Co-Expressions in Icon*

Stephen B. Wampler[†] *and Ralph E. Griswold*

TR 82-4

April 1982

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported by the National Science Foundation under Grants MCS79-03890 and MSC81-01916.

[†]Current address: College of Engineering, Northern Arizona University, Flagstaff, Arizona 86011.

Co-Expressions in Icon

1. Introduction

Icon is a high-level programming language that features facilities for string and list processing. In addition to these facilities, it has expressions, called *generators*, that are capable of producing sequences of results. A goal-directed evaluation mechanism automatically produces the results of generators in an attempt to produce 'successful' computations. Generators and goal-directed evaluation make it possible to formulate concise natural solutions for many programming problems.

Generators in Icon are limited in their use by the syntax of the language. This has the advantage of providing straightforward means of controlling generators, as well as permitting an efficient implementation.

The sequence of results that can be produced by a generator is limited to a single lexical site in the program, however. Furthermore, every evaluation of a generator produces results from the result sequence for that generator starting at the *first* result.

To overcome these problems, *co-expressions* were introduced in Version 5 of Icon.

The following sections describe co-expressions and give examples of their use. The reader is expected to be familiar with the Icon programming language. Much of the material here originally appeared in References 1 and 2; this report is intended as a supplement to the reference manual for Version 5 of Icon [3].

2. The Features of Co-Expressions

2.1 Co-Expression Environments and Resumption

The expression

```
create expr
```

creates a *co-expression environment* for *expr*. A co-expression environment (subsequently referred to simply as a co-expression), is a data object that contains the information necessary to evaluate an expression: a reference to the expression itself, a 'program counter' indicating where evaluation of the expression is to resume, and copies of the local identifiers referenced in the expression with initial values as they are when the co-expression is created.

The expression within a co-expression can be explicitly resumed whenever a result from the sequence of results for the expression is needed. The resumption operation is

```
@x
```

where x is a co-expression. For example

```
texp := create !tlist
```

creates a co-expression for the generator !tlist and successive resumptions of texp produce the results from this generator. Resumption of a co-expression fails once all the results from its expression have been generated. For example

```
while write(@texp)
```

writes all the values in tlist, but

```
while write(@texp) do
   @texp
```

writes only the odd-numbered values in tlist.

Sometimes is is useful to be able to transmit a value to a co-expression when it is resumed The operation

```
expr @ x
```

resumes the co-expression x and supplies the value produced by *expr* to it (This result is ignored if the co-expression is being resumed for its first result) The transmission of a value to a resumed co-expression is most frequently useful in producer consumer contexts An example is given in Section 3 3

The operation

```
*x
```

produces a count of the number of results that have been produced by resuming the co-expression x The operator chosen reflects the similarity of this operation to the computation of the size of a string or a list

2.2 Refreshing Co-Expressions

The *refresh* operation

```
^x
```

produces a copy of the co-expression x restored to its state when it was created Thus the refresh operation provides a means of repeating the sequence of results of a co-expression For example,

```
x  = create find("ab", "abracadabra")
write("The first position is ", @x)
write("The second position is ", @x)
x .= ^x
write("The first position still is ", @x)
```

writes

```
The first position is 1
The second position is 8
The first position still is 1
```

Global side effects, of course are not reversed by the refresh operation

2.3 Built-In Co-Expressions

There are two built in co expressions to aid in the use of co-expressions in a general coroutine style These co expressions are the values of the keywords &main and &source

Program execution in Icon is initiated by an implicit call to the procedure main The keyword &main is a co-expression for this call Resumption of &main from any co-expression returns control to the point of interruption in the evaluation of the call to main

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

With &main and &source it is possible for any co-expression to transfer control to *any* other co-expression providing a general coroutine facility Examples are given in the following sections

3. Examples of Co-Expression Usage

3.1 Parallel Evaluation

As mentioned earlier goal directed evaluation provides a cross-product form of analysis that is suitable for many combinatorial applications Parallel, or 'dot-product' evaluation is not possible without co-expressions

Consider the problem of determining, without co-expressions whether two expressions, $expr_1$ and $expr_2$ produce the same sequences of results Since the results from two separate expressions cannot be produced in an arbitrary manner, some other method is needed to obtain corresponding values for comparison One possibility is to generate all the values for one expression first and 'capture' them by putting them in a (physical) list

```
seq := []
every put(seq, expr1)
```

The values for $expr_2$ can now be generated and compared with those in seq There is no longer a problem with parallel evaluation, since the elements of seq can be accessed by position

This approach has several disadvantages, the most serious of which is that a list of all the results for one expression must be produced before a single result is produced for the other This process may be time and space consuming and must be carried to completion, even if the first results in the two sequences are different

With co-expressions, the results of two expressions can be generated and compared in parallel A procedure to do this is

```
procedure compseq(x1, x2)
   local r1, r2
   while r1 := @x1 do {
      (r2 := @x2) | fail
      (r1 === r2) | fail
      }
   if @x2 then fail else return
end
```

