Programmer-Defined Control Operations In Icon*

Ralph E. Griswold and Michael Novak

TR 82-8a

August 3, 1982. Revised November 22, 1983

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported by the National Science Foundation under Grant MCS881-01916.

Programmer-Defined Control Operations In Icon

1. Introduction

Broadly speaking, a control operation is any program mechanism that directly affects the flow of control in a program. Fisher (1970) identifies six types of control operations: sequential processing, parallel processing, testing, monitoring, synchronization, and relative continuity. The scope of this report, and hence of the discussion that follows, is more limited and only considers operations that affect the sequencing of expression evaluation. The term *control operation* is used here to cover mechanisms in this domain, implicit or explicit, while the term *control structure* is reserved for such operations that are distinguished syntactically. For example,

if $expr_1$ then $expr_2$ else $expr_3$

is a control structure. On the other hand, argument evaluation in a typical expression-oriented programming language is an implicit control operation. A control operation that is broadly applicable to expression evaluation, as is argument evaluation, is referred to as a *control regime*.

While most programming languages have facilities that allow programmers to define procedures that augment the repertoire of built-in functions, comparatively few programming languages have facilities for defining control operations. There are exceptions, including Madcap 6 (Morris and Wells, 1972) and the extensible language EL1 (Wegbreit, 1970). Some work (Leavenworth, 1969 and Fisher, 1970) has been focused directly on the definition of control operations.

The comparative lack of facilities for programmer-defined control operations is not surprising. While the number of programming language constructs that can be classified as control operations is large, most of them are variants on a few control themes. Furthermore, the number of different control operations that can be comfortably and consistently accommodated in most programming languages is relatively small. The design of a set of control operations usually is more a matter of selection and refinement than of invention. The motivation for programmer-defined control operations therefore is relatively small in the context of conventional programming languages.

The Icon programming language is another matter. Expression evaluation in Icon, in which an expression can generate a sequence of results, poses many interesting and novel problems in control operation design. In the evolution of Icon through several versions, more significant changes were made in control operation design than in any other area of the language (Griswold, 1982).

The manipulation of sequences of results adds a dimension to control operations that is lacking in most programming languages. The question is not what traditional control operations to select, but which of many possibilities to explore. In Icon, the problem of control operation design becomes very similar to the design of the built-in operation repertoire in most programming languages. As with the selection of a set of built-in operations, the problem remains of providing facilities for the definition of others.

This report describes a simple programmer-defined control operation mechanism (PDCO) and gives a number of examples of its use.

2. Control Aspects of Icon

In order to understand all the programming examples in this report, the reader should be familiar with Icon (Griswold and Griswold, 1983). The following sections review the most relevant material, concentrating on concepts related to control operations.

2.1 Expression Evaluation

2.1.1 Result Sequences

In most programming languages, the evaluation of an expression produces exactly one result. Thus it is typical for a comparison expression, such as

```
i > j
```

to produce a Boolean value, *true* or *false*, depending on whether or not the specified relation holds.

In Icon, an expression is capable of producing a sequence of zero or more results. Ordinary computational operations, such as

```
i + j
```

produce a single result as they do in more conventional languages. On the other hand, a comparison expression, such as

```
i > j
```

produces a result (the value of j) if the specified relation holds, but it does not produce a result if the relation does not hold. That is, a comparison expression has a sequence of zero or one results, depending on the values of its arguments. A sequence of zero results corresponds to *failure*, while a sequence of more than zero results corresponds to *success*.

SNOBOL4 (Griswold, Poage and Polonsky, 1971) resembles Icon in this respect. While SNOBOL4 terminology refers to success and failure as 'signals', expression evaluation in SNOBOL4 can be described equally well in terms of sequences of zero or one results.

The motivation for using sequence terminology rather than signals comes from expressions that may produce more than one result. An example is

```
i to j
```

which is capable of producing the integers in sequence from i to j, inclusive. Thus

```
1 to 5
```

is capable of producing the sequence 1, 2, 3, 4, 5.

The sequence of results that an expression is capable of producing is called its *result sequence* (Wampler 1981) and is denoted by the sequence of results enclosed in braces. For example, the result sequence for

```
1 to 5
```

is {1, 2, 3, 4, 5}.

Expressions that are capable of producing more than one result are called *generators* (Griswold, Hanson and Korb, 1981). Such expressions occur in pattern matching in SNOBOL4. For example, the pattern ARB is capable of matching strings of length zero, one, two, and so on. There is no way in SNOBOL4, however, to access these results directly; they are implicit in the pattern-matching process. The used of the term generator serves to emphasize the capacity of some expressions to produce more than one result. There is, however, no actual distinction in Icon between generators and other expressions. All expressions have result sequences, although some result sequences may be of length zero or one.

Icon has many generators. One example is find(s1,s2), whose result sequence is the positions in s2, from left to right, at which s1 occurs as a substring. For example,

```
find("on","one motion is optional")
```

has the result sequence {1, 9, 19}.

A similar generator is upto(c, s), whose result sequence is the positions in s, from left to right, at which any character in c occurs. For example,

```
upto("on", "one motion is optional")
```

has the result sequence {1, 2, 6, 9, 10, 15, 19, 20}

Another generator is !x, whose result sequence is the elements of x, from left to right. For example, if a is a list

```
a := ["a", "an", "the"]
```

then the result sequence for !a is {"a", "an", "the"}.

