Co-expressions in Icon*

S. B. Wampler† and R. E. Griswold
Department of Computer Science, The University of Arizona, Tucson, Arizona 85721, USA

The Icon programming language has *generators* that are capable of producing sequences of results and a goal-directed evaluation mechanism that allows concise formulation of many kinds of computations. The evaluation of generators is restricted to their lexical site in a program, however. This paper describes *co-expressions*, an extension to Icon that allows generators to be used at any time or place in a program. Examples of co-expression usage are given and the relationship of co-expressions to coroutines is discussed.

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. The evaluation of a generator is limited, however, to its lexical site in a program. This paper describes a mechanism that frees generators from their lexical sites so that their results may be used as needed, where needed.

The following section describes the basic aspects of Icon, including enough of its features to understand the examples given in subsequent sections. Section 2 describes *co-expressions*, which are the expression level analog of coroutines and which allow the results of a generator to be used anywhere in a program. Examples of co-expression usage are given in Section 3, followed by conclusions and discussion.

1. FEATURES OF ICON

Icon is a fully developed programming language with a wide range of features. A few of its string and list processing features, together with control structures and the essentials of expression evaluation, are sufficient for the central issues of this paper. The interested reader may wish to refer to Refs 1–3 for more information.

1.1 String and list processing

A string is a sequence of characters. Strings are data objects in Icon, rather than being arrays of characters. In this respect, Icon is similar to SNOBOL4.[4] Strings can be represented literally, as in

text := "this theory is the third attempt"

which assigns a string of 32 characters to *text*. The size of a string *s* is produced by the operation **s*. For example

write(**text*)

writes 32.

* This work was supported by the National Science Foundation under Grants MCS79–03890 and MCS81–01916.
† Present address: College of Engineering, Northern Arizona University, Flagstaff, Arizona 86011, USA.

Strings can be computed in a variety of ways. Concatenation, given by the operation

*s*1 || *s*2

produces a string consisting of *s*1 followed by *s*2. The *empty string*, which contains no characters, is the identity with respect to concatenation. The empty string is represented literally by "".

There are a number of operations that analyze strings. A typical one is

find(*s*1, *s*2)

which produces the position at which *s*1 occurs as a substring of *s*2. For the value of *text* given above.

i := find("is", *text*)

assign the value 3 to *i*.

Lists are sequences of objects of any type and are created by enclosing the list of objects in brackets. For example

tlist := [*text*, **text*]

assigns a list of two values to *tlist*. The first value in *tlist* is a string, and the second is an integer. The elements in a list are referenced by position, using subscripting expressions. For example

write(*tlist*[2])

writes the second element in *tlist*, and

tlist[2] := *tlist*[2] − 1

decrements the second element in *tlist*. Icon has *augmented assignment* operations that combine other operations with assignment. For example

tlist[2] +:= 1

increments the value of the second element in *tlist*. Similarly

s ||:= ":"

appends a colon to value of *s*.

The size of a list is produced by the same operation that produces the size of a string; the value of **tlist* is 2.

There are stack and queue access functions for lists that allow lists to grow and shrink automatically. For example,

push(*tlist*, "Example-1")

CCC–0010–4620/83/0026–0072 $03.50

pushes the string "Example-1" on the left end of *tlist*. The value of **tlist* is now 3 and the value of *tlist*[3] is 32. Conversely,

pop(*tlist*)

removes the leftmost value from *tlist*, restoring it to its former size. The function put(*tlist*, *x*) adds *x* to the right end of *tlist*, whereas the converse queue access function get(*tlist*) is synonymous with pop(*tlist*).

1.2 Generators

In Icon an operation may succeed and produce a result, or it may fail. (Icon is similar to SNOBOL4 in this respect.) The function find(*s*1, *s*2) described above provides a natural example of this possibility, since *s*1 may not occur as a substring of *s*2. For example, for the value of *text* given above

i := find("two", *text*)

fails, since "two" is not a substring of the value of *text*. When an operation fails, this failure is 'inherited' by surrounding expressions, which are not evaluated. For this example, the assignment is not performed and the value of *i* is not changed. Stated another way, the assignment to *i* is contingent on the success of find(*s*1, *s*2).

Another example of a function that may fail is read(), which reads a line of input, but fails when the end of the input file is reached.

The function find(*s*1, *s*2) also provides a natural example of a situation in which there can be more than one result. For example, in

find("th", *text*)

there are four places where "th" occurs as a substring of the value of *text*: 1, 6, 16, and 20, as illustrated by the arrows below:

```
this theory is the third attempt
↑    ↑         ↑   ↑
```

Icon takes advantage of such possibilities by allowing expressions to produce more than one result. Such expressions are called *generators*. In simple contexts, such as

i := find("th", *text*)

a generator only produces its first value. In this case, the value assigned to *i* is 1, since find(*s*1, *s*2) produces the positions of *s*1 in *s*2 from left to right. In more complicated situations, such as the comparison operation

find("th", *text*) = 16

a generator produces its results until the enclosing expression succeeds or until there are no more results. In this case, the first two results produced by find, 1 and 6, do not satisfy the comparison. The third result, 16, does satisfying the comparison, so the entire expression succeeds. This is an example of *goal-directed evaluation*, which is implicit in expression evaluation in Icon.

