icon version is simplified slightly by the ability to transfer results between co-expressions, however.

The following example uses co-expressions to implement a prime number sieve. The technique is based upon a similar one used by McIlroy [14] to illustrate a use of coroutines.

The organization of the sieve is to supply an infinite stream of integers through a cascade of "filters", each of which checks to see if the integer is divisible by a specific known prime. Each filter activates the next filter in the cascade if the integer passes its test. If a filter finds an integer that is a multiple of its prime, the filter activates the source of integers and the cascade is restarted on the next integer. If the integer passes through the entire set of filters successfully, it is output as a prime and a new filter is added to the cascade to test subsequent integers against this prime.

The major uses of co-expressions are in source, which generates the integers and starts the cascade on each integer, the filters in the cascade, and in sink, which processes new primes. sink is actually treated as the last filter in the cascade. An additional co-expression is used to sequence through the filters in cascade using the selection operator.

```
global num, cascade, source, nextfilter

procedure main()

   source := create {
               every num := 2 to huge_number do
                  @@(nextfilter := create !cascade)
               @&main
               }
   cascade := []
   push(cascade, create sink())

   @source

end

procedure sink()
local prime

   repeat {
      write(prime := num)
      push(cascade, create filter(prime))
      @source
      }

end

procedure filter(prime)

   repeat
     if num % prime = 0 then @source
     else @@nextfilter

end
```

Note that each filter is invoked exactly once. From then on, control is simply passed between source and the various filters (including sink).

Actually, there is no need for any of the procedures other than main. This example can be written as

```
global num, cascade, source, nextfilter

procedure main()
local prime

   source := create {
                every num := 2 to huge_number do
                   @@(nextfilter := create !cascade)
                @&main
                }
   cascade := []
   push(cascade, create
      repeat {
         write(prime := num)
         push(cascade, create
              repeat
                 if num % prime = 0 then @source
                 else @@nextfilter)
         @source
         }

   @source

end
```

This version does not show the logical divisions of the algorithm as well as the previous version, however.

Note that this version works only because co-expressions maintain their own copies of local identifiers.

Acknowledgement

I am indebted to Cary Coutant, Ralph Griswold, and Dave Hanson for their discussions on co-expressions and control structures. Cary Coutant provided invaluable assistance with the implementation.

References

1. Coutant, Cary A., Ralph E. Griswold, and Stephen B. Wampler. *Reference Manual for the Icon Programming Language Version 3 (C Implementation for UNIX).* Technical Report TR 80-2, Department of Computer Science, The University of Arizona. May 1980.
2. Coutant, Cary A., Ralph E. Griswold, and Stephen B. Wampler. *Reference Manual for the Icon Programming Language; Version 4 (C Implementation for UNIX).* Technical Report TR 81-4, Department of Computer Science, The University of Arizona. July 1981.
3. Dahl, O.-J., and C. A. R. Hoare. "Coroutines", *Structured Programming*, Academic Press, London, 1972. pp. 184-193.
4. Griswold, Ralph E. "The Use of Character Sets and Character Mappings", *The Computer Journal*, Vol. 23, No. 2 (May 1980). pp. 107-114.
5. Griswold, Ralph E. *Expression Evaluation in Icon.* Technical Report TR 80-21, Department of Computer Science, The University of Arizona. August 1980.
6. Griswold, Ralph E., and David R. Hanson. "An Alternative to the Use of Patterns in String Processing", *ACM Transactions on Programming Languages and Systems*, Vol. 2, No. 2 (April, 1980). pp. 153-172.
7. Griswold, Ralph E., David R. Hanson, and John T. Korb. "The Icon Programming Language; An Overview", *SIGPLAN Notices*, Vol. 14, No. 4 (April 1979). pp. 18-31.
8. Griswold, Ralph E., David R. Hanson, and John T. Korb. "Generators in Icon", *ACM Transactions on Programming Languages and Systems*, Vol. 3, No. 2 (April 1981). pp. 144-161.
9. Grune, D. "A View of Coroutines", *SIGPLAN Notices*, Vol. 12, No. 7 (July 1977), pp. 75-81.
10. Hanson, David R. and Ralph E. Griswold. "The SL5 Procedure Mechanism", *Communications of the ACM*, Vol. 21, No. 5 (May 1978). pp. 392-400.
11. Hewitt, Carl, and Michael Patterson. "Comparative Schematology", *Record of Project MAC Conference on Concurrent Systems and Parallel Computation*, June 1970.
12. Korb, John T. *The Design and Implementation of a Goal-Directed Programming Language*. Technical Report TR 79-11, Department of Computer Science, The University of Arizona. June 1979.
13. Lynning, E. letter to the editor, *SIGPLAN Notices*, Vol. 13, No. 2 (February 1978), pp. 12-14.
14. McIlroy, M. Douglas. *Coroutines*. Technical Report, Bell Telephone Laboratories, May 1968.
15. Marlin, Christopher D. "Coroutines", *Lecture Notes in Computer Science*, Vol. 95. Springer-Verlag. New York, 1980.