The alternation control structure,

$expr_1$ | $expr_2$

has a result sequence consisting of the concatenation of the result sequences for $expr_1$ and $expr_2$. For example, the result sequence for

```
(1 to 4) | (6 to 10)
```

is {1, 2, 3, 4, 6, 7, 8, 9, 10}.

Whether an expression produces all the results in its result sequence depends on context. Once a generator has produced a result, it must be *resumed* to produce another result. A generator that has produced all the results in its result sequence is said to be *depleted*. The resumption of a depleted generator does not produce a result.

There are two contexts in which expressions are resumed: *iteration* and *goal-directed evaluation*.

2.1.2 Iteration

The control structure

every $expr_1$ do $expr_2$

resumes $expr_1$ repeatedly, producing all the results in the result sequence for $expr_1$. For each result produced by $expr_1$, $expr_2$ is evaluated (not resumed). For example

```
every i := (1 to 5) do write(i ^ 2)
```

writes

```
1
4
9
16
25
```

Note that the assignment is performed for each result that is produced by its right argument as if the following expressions had been evaluated

```
i := 1
i := 2
i := 3
i := 4
i := 5
```

The *do* clause in the iteration control structure is optional, allowing the expression above to be recast as

```
every write((1 to 5) ^ 2)
```

2.1.3 Goal-Directed Evaluation

While iteration over result sequences is performed explicitly by a control structure, goal-directed evaluation is implicit.

In the evaluation of an expression, any generators in it are resumed until the expression produces a result (succeeds) or until all the generators in the expression are depleted

Consider the expression

```
(x | y) = 10
```

The left argument of the comparison operation has the result sequence {x, y} The first result produced by the left argument expression is x The situation at this point is equivalent to the evaluation of

```
x = 10
```

If the value of x is 10, the comparison succeeds and the evaluation of

```
(x | y) = 10
```

is complete The left argument expression is not resumed and the result y is not produced

If the value of x is not 10, however, the comparison fails and the left argument expression is resumed producing y The situation at this point is equivalent to the evaluation of

```
y = 10
```

If the value of y is 10 the comparison succeeds If the value of y is not 10, the left argument expression is resumed again Since it is depleted no result is produced and the entire expression fails

Thus

```
(x | y) = 10
```

succeeds if either x or y has the value 10 but it fails otherwise

2.1.4 Compound Generators

If an expression contains several generators, the results it produces are determined by the order in which the generators are resumed This order is fundamental to expression evaluation in Icon

Operators in expressions such as

```
i + j
```

and functions in expressions such as

```
find(s1, s2)
```

differ only in syntax To simplify the discussion that follows, the term *function* is used for both

In the absence of control structures, the arguments of a function call are evaluated from left to right If evaluation of any argument expression fails, the last argument expression to be evaluated is resumed to produce another result If it is depleted and does not produce a result, the next previous argument expression is resumed, and so on When a resumed argument expression produces a result, the remaining argument expressions to the right are evaluated again If the first argument expression fails to produce a result, the function is not invoked and the entire expression fails If all the arguments produce a result, the function is invoked with those argument values If the function fails for these argument values, the last argument expression is resumed for another result and evaluation of the argument expressions proceeds as described above Thus generators are resumed in a last-in, first-out manner

In the context of iteration, all possible combinations of results from the result sequences of all expressions are produced In goal-directed evaluation, results are produced until the expression succeeds or until all generators are depleted For example, in

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

the order of function invocation is

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

Similarly, the result sequence for

```
(1 | 3) + (2 to 5)
```

is {3, 4, 5, 6, 5, 6, 7, 8}

To summarize, left-to-right evaluation, coupled with last-in, first-out resumption applies to the evaluation of all operations and functions in Icon and is the only built-in argument evaluation regime

2.1.5 Other Generative Control Structures

The control structure

$expr_1 \setminus expr_2$

limits the result sequence for $expr_1$ to at most $expr_2$ results For example, the result sequence for

```
find(s1, s2) \ 10
```

is at most the first 10 positions at which **s1** occurs as a substring of **s2** $expr_2$ is evaluated before $expr_1$ (contrary to the normal left-to-right mode of evaluation in Icon) and $expr_2$ can be a generator For example the result sequence for

```
(1 to 3) \ (1 to 3)
```

is {1, 1, 2, 1, 2, 3}

Repeated alternation,

$|expr$

produces the result sequence for *expr* repeatedly For example, the result sequence for

```
|(1 to 3)
```

is {1, 2, 3, 1, 2, 3, 1, 2, 3, ... } This result sequence is infinite, but it can be limited as described above To prevent the possibility of an internal resumption loop that could not be limited at the source-language level repeated alternation has the additional property that if *expr* ever produces an empty result sequence, repeated alternation terminates at that point For example, the result sequence for

```
|read()
```

is the sequence of lines from input This sequence terminates when **read()** fails at the end of the input file