Another generator is

i **to** *j*

which generates the integers in sequence from *i* to *j*. For example

tlist[1 **to** **tlist*]

1.3 Control structures

Icon has several traditional control structures. An example is

while $expr_1$ **do** $expr_2$

The control expression, $expr_1$, is treated somewhat differently in Icon than in most programming languages. Rather than depending on the production of a Boolean value *true* or *false*, control is determined by the success or failure of $expr_1$. For example

```
while line := read() do
   write(line)
```

copies input to output. The loop is terminated when read() fails. The **do** clause can be omitted. An equivalent expression is

while write(read())

Generators provide the motivation for a number of more novel control structures. One such control structure is *alternation*,

$expr_1$ | $expr_2$

This control structure produces the sequence of results for $expr_1$ followed by the sequence of results for $expr_2$. For example

2 | 3 | 5

produces the sequence 2, 3, 5. Like find(*s*1, *s*2), alternation produces more than one result only when the surrounding context requires it. Similarly

find(*s*1, *s*2) = (10 | 20 | 30 | 40)

succeeds if *s*1 occurs as a substring of *s*2 at position 10, 20, 30, or 40.

A somewhat more unusual control structure is *repeated alternation*,

|*expr*

which produces the sequence of results for *expr* repeatedly, stopping only when *expr* fails. For example, the sequence of results for

|read()

is the lines of input to a program.

Since sequences of results are natural in Icon, iteration over sequences is frequently useful, and is performed by

every $expr_1$ **do** $expr_2$

which evaluates $expr_2$ for every result produced by $expr_1$. For example

```
every i := find(s1, s2) do
   write(i)
```

writes the positions at which *s*1 occurs as a substring of *s*2. The **do** clause can be omitted. An equivalent expression is

every write(find(*s*1, *s*2))

Since strings and lists are sequences of characters and arbitrary values, respectively, it is useful to be able to sequence through their elements concisely. The expression !*x* is a generator that produces the elements of *x*, which may be a string or list, in order from left to right. For example

```
every write(!tlist)
```

writes all the values in *tlist*, whereas

```
every write(!text)
```

writes all the characters of the string *text* on separate lines.

1.4 Procedures

Procedures in Icon are similar to those in many traditional programming languages, except that they can fail or produce a sequence of results, as well as produce a single result. Return of a single result is indicated by

```
return expr
```

while failure is indicated by **fail.** For example

```
procedure cmax(i, j)
  if i > j then return j else fail
end
```

returns *j* if *i* is greater than *j*, but fails otherwise. A sequence of results can be produced by using

```
suspend expr
```

which returns the value of *expr* but leaves the procedure environment intact so that it can be resumed to produce another result. For example

```
procedure To(i, j)
  while i < = j do {
    suspend i
    i + := 1
    }
end
```

is a procedural version of

```
i to j
```

1.5 The order of evaluation in expressions

In Icon, expressions are evaluated from left to right (in the absence of control structures) and generators are resumed in a last-in, first-out fashion. Thus in an expression such as

```
find(s1, s2) > find(s3, s4)
```

the expression find(*s*1, *s*2) produces its first result (if any) and then find(*s*3, *s*4) produces its first result (if any). If the comparison then fails, find(*s*3, *s*4) is resumed to produce its next result. Only when find(*s*3, *s*4) has produced all its results is find(*s*1, *s*2) resumed to produce its second result. When find(*s*1, *s*2) produces its second result, find(*s*3, *s*4) is *evaluated* again and produces its first result again. This 'cross-product' evaluation mechanism makes Icon well suited to combinatorial applications.[2]

2. CO-EXPRESSIONS

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. For example, !*tlist* produces the values from *tlist* in sequence, but there is no straightforward way to use this generator to write, for example, only every other value in *tlist*, since there is no way to resume !*tlist* at several sites in a program without reproducing its sequence of results from the beginning.

To overcome these problems, the concept of *co-expression* has been introduced.