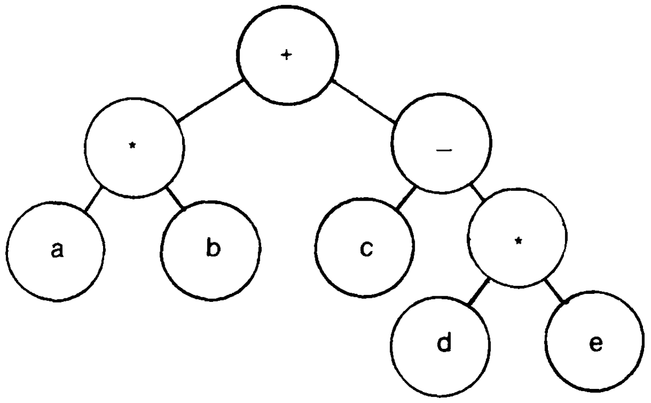
Since the two sequences may have different lengths, one, x1, is chosen to control the loop There are two situations in which the sequences may fail to compare within the loop if the sequence for x2 terminates first, or if corresponding values are different If resumption of x2 fails, assignment to r2 fails, and the second expression in the alternation causes the procedure to fail The operation r1 === r2 compares values of arbitrary type and fails if they are not identical Again, the procedure fails if the comparison fails Finally, if resumption of x1 fails terminating the loop a check must be made to determine if x2 has additional values, if so, the procedure fails

The structural asymmetry in the procedure is imposed by the need to check the lengths of the two sequences of results as well as their values (there is no way, *a priori*, to determine the length of a sequence of results) The same problem occurs in comparing a physical list of values produced by one expression with those generated by another, as is evident if the details of the coding are carried out

3.2 The 'Same-Fringe' Problem

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 [4] that require access to the leaves of a tree

Suppose that a tree is represented by a list whose first element is a value associated with that node and whose subsequent elements are subtrees For example, the tree



is represented by the list

```
["+", ["*", ["a"], ["b"]], ["—", ["c"], ["*", ["d"], ["e"]]]]
```

A procedure to generate the leaves of such a tree is

```
procedure leaves(tree)
   if *tree = 1 then return tree[1]
   else suspend leaves(tree[2 to *tree])
end
```

This procedure can be used in a solution to the same-fringe problem to walk two trees in parallel to determine if their leaf nodes have the same values in the same order:

```
if compseq(create leaves(tree2), create leaves(tree2)) then
   write("same fringe")
else
   write("different fringes")
```

3.3 Grune's Problem

As indicated above, co-expressions have coroutine capabilities [5-7].

The following problem was originally posed by Grune [8] to illustrate a number of coroutine facilities.

"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."

Using co-expressions, this problem can be solved as follows.

```
global A, B

procedure main()
   A := create compress("a", "b", create |reads(), B)
   B := create compress("b", "c", A, &main)
   repeat 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 [9] and translated into ACL by Marlin [10]. Like their solutions and those proposed by Grune, it assumes an infinite stream of input, although it is not hard to modify the solution above for a finite input stream. Like their 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 explicitly between co-expressions.

3.4 The Sieve of Eratosthenes

The following example uses co-expressions to implement the Sieve of Eratosthenes. The technique is based upon a similar one used to illustrate a use of coroutines [11] and filtered variables [12].

The sieve supplies 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.

```
global number, cascade, source, nextfilter

procedure main()
   cascade := []
   source := create {          # root of sieve
      number := 1
      repeat {
         number +:= 1
         nextfilter := create !cascade               # sequence of filters
         @@nextfilter                # get first filter and activate it
         }
      }
   push(cascade, create sink())      # sink starts as the only filter
   @source                     # start the sieve
end

procedure sink()
   local prime
   repeat {
      write(prime := number)
      push(cascade, create filter(prime))            # add filter to cascade
      @source                  # start processing next number
      }
end
```

```
procedure filter(prime)
   repeat {
      if number % prime = 0 then @source        # try next number
      else @@nextfilter       # get next filter and activate it.
      }
end
```

The co-expression source generates the integers and starts the cascade on each integer. Each filter in the cascade is a co-expression that tests the potential prime against a specific known prime. The co-expression sink processes new primes and is always the last filter in the cascade. An additional co-expression is used to sequence through the filters in cascade. 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 number, cascade, source, nextfilter

procedure main()
   local prime
   cascade := []
   source := create {
      number := 1
      repeat {
         number +:= 1
         @@(nextfilter := create !cascade)
         }
      @&main
      }
   push(cascade, create
      repeat {
         write(prime := number)
         push(cascade, create repeat
            if number % prime = 0 then @source
               else @@nextfilter
            )
         @source
         }
      )
   @source
end
```

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

3.5 Modeling Generative Control Structures

Since co-expressions allow control over the generation of results, they can be used to model generative control structures and to gain insight into their relationship with traditional control structures.