2.2 Procedures

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

return *expr*

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

```
procedure fcount(s1, s2)
   count := 0
   every find(s1, s2) do count +:= 1
   if count > 0 then return count else fail
end
```

produces the number of times **s1** occurs as a substring in **s2** unless the count is zero, in which case it fails

Flowing off the end of a procedure body without an explicit return is equivalent to **fail**

A sequence of results can be produced by using

```
suspend expr
```

which returns the result of evaluating *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
```

The *suspend* control structure iterates over the result sequence for its argument in the fashion of *every-do*, suspending with successive results. Consider the procedure

```
procedure octcode()
   suspend (0 to 1) || (0 to 7) || (0 to 7)
end
```

The expression octcode() has the result sequence {000, 001, ..., 007, 010, ..., 077, 100, ..., 177}.

Sometimes it is useful to *encapsulate* a generator in a procedure so that its result sequence can be obtained anywhere the procedure is called. Given an expression *expr* and a procedure

```
procedure p()
   suspend expr
end
```

both *expr* and p() have the same result sequence, provided there are no side effects or dependencies in *expr* on the values of local identifiers.

For example,

```
procedure odd()
   suspend (i := 1) | |(i +:= 2)
end
```

encapsules the expression

```
(i := 1) | |(i +:= 2)
```

and both odd() and this expression have the infinite result sequence {1, 3, 5, 7, ...}.

Procedures and functions, which are simply built-in procedures, are data objects in Icon. Thus the value of write is a function; write is a global identifier whose initial value is a function. Similarly, a procedure declaration causes the name of the procedure to be a global identifier whose initial value is the procedure itself. The term procedure is used subsequently to refer to functions as well as procedures.

In a call of the form

$expr(expr_1, expr_2, \ldots, expr_n)$

expr can be any procedure-valued expression. Consider, for example, the following list of two procedures

```
plist := [find, upto]
```

Then

```
plist[1](s1, s2)
```

is equivalent to

```
find(s1, s2)
```

The expression that produces the procedure value also can be a generator. For example, the result sequence for

```
(!plist)(s1, s2)
```

consists of the concatenation of the result sequences for find(s1, s2) and upto(s1, s2).

2.3 Co-Expressions

The only way that an expression can be resumed to produce a sequence of results is by iteration or goal-directed evaluation. Consequently, the results that an expression can produce are strictly limited to the lexical site of the expression.

Co-expressions overcome this limitation. A co-expression 'captures' an expression and its environment so that the expression can be explicitly resumed at any time and place.

The expression

```
create expr
```

produces a co-expression for *expr*. This co-expression is a data object that consists of the information that is necessary to evaluate *expr*: a reference to *expr* itself, a location that indicates where the evaluation of *expr* is to resume, and copies of the local identifiers that are referenced in *expr*. For example,

```
e := create find(s1, s2)
```

assigns to e a co-expression for the expression find(s1, s2). If s1 and s2 are local to the procedure in which this expression occurs, the co-expression contains copies of these identifiers with the values that s1 and s2 have when the create is performed. The expression find(s1, s2) is *not* evaluated when the create is performed.

A co-expression is *activated* by the operation

```
@e
```

When a co-expression is activated, its expression is resumed to produce a result (when a co-expression is activated the first time, the expression is evaluated to produce its first result). For example,

```
write(@e)
```

writes the first position at which s1 occurs as a substring of s2. The activation of a co-expression fails if its expression does not produce a result. Thus

```
e := create find(s1, s2)
while write(@e)
```

is equivalent to

```
every write(find(s1, s2))
```

Since activation is an explicit operation, the results of an expression can be produced wherever or whenever they are needed. For example,

```
e := create find(s1, s2)
while write(@e) do
   @e
```

writes the odd-numbered results from the sequence for find(s1, s2).

The operation

```
*e
```

produces the number of results that have been produced by activating e — its current 'size'. For example

```
e := create find(s1, s2)
while @e
write(*e)
```

writes the number of positions at which s1 occurs as a substring of s2

The activation of a co-expression fails after its expression has produced its last result, and the 'size' of the co-expression does not increase

The operation

```
^e
```

produces a copy of the co-expression e with its evaluation location and the values of its local identifiers restored to the values they had when e was created Thus the refresh operation provides a means of repeating the sequence of results for an expression For example

```
e := create ("L" || (1 to 1000))
write(@e)
write(@e)
e := ^e
write(@e)
```

writes

```
L1
L2
L1
```

When a refreshed copy of a co-expression is produced, the copies of the local identifiers in the co-expression are restored to their values at the time the co-expression was created Global identifiers are not affected by refreshing

For more information on co-expressions, see Wampler and Griswold (1983)

3. The PDCO Facility

The PDCO facility is an extension to Icon It relies on co-expressions to provide the control over expression evaluation and resumption that is necessary to define control operations

The expression

p{$expr_1$, $expr_2$, ..., $expr_n$}

indicates a call of the procedure p with a single argument that consists of a list of co-expressions for $expr_1$, $expr_2$, ..., $expr_n$ That is,

p{$expr_1$, $expr_2$, ..., $expr_n$}

is equivalent to

p([create $expr_1$, create $expr_2$, ..., create $expr_n$])

Thus when p is called, $expr_1$, $expr_2$, ..., $expr_n$ are not evaluated but instead are passed to p as a list of co-expressions The procedure p can then activate these co-expressions as necessary to perform a desired control operation The number of arguments in the call is not limited Some control operations may expect a fixed number of co-expressions, while other control operations may operate on an arbitrary number of co-expressions

The braces in place of the usual parentheses to indicate a procedure call serve two purposes (1) they obviate the writing of the list and creation expressions, and (2) they differentiate visually between an ordinary procedure call and the invocation of a *control procedure*

An example of the use of this facility is given by the control operation

Alt{$expr_1$, $expr_2$}

that models the control structure

$expr_1$ | $expr_2$

The control procedure is