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 expressions 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 it 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-EXPRESSIONS 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
```

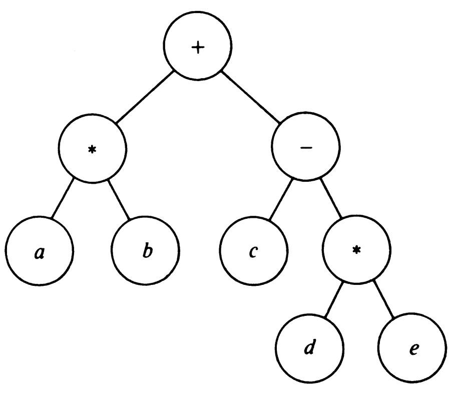
Since the two sequences may have different lengths, one, *x*1, is chosen to control the loop. There are two situations in which the sequences may fail to compare within the loop—if the sequence for *x*2 terminates first, or if corresponding values are different. If resumption of *x*2 fails, assignment to *r*2 fails, and the second expression in the alternation causes the procedure to fail. The operation *r*1 === *r*2 compares arbitrary objects and fails if they are not identical. Again, the procedure fails if the comparison fails. Finally, if resumption of *x*1 fails, terminating the loop, a check must be made to determine if *x*2 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[5] 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.[6–8]

The following problem was originally posed by Grune[9] 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
```

The control structure

```
repeat expr
```

evaluates *expr* in an infinite loop. The operation *s*1 == *s*2 succeeds if *s*1 and *s*2 are the same strings. The function reads() reads a single character and writes(*s*) writes *s* in stream mode without line terminators.

This solution is similar to a solution originally presented in Simula[10] and translated into ACL by Marlin.[11] 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[12] and filtered variables.[13]

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 operation $n \% m$ produces the remainder of n divided by m). 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 Modelling 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 modelled 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** $expr_1$, **create** $expr_2$)

This model clearly demonstrates the relationship between the sequences of results for $expr_1$ and $expr_2$ and the sequence of results for

$expr_1 \mid expr_2$

and avoids complicated explanations of alternation in terms of control backtracking.[2]

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 modelled using co-expressions and traditional control structures.

4. DISCUSSION AND CONCLUSIONS

Co-expressions have been implemented in Version 5 of Icon. In addition to the kinds of uses illustrated by examples in this paper, co-expressions have been used to experiment with new kinds of control structures without the need for modifying the implementation of Icon itself.

Co-expressions represent a significant step in increasing the functionality of generators, since they free the evaluation of generators from their lexical sites and permit access to the results of a generator when and where needed. In this sense, co-expressions represent the instantiation of sequences of results into the programming language as data objects that can be manipulated in much the same way as other data objects are manipulated.

Co-expressions also provide insight into the operation of coroutine facilities. Most languages that incorporate coroutines do so by associating the coroutine mechanism with procedures. In actuality, it is the expression instance containing the *invocation* of a procedure that functions as the coroutine. Co-expressions make this clear by associating the coroutine mechanism with expression instances, rather than with procedures.

The concept of expression instances is also a useful descriptive tool leading to a better understanding of the operation of goal-directed evaluation and co-expressions. Considering expression evaluation as occurring within expression instances and then examining goal-directed evaluation and co-expressions as operations upon expression instances simplifies the development and analysis of implementation techniques for these language features.

Acknowledgements

Tim Korb[14] 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. C. A. Coutant, R. E. Griswold and S. B. Wampler, *Reference Manual for the Icon Programming Language; Version 5 (C Implementaton for UNIX)*, Technical Report TR 81-4a, Department of Computer Science, The University of Arizona (1981).
2. R. E. Griswold, D. R. Hanson and J. T. Korb, Generators in Icon. *TOPLAS* **3** (2), 144–161 (1981).
3. R. E. Griswold, The evaluation of expressions in Icon. *TOPLAS* in press.
4. R. E. Griswold, J. F. Poage and I. P. Polonsky, *The SNOBOL4 Programming Language*, 2nd Edn., Prentice-Hall, Englewood Cliffs, New Jersey (1971).
5. C. Hewitt and M. Patterson, Comparative schematology, in *Record of Project MAC Conference on Concurrent Systems and Parallel Computation* (1970).
6. M. Conway, Design of a separable transition-diagram compiler. *Communications of the ACM* **6**, 396–408 (1963).
7. O.-J. Dahl and C. A. R. Hoare, Coroutines, in *Structured Programming*, Academic Press, New York (1972).
8. J. D. Ichbiah and S. P. Morse, General concepts of the Simula 67 programming language. *Annual Review in Automatic Programming* **7** (1), 65–93 (1972).
9. D. Grune, A view of coroutines. *SIGPLAN Notices* **12** (7), 75–81 (1977).
10. E. Lynning, Letter to the editor. *SIGPLAN Notices* **13** (2), 12–14 (1978).
11. C. D. Marlin, Coroutines. *Lecture Notes in Computer Science* **95**, Springer-Verlag, Berlin (1980).
12. M. D. McIlroy, *Coroutines*, Technical Report, Bell Laboratories, Murray Hill, New Jersey (1968).
13. D. R. Hanson, Filters in SL5. *The Computer Journal* **21** 134–148 (1978).
14. J. T. Korb, *The Design and Implementation of a Goal-Directed Programming Language*, Doctoral Dissertation, Department of Computer Science, The University of Arizona (1979).

Received April 1982