For example, alternation

$expr_1 \mid expr_2$

can be modeled by a procedure such as

```
procedure Alt(x1, x2)
   local r
   while r := @x1 do suspend r
   while r := @x2 do suspend r
end
```

which is invoked as

```
Alt(create expr1, create expr2)
```

This model clearly demonstrates the relationship between the sequences of results for *expr*$_1$ and *expr*$_2$ and the sequence of results for

expr$_1$ | *expr*$_2$

and avoids complicated explanations of alternation in terms of control backtracking [13]

Similarly, the relationship between

every *expr*$_1$ do *expr*$_2$

and

while *expr*$_1$ do *expr*$_2$

is illuminated by the model

```
procedure Every(x1, x2)
   while @x1 do @^x2
end
```

In fact, all the generative control structures in Icon can be modeled using co-expressions and traditional control structures

4. The Status of Co-Expressions in Version 5 of Icon

Co-expressions are included in Version 5 of Icon as an unsupported feature Co-expressions are not supported since the nature of their implementation in Version 5 limits their usefulness and allows possible malfunction of programs in which they are used In particular, only a few co-expressions can be in existence at any one time in a program More importantly, stack overflow in co-expressions is not checked, such overflow may cause of variety of program malfunctions

Nonetheless co-expressions can be used safely in many situations and offer the opportunity for interesting programming methods The following additional information about co-expressions may be useful

- The infix operator @, which supplies a result to the activation of a co-expression, associates to the left and has the same precedence as the operator \ (see Reference 3) The augmented assignment operator @:= is available
- If x is a co-expression, copy(x) simply returns x, not a physically distinct copy of it
- There are two error messages associated with the use of co-expressions

118	co-expression expected
305	insufficient storage for co-expressions

There are two environment variables that may be assigned values to control the storage utilization of co-expressions [14, 15]

NSTACKS	Set the number of stacks initially available for co-expressions Normally, two stacks are available More are automatically allocated if needed
STKSIZE	Set the size of each co-expression stack (in words) Normally 1000 words per stack are available

Acknowledgements

Tim Korb [13] originally suggested the idea of 'capturing' expressions, on which co-expressions are based. We are indebted to Cary Coutant and Dave Hanson for their discussions and suggestions on co-expressions. Cary Coutant provided invaluable assistance with the implementation.

References

1 Wampler, Stephen B. *New Control Structures in Icon*. Technical Report TR 81-1a, Department of Computer Science, The University of Arizona. July 1981.

2 Wampler, Stephen B. *Control Mechanisms for Generators in Icon*. Technical Report TR 81-18, Department of Computer Science, The University of Arizona. December 1981.

3 Coutant, Cary A., Ralph E. Griswold, and Stephen B. Wampler. *Reference Manual for the Icon Programming Language, Version 5 (C Implementation for UNIX)*, Technical Report TR 81-4a, Department of Computer Science, The University of Arizona. December 1981.

4 Hewitt, Carl and Michael Patterson. "Comparative Schematology", *Record of Project MAC Conference on Concurrent Systems and Parallel Computation*. June 1980.

5 Conway, Melvin. "Design of a Separable Transition-Diagram Compiler", *Communications of the ACM*. Vol. 6, No. 7 (July 1963) pp. 396-408.

6 Dahl, Ole-Johan and C. A. R. Hoare. A "Coroutines", *Structured Programming*, Academic Press. 1972. pp. 184-193.

7 Ichbiah, J. D. and S. P. Morse. "General Concepts of the Simula 67 Programming Language", *Annual Review in Automatic Programming*, Vol. 7, No. 1 (1972) pp. 65-93.

8 Grune, Dick. "A View of Coroutines", *SIGPLAN Notices*, Vol. 12, No. 7 (July 1977) pp. 75-81.

9 Lynning, F. (1978). Letter to the editor, *SIGPLAN Notices*, Vol. 13, No. 2 (February 1978) pp. 12-14.

10 Marlin, C. D. "Coroutines", *Lecture Notes in Computer Science*, Vol. 95, Springer-Verlag. 1980.

11 McIlroy, M. D. *Coroutines*, Technical Report, Bell Telephone Laboratories, Murray Hill, New Jersey. 1968.

12 Hanson, David R. "Filters in SL5", *The Computer Journal*, Volume 21, No. 2 (May 1978) pp. 134-143.

13 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.

14 Coutant, Cary A. and Stephen B. Wampler. *ICONC(1)*. Local manual page for *UNIX Programmer's Manual*. Department of Computer Science, The University of Arizona. December 1981.

15 Coutant, Cary A. and Stephen B. Wampler. *ICONX(1)*. Local manual page for *UNIX Programmer's Manual*. Department of Computer Science, The University of Arizona. December 1981.