```
procedure Alt(a)
   local x
   while x := @a[1] do suspend x      # produce sequence for first expression
   while x := @a[2] do suspend x      # produce sequence for second expression
end
```

which is invoked as

Alt{$expr_1$, $expr_2$}

The expressions a[1] and a[2] are co-expressions for $expr_1$ and $expr_2$, respectively. Alt first activates a[1] repeatedly, suspending with each result for $expr_1$. When the activation of a[1] fails, the same process is performed for a[2]. This control procedure shows how simple alternation really is.

Note that the way Alt is written, it must be called with two arguments. A check on the size of a could be added to detect a call with on incorrect number of arguments.

Since a control procedure can be called with an arbitrary number of arguments, it is easy to generalize operations like Alt. For example,

```
procedure Galt(a)
   local e, x
   every e := !a do                   # get next expression
      while x := @e do suspend x      # produce sequence for expression
end
```

produces the alternation of an arbitrary number of arguments (the generator !a produces the co-expressions in the list from left to right). For example, the result sequence for

Galt{1 to 5, 4 to 6, 2 to 5}

is {1, 2, 3, 4, 5, 4, 5, 6, 2, 3, 4, 5}.

4. Examples

The following sections present a number of examples of the use of PDCO. First some of the built-in generative control structures of Icon are modeled using PDCO. Next some examples of control operations that are not built into Icon are introduced. Finally, the power of PDCO is demonstrated by the definition of new argument evaluation regimes.

4.1 Modeling Built-In Control Structures

4.1.1 Iteration

The relationship between the traditional control structure

while $expr_1$ do $expr_2$

in which $expr_1$ is repeatedly evaluated, and

every $expr_1$ do $expr_2$

in which $expr_1$ is repeatedly resumed, is shown in the following model for iteration

```
procedure Every(a)
   while @a[1] do {               # resume first expression
      @a[2]                       # evaluate second expression
      a[2] := ^a[2]               # refresh for next time
      }
end
```

That is

Every{$expr_1$, $expr_2$}

models

every $expr_1$ do $expr_2$

Note that a refreshed copy of the second argument co-expression is made after it is activated. This corresponds to the fact that $expr_2$ is *evaluated* anew for each result produced by the *resumption* of $expr_1$. This procedure can be made more concise by noting that the refreshed copy can be activated directly without changing the second value in the argument list:

```
procedure Every(a)
   while @a[1] do @^a[2]
end
```

This procedure assumes that it is called with two arguments. A check on the size of a can be added easily to take care of the common usage

every $expr$

4.1.2 Limiting Result Sequences

The limitation control structure

$expr_1$ \ $expr_2$

can be modeled by the following control procedure:

```
procedure Limit(a)
   local i, x
   while i := @a[2] do {          # get limit
      every 1 to i do             # produce sequence to limit
         if x := @a[1] then suspend x
         else break
      a[1] := ^a[1]
      }
end
```

In Limit the second argument co-expression is repeatedly activated in a loop to produce a sequence of limits, i, for activations of the first argument. The first argument co-expression is activated repeated and Limit suspends with each result it produces. When the inner loop is completed, a refreshed copy of the first argument is made for use with subsequent values of i. The second argument co-expression is activated again in the outer loop, and so on.

4.1.3 Repeated Alternation

Repeated alternation is modeled by the following control procedure:

```
procedure Repalt(a)
   local x
   repeat {
      while x := @a[1] do suspend x  # produce the sequence
      if *a[1] = 0 then fail         # exit on empty sequence
      else a[1] := ^a[1]             # else refresh and repeat
      }
end
```

After suspending with the sequence of results for the argument, the size of the co-expression is checked. If it is zero, indicating that no results were produced, the procedure terminates. Otherwise, the argument co-expression is refreshed and the loop is repeated.

It is worth noting that repeated alternation can be used to make the coding of some control procedures more concise. The expression

```
suspend |@a[i]
```

suspends with the same sequence of results as

```
while x := @a[i] do suspend x
```

For example, Galt can be written as

```
procedure Galt(a)
   local e
   every e := !a do suspend |@e
end
```

Consider, however, the following proposed revision

```
procedure Galt(a)
   suspend |@!a
end
```

This procedure does not work as intended, since the generator !a is resumed before the repeated alternation. The analysis of the result sequence produced by this procedure is a good test of understanding of generators and argument evaluation in Icon. See Section 4.2.4.

4.2 New Control Operations

While the implementation of built-in control operations using PDCO demonstrates its capabilities and illustrates programming techniques used in control procedures, the really interesting applications involve control operations that are not built into Icon.

4.2.1 The LISP Conditional Control Structure

The LISP conditional control structure, *cond* (McCarthy, 1965), is an example of a control structure that does not appear in Icon and has no direct relation to generators. It can be modeled by

Lcond{$expr_1, expr_2, \ldots, expr_n$}

where n is even. Beginning with $expr_1$, every other 'test' expression is evaluated from left to right until one succeeds (corresponding to a value that is not *nil* in LISP). The argument immediately to the right of this one is evaluated and produces the result of the control operation. If none of the tests succeeds, Lcond fails (in LISP the result is undefined). It is natural to adapt the LISP form of *cond* to Icon so that the selected expression produces a sequence of results, not just one.

```
procedure Lcond(a)
   local i
   every i := 1 to *a by 2 do          # test expressions
      if @a[i] then {
         suspend |@a[i + 1]            # produce selected sequence
         fail
         }
end
```

Even more natural to Icon is the combination of the test with the selection, so that the first expression that succeeds provides the result sequence for the control operation

```
procedure Cond(a)
   local i, x
   every i := 1 to *a do
      if x := @a[i] then {             # test for success
         suspend x                     # produce first result
         suspend |@a[i]                # produce rest of results
         fail
         }
end
```

4.2.2 Selecting Results from Sequences

In the limitation control structure

$expr_1 \setminus expr_2$

if the value of $expr_2$ is i, results 1, 2, ..., i are produced from the result sequence for $expr_1$. This is just a special case of selecting results i_1, i_2, ... from the result sequence for $expr_1$. If this selection operation is called Select then, for example, the result sequence for

```
Select{octcode(), odd()}
```

is {000, 002, ..., 008, 010, ..., 100, ..., 176}

In order to make a reasonable implementation of this control operation possible, the selecting values are required to be in monotone nondecreasing order. The control procedure for this operation is

```
procedure Select(a)
   local i, j, x
   j := 0
   while i := @a[2] do {               # selector
      while j < i do                   # count through the sequence
         if x := @a[1] then j +:= 1    # count up
         else fail                     # or exit if none
      if i = j then suspend x          # produce the selected one
      else stop("selection sequence error")
      }
end
```

The control operation fails if the result sequence for the first argument is depleted before the selecting value reached. If a selecting value is not in monotone nondecreasing order, program execution is terminated with an error message

4.2.3 Limited Iteration

Expressions of the form

every $expr_1$ \ $expr_2$

occur so frequently in Icon that a control operation for explicitly resuming an expression a fixed number of times is useful. The control operation

Resume{$expr_1$,$expr_2$}

does this. The control procedure is

```
procedure Resume(a)
   local i
   while i := @a[2] do {              # number of resumptions
      every 1 to i do @a[1] | fail
      a[1] := ^a[1]                   # refresh first argument
      }
end
```

4.2.4 Collating Results

Because of the order of resumption in iteration and goal-directed evaluation, it is not possible to produce the results from several expressions 'in parallel'. For example, if a1 and a2 are lists, alternate results cannot be obtained from them by using !a1 and !a2.

It is easy to formulate a control operation, Colseq, that produces results from several expressions in parallel. In fact the proposed, but incorrect compact version of Galt performs just this control operation:

```
procedure Colseq(a)
   suspend |@!a
end
```

Thus the result sequence for

Colseq{1 to 5, 6 to 10}

is {1, 6, 2, 7, 3, 8, 4, 9, 5, 10}.

In this formulation, if the result sequence for one expression is depleted before another, the remaining results from the longer sequences are produced. Therefore the result sequence for

Colseq{1 to 3, 6 to 10}

is {1, 6, 2, 7, 3, 8, 9, 10}.

4.2.5 Comparing Result Sequences

Another operation that cannot be performed with the built-in control operations of Icon is the comparison of two result sequence to determine if they are the same. The following control procedure performs this operation, failing if the result sequences for the two expressions are different, but returning the length of the result sequences if they are the same.

```
procedure Comseq(a)
   local x1, x2
   while x1 := @a[1] do              # result from first compared
      (x1 === @a[2]) | fail          # to result from second
                                     # fail if second is longer
   if @a[2] then fail else return *a[1]
end
```

Note that this procedure does not terminate if the result sequences are the same and they are infinite in length. It is easy to modify this control procedure to limit the comparison to a finite number of results.

4.2.6 Random Argument Resumption

The following control procedure is somewhat whimsical but it suggests some unusual possibilities for control operations

```
procedure Ranseq(a)
   local x
   while x := @?a do suspend x
end
```

Ranseq repeatedly selects at random an argument co-expression to be activated Note that Ranseq terminates if activation of the selected co-expression fails, even if there are remaining results for other arguments

4.3 Argument Evaluation Regimes

4.3.1 Lifo Resumption

As described in Section 2 2, the call of a procedure amounts to the evaluation of a list of expressions the first of which produces the procedure to which the remaining values are supplied The evaluation of the expressions is from left to right In the case that an expression fails, the previous expression is resumed This lifo resumption is built into the evaluation of all procedure calls and is implicit in Icon (Wampler and Griswold 1983a) In the absence of side effects, the left-to-right order of argument evaluation is not important The order of resumption is important, since it determines the order in which the possible combinations of argument values are produced

Modeling the built-in argument evaluation regime of Icon illustrates the capabilities of PDCO and also focuses attention on an essential aspect of expression evaluation in Icon

Since the expression that produces the procedure to be applied is not treated any differently from the other expressions in the argument evaluation process, a call that has the form

$expr_1(expr_2, expr_3, \ldots, expr_n)$

can be modeled by the control operation

Lifo$\{expr_1, expr_2, expr_3, \ldots, expr_n\}$

There are two parts to the modeling of a procedure call (1) the evaluation of the argument expressions and (2) the invocation of the procedure If the invocation of the procedure fails, both parts are repeated This process is repeated until there are no more combinations of argument values

The control procedure is

```
procedure Lifo(a)
   local i, x
   x := list(*a)                        # list for argument values
   i := 1
   repeat {
      while 1 <= i <= *a do
         if x[i] := @a[i] then {        # if argument produces value, go to next
            i +:= 1
            a[i] := ^a[i]               # refresh it
            }
         else i -:= 1                   # if argument fails, go back to previous one
      if i < 1 then fail                # fail if first argument failed
      else {
         suspend Call(x)                # else call function
         i := *a                        # set up to resume last argument
         }
      }
end
```

The do clause in the while loop is evaluated as long as i is in range of the argument list. If an argument produces a result, i is incremented. In this case the next argument is refreshed so that it will produce its first result when it is next activated. This expression fails if the new value of i is greater than *a, but this does not matter since the while loop terminates immediately.

If an argument does not produce a result, i is decremented so that the previous argument is activated again on the next iteration of the while loop.

The while loop terminates when i is either less than 1 or greater than *a. The former case occurs if there is no combination of argument values to pass to the procedure. The latter case occurs when there is a combination of argument values to pass to the procedure.

The procedure Call implements the actual application of the procedure to its arguments. Note that if Call fails, the argument evaluation process resumes argument expressions to provide another list of argument values for Call. On the other hand, if call succeeds, the value it produces is produced by Lifo. If Lifo is resumed again, either because of iteration or goal-directed evaluation, Call is resumed first. Argument expressions are resumed only if Call fails.

Since there is no way in Icon to invoke a procedure with an arbitrary number of arguments, the invocation in Call is broken down into cases according to the size of a.

```
procedure Call(a)
   suspend case *a of {
      1:  a[1]()
      2:  a[1](a[2])
      3:  a[1](a[2], a[3])
      4:  a[1](a[2], a[3], a[4])
      5:  a[1](a[2], a[3], a[4], a[5])
      6:  a[1](a[2], a[3], a[4], a[5], a[6])
        :
      default:  stop("too many arguments to Call")
      }
end
```

Note that a[1] is the procedure that is actually invoked. If the invocation succeeds, Call suspends with the result so that if Lifo is resumed, a[1] is resumed in turn.

4.3.2 Fifo Resumption

Although lifo resumption is built into Icon it is not the only way that argument lists can be produced An alternative method is *fifo* resumption, in which the first rather than the last, argument is resumed if invocation of the procedure fails For convenience, arguments are evaluated from right to left

This alternative argument evaluation regime requires only a small variation on the control procedure Lifo

```
procedure Fifo(a)
   local i, x
   x := list(*a)
   i := *a
   repeat {
      while 1 <= i <= *a do
         if x[i] := @a[i] then {
            i −:= 1
            a[i] := ^a[i]
            }
         else i +:= 1
      if i > *a then fail
      else {
         suspend Call(x)
         i := 1
         }
      }
end
```

The difference between lifo and fifo resumption is illustrated by the order of the calls for an expression like

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

In the model for lifo resumption, the call is

```
Lifo{find, s1 | s2, s3 | s4}
```

and the order of invocation of **find** is

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

In the model for fifo resumption, the call is

```
Fifo{find, s1 | s2, s3 | s4}
```

and the order of invocation of **find** is

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

These two different orders of invocation of **find** produce the same results but in different orders In lifo resumption, the last argument is 'varied' first, which the converse is true in fifo resumption Either order might be preferred, depending on the situation In lifo resumption, the primary concern is on the positions of a substring in different strings, while in fifo resumption, the primary concern is where different substrings occur in a string

4.3.3 Parallel Resumption

One of the well-known deficiencies of lifo resumption is its inability to allow parallel evaluation of expressions (Griswold, Hanson and Korb, 1981). A special case of parallel generation is given in Section 4.2.4, but there is no control regime for parallel evaluation. One approach to parallel evaluation is to simply resume every argument each time a new list of argument values is required. The control procedure for doing this is considerably simpler than for lifo and fifo resumption.

```
procedure Parallel(a)
   local i, x
   x := list(*a)
   repeat {
      every i := 1 to *a do
           x[i] := @a[i] | fail
      suspend Call(x)
      }
end
```

Evaluation stops when any argument fails to produce a value.

The usefulness of parallel resumption is illustrated by the following call

```
Parallel{|write, octcode(), |"     ", deccode(), |"     ", hexcode()}
```

In this expression, octcode is a generator of octal codes as given in Section 2.2, while deccode and hexcode are similar generators of decimal and hexadecimal codes, respectively.

Since all argument are resumed in parallel, arguments, such as write that would be single values in lifo or fifo resumption are generated repeatedly in a parallel resumption call.

The result of evaluating this expression is a table of corresponding octal, decimal, and hexadecimal codes. The expression terminates when any of the generators, such as octcode, is depleted. The form of the output for this example is

```
000         000         00
001         001         01
002         002         02
003         003         03
004         004         04
005         005         05
006         006         06
007         007         07
010         008         08
011         009         09
012         010         0A
013         011         0B

             .
             .
```

5. Implementation

The implementation of PDCO is quite simple. All that is necessary is a mechanism for translating the syntax for invoking control procedures into standard Icon. That is, expressions of the form

p$\{expr_1, expr_2, \ldots, expr_n\}$

must be translated into

p([create $expr_1$, create $expr_2$, ..., create $expr_n$])

There are several ways of doing this. One is a preprocessor. Such a preprocessor, to be correct and general must accurately parse Icon programs.

Instead a variant Icon translator was constructed. This was easy to do, since the Icon parser (Griswold, Mitchell and Wampler, 1983) is generated automatically by yacc (Johnson, 1978). A rule was added to the grammar to recognize constructions of the form

p{$expr_1$, $expr_2$, ..., $expr_n$}

with a semantic action to produce the same result as

p([create $expr_1$, create $expr_2$, ..., create $expr_n$])

Because normal procedure invocations and control procedure invocations can be nested within each other, it is necessary to maintain a stack in the parser-generator whose top value indicates whether an expression in an argument list is to have co-expression creation code inserted.

Note that standard Icon syntax is a proper subset of the PDCO syntax. The variant translator therefore correctly processes standard Icon programs.

6. Limitations

One possible objection to PDCO is that it provides no syntactic support for casting defined control operations as control structures. For example, there is no way to cast Select{$expr_1$, $expr_2$} as a syntactically distinguished construction such as

$expr_1$ \\ $expr_2$

This objection is not really relevant to PDCO, which is designed to provide a means of adding control operations to the repertoire of built-in ones much as procedures provide a means of adding to the repertoire of built-in functions. Indeed, different programs may use different control operations. Casting these in a control structure syntax would be little more useful than casting every procedure in a different syntax.

Problems in the programming and use of control operations are more significant. The absence of some features in Icon forces awkward program constructions. Most of these problems arise because Icon procedures cannot be declared with an arbitrary number of parameters. Furthermore, there is no way for an Icon procedure to determine how many arguments have been passed to it.

These considerations motivated the model of control procedures with a single argument that is a list of co-expressions. This in itself if not particularly awkward, since any built-in language mechanism for dealing with an arbitrary number of arguments would necessarily involve some syntactic overhead also.

The most offensive instance of this problem occurs in the implementation of argument evaluation regimes where Call applies an actual procedure to a list of arguments. There is no way to sublimate the problem at this point and a number of special cases must be written explicitly. In theory there is the additional problem that no fixed number of cases can handle the general instance, but in practice this is not an important consideration. Fortunately this manifestation of the problem can be isolated in one procedure.

There are, however, more significant problems with the PDCO facility. These have to do with scope, context, and dereferencing.

When an co-expression is created, copies of the local identifiers in the expression are made. These copies then have no further connection with the corresponding local identifiers in the procedure in which the co-expression was created. Consequently, assignment to a local identifier in a co-expression has no effect on the value of the corresponding identifier outside that co-expression. Thus local identifiers cannot be used to communicate values between the arguments in a control operation. An expression such as

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

does not work properly when cast as the control operation

```
Every{i := find(s1, s2), write(i)}
```

unless i is a global identifier

There are other manifestations of this problem. While

```
every (i := odd()) \ 7
```

assigns the seventh result in the octal sequence to i, the corresponding control operation does not affect the value of i unless i is global.

Another scoping problem occurs when co-expressions are refreshed, since the values of the copies of the local identifiers in the co-expression are restored to the values they had at the time the co-expression was created. Thus a co-expression cannot use local identifiers for memory after it is refreshed. For example, in

```
i := 0
every write(|(i +:= 1) \ (1 to 3))
```

the values written are 1, 2, 3, 4, 5, and 6. On the other hand,

```
i := 0
every write(Repalt{i +:= 1} \ (1 to 3))
```

writes 1, 1, 2, 1, 2, 3 unless i is global.

These scoping problems can be circumvented at the expense of program organization by using only global identifiers in control operations.

There are also problems of syntactic context. For example, the return expressions return, fail, and suspend cannot occur in the scope of a create, since they subsequently could be used out of context. While it is possible (and good idiomatic Icon) to use expressions of the form

```
expr | fail
```

such as appears in Comseq

```
Alt{expr, fail}
```

is syntactically erroneous.

More seriously, break and next cannot occur in the scope of create unless they are within loops that are in the scope of create. Therefore the common Icon idiom

```
every expr do
   if expr1 then expr2 else break
```

cannot be cast as

```
Every{expr,if expr1 then expr2 else break}
```

A less obvious, but sometimes annoying problem occurs because the result produced by activating a co-expression always is dereferenced. Control procedures therefore can only return values, not variables. While it is possible, if obscure, to assign 0 to three identifiers by

```
every (x | y | z) := 0
```

the expression

```
every Galt{x, y, z} := 0
```

produces a run-time error, since Galt returns only the *values* of the identifiers. This problem exists whether or not x, y, and z are local or global and no matter how Galt is written, since the activation operation always produces a dereferenced result. Since there is no fundamental reason why the activation operation has to dereference global identifiers, this problem can be attributed to the implementation of co-expressions instead of to language design.

7. Conclusions

Programmer-defined control operations in Icon have proved to be useful in two ways. In the first place they have been used in a number of programming situations where the existing features of Icon are inadequate. An example is the generation of tabular text, in which the parallel resumption of arguments is particularly apt. More importantly, programmer-defined control operations have provided insights into the interaction of generators and sequencing of expression evaluation. PDCO provides a tool that can be used to verify conjectures and to stimulate new ideas.

Experience has shown that despite the limitations mentioned in the preceding section, PDCO is nonetheless a useful facility. This is probably largely due to the fact that the built-in control structures are adequate for handling those cases that would otherwise produce problems if defined control operations had to be used. The situation would be quite different if, for example, *every-do* were not built into the control repertoire of Icon.

A facility for programmer-defined control operations in the style of PDCO can be added to any programming language, such as LISP, in which expressions can be treated as data objects. Unevaluated expressions in SNOBOL4 and in SL5 (Griswold and Korb, 1977) are in fact quite similar to co-expressions in Icon and the techniques of PDCO can be carried over into these languages in a straightforward way. In fact, the extended function definition facility for SNOBOL4 (Druseikis and Griswold, 1973) and SL5's defined argument transmission mechanism allow arguments to be transmitted by expression without any additional support. In other languages, a preprocessor can be provided.

The usefulness of programmer-defined control operations in other programming languages is debatable, however. Most of the interesting applications of PDCO depend on the properties of generators. For example, when an unevaluated expression is evaluated in SNOBOL4, it can only fail or produce a single result. It is the richness of expression evaluation provided by sequences of results that makes programmer-defined control operations potentially valuable for users instead of being just a programming language design tool. The emergence of generators as a general aspect of expressions in other languages (Budd, 1982) suggests a growing area of applicability for defined control operations.

The most promising areas for further exploration of control operations in Icon appear to lie in argument evaluation regimes and control of procedure invocation. Work in these areas will be presented in a subsequent report.

Acknowledgements

The authors are indebted to Steve Wampler for the co-expression facility upon which PDCO is built. Steve Wampler and Dave Hanson provided a number of helpful suggestions on the PDCO facility and on the presentation of the material in this report.

References

Budd, T. A. (1982). An implementation of generators in C. *Computer Languages*, Vol. 7, 69-87.

Druseikis, F. C. and Griswold, R. E. (1973). *An Extended Function Definition Facility for SNOBOL4*. Technical Report S4D36, Department of Computer Science, The University of Arizona.

Fisher, D. A. (1970). *Control Structures for Programming Languages*, Ph.D. dissertation, Computer Science Department, Carnegie-Mellon University.

Griswold, R. E. (1982). The evaluation of expressions in Icon. *TOPLAS*, Vol. 4, No. 4, pp. 563-584.

Griswold, R. E., and M. T. Griswold (1983). *The Icon Programming Language*, Prentice-Hall, Inc.

Griswold, R. E., Hanson D. R., and Korb, J. T. (1981). Generators in Icon. *TOPLAS*, Vol. 3, No. 2, pp. 144-161.

Griswold, R E and Korb J T (1977) *A Catalog of Built-In SL5 Operators and Functions*, Technical Report S5LD3g, Department of Computer Science, The University of Arizona

Griswold R E Mitchell W H and Wampler, S B (1983) *The C Implementation of Icon, A Tour Through Version 5* Technical Report TR 83-11, Department of Computer Science, The Universsity of Arizona

Griswold, R E , Poage, J F , and Polonsky, I P (1971) *The SNOBOL4 Programming Language*, 2nd ed . Prentice-Hall Inc

Johnson, S C (1978) *Yacc Yet Another Compiler-Compiler* Bell Telephone Laboratories, Inc

Leavenworth, B M (1969) Programmer-defined control structures, *Proceedings of the Third Annual Prince-ton Conference on Information Sciences and Systems*, pp 30-34

McCarthy, J et al (1965) *LISP 1 5 Programmer's Manual*, 2nd ed , M I T Press

Morris, J B and Wells, M B (1972) The specification of program flow in Madcap 6, *SIGPLAN Notices* Vol 7, No 11, pp 28-35

Wampler, S B (1981) *Control Mechanisms for Generators in Icon* Ph D dissertation, Department of Computer Science, The University of Arizona

Wampler, S B and Griswold, R E (1983) Co-expressions in Icon, *The Computer Journal*, Vol 26, No 1, pp. 72-78

Wampler, S B and Griswold, R E (1983a) Result sequences, *Computer Languages*, Vol 8, No 1, pp 1-14

Wegbreit B (1970) *Studies in Extensible Programming Languages* Ph D dissertation, The Division of Engineering and Applied Physics, Harvard University