The Implementation of Goal-Directed Evaluation and Co-Expressions*

Stephen B. Wampler

TR 81-9

June 1981

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

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

The Implementation of Goal-Directed Evaluation and Co-Expressions

1. Introduction

Expressions in Icon are capable of generating a sequence of results and are termed *generating expressions* or *generators* [5,6,7]. For example, the expression

```
1 to 10
```

is capable of producing the results 1, 2, ..., 10. Some expression produce no result at all, such as

```
1 < 0
```

The primary control mechanism for the evaluation of generators is *goal-directed evaluation* [9]. Goal-directed evaluation provides an efficient means of control backtracking [7].

While goal-directed evaluation is a fundamental aspect of Icon, it is not restricted to Icon, and has a straightforward and efficient implementation in stack-based languages. The language Cg demonstrates that goal-directed evaluation can be a useful feature in more conventional languages [1]. Cg adds goal-directed evaluation to the C programming language through the use of a preprocessor and a few run-time support routines.

Co-expressions are data-objects that encapsulate expressions [11]. This encapsulation permits the evaluation of the encapsulated expressions to occur as needed throughout a program, in much the same way that procedures may be invoked as needed throughout a program. Co-expressions provide programming capabilities at the expression level similar to those provided at the procedure level by coroutines. Co-expressions can be used to free generators from lexical constraints, providing increased flexibility in program design.

The ease of implementing co-expressions depends to a large degree on the implementation chosen for goal-directed evaluation, and is simplest in a stack-based language. Version 4 of Icon [3] is an extension of Version 3 [2] in which co-expressions and several new control mechanisms for the evaluation of generators are implemented.

The implementation of goal-directed evaluation and co-expressions consists of two parts: run-time support and generated code. The run-time support consists of evaluation routines to provide primitive actions needed in the evaluation process. The generated code organizes these primitives into *control regimes.* Control regimes perform the semantic actions associated with various control structures [12].

This paper gives models of the implementations of goal-directed evaluation and co-expressions. The first model of goal-directed evaluation provides the most straightforward implementation, but requires the use of two physically separate stacks. The second model merges the two stacks of the first model into a single stack, providing greater efficiency and simplifying the implementation of co-expressions. The model given for the implementation of co-expressions is based on the second model for goal-directed evaluation.

The second model of goal-directed evaluation and the corresponding model of co-expressions are compared to the actual implementation of these features in Icon and Cg. Restrictions imposed upon the actual implementations by scoping conventions and machine architecture limitations are discussed.

2. Expression Instances

The evaluation of a program proceeds through a sequence of *expression instances* in much the same way that it proceeds through a sequence of procedure instances. An expression instance is the environment in which evaluation of an expression occurs. Variables that are declared as global entities, or that are local to the surrounding procedure instance, are usually treated as global to an expression instance.

There are five states in which an expression instance may exist:

1. When an expression instance is initially formed, but evaluation has yet to take place within that instance, it is said to be *created.* An expression instance for an expression may be created at any time prior to evaluation of that expression. Typically the expression instance is created just prior to the time the evaluation starts.
2. When evaluation is occurring within an expression instance, that instance is said to be *active.*
3. Because expressions are composed of other expressions, expression instances are often nested during evaluation. Thus an expression instance may become *passive* while awaiting the result from another instance. For example, assuming evaluation of j + k occurs within a separate expression instance, an expression instance for the evaluation of i > (j + k) becomes passive while the instance for (j + k) is active.
4. When an expression instance computes a result and is capable of producing subsequent results, it becomes *inactive.* An inactive expression instance may be *reactivated* to produce a subsequent result. Evaluation is said to be *suspended* within an inactive expression instance.
5. An expression instance that has produced all possible results is said to be *exhausted.* Typically expression instances are destroyed as soon as they become exhausted. An expression instance is said to *exist* between the time it is created and the time it is destroyed.

In conventional Algol-like languages, expressions instances become active as soon as they are created, become exhausted as soon as a single result is produced, and are destroyed immediately upon becoming exhausted. Thus expression instances are all either active or passive in conventional Algol-like languages. For this reason, expression instances in such languages are relatively uninteresting, amounting to little more than storage for temporary values used during evaluation of the expression. Furthermore, because expressions in these languages produce exactly one result, a single stack can be used to maintain expression instances during evaluation.

In Icon, however, expression instances can exist in any of the five states. Expression instances in Icon must therefore contain more information concerning the state of the evaluation of that instance, and an Algol-like stack is no longer sufficient to maintain expression instances.

While the information contained within an expression instance may vary depending upon language features and implementation techniques, information typically associated with Icon expression instances includes:

1. A *passive instance pointer* pointing to a linked list of enclosing passive expression instances.
2. An *inactive instance pointer* pointing to a linked list of inactive subexpressions.
3. An *activation address* acting as a pointer into the program code. The activation address assumes different meanings depending upon the state of the expression instance, and is explained in more detail later.
4. An *expression stack* used to hold any temporary results created during evaluation of the expression.

The first three items are referred to collectively as the *expression marker.* In practice, expression stacks are not separate entities, but simply represent areas on some *system stack* that are separated by expression markers. Nevertheless, it is convenient to view the system stack as a stack of expression instances, with each expression instance maintaining its own expression stack.

Maintaining expression instances for each expression adds considerable overhead to expression evaluation. However, it is often possible to coalesce expression instances by factoring the expression markers out of many expressions. The term *subexpression* is used to refer to an expression that has had its expression marker factored out to some enclosing instance. Note that there is nothing that prohibits subexpressions from containing expressions.

In Algol-like languages, expression markers are unnecessary, and all expression instances are coalesced into a single instance. Expression markers in Icon are factored to the points at which program control decisions are made. Typical control decision points are the control clauses of control structures. For example, in the expression

```
    f(if x > y then 2*x else 2*y)
```

a new expression instance is created for the evaluation of x > y, but the selected arm of the if-then-else (either

2∗x or 2∗y) is treated as a subexpression in the surrounding expression instance. Thus the above expression evaluates as either f(2∗x) or f(2∗y).

3. Operations on Expression Instances

Various implementation schemes for generators can be formulated in terms of operations upon expression instances. Program segments written in Icon are used in this paper to describe several implementation schemes for generators.

In addition to the conventional features of Version 4 of Icon, the following operations involving expression instances are assumed.

1. Creating an expression instance is accomplished by the function

   ```
   create_instance()
   ```

   which returns a created expression instance of the form:

passive	
inactive	
save_pc	
estack	

   In this figure, passive is the passive instance pointer for that expression, inactive is the inactive instance pointer, save_pc is the activation address, and estack is the expression stack. Any changes to this form dictated by different implementation approaches are indicated where appropriate.

2. An expression instance, i, can be copied using the function

   ```
   copy(i)
   ```

3. The fields of an expression instance are accessed using the field reference operator of Icon. That is,

   ```
   i.passive := &null
   ```

   clears the passive instance pointer field of instance i, and

   ```
   \i.inactive
   ```

   succeeds if the inactive instance pointer of i is non-null.

4. Expression instances can be manipulated as data objects. Because expression instances are maintained on stacks (most notably the system stack), pushing and popping them on to and off of a stack is accomplished with

   ```
   push(stack,i)
   ```

   and

   ```
   pop(stack)
   ```

   respectively. Another useful stack operation is

   ```
   popto(stack,object)
   ```

   which pops stack so that object is on top of the stack, and fails if object is not on the stack. The stack is left unchanged if popto fails.

5. Finally, the global identifiers active and ps are a pointer to the currently active expression instance and the machine location counter, respectively.

4. Goal-Directed Evaluation

Goal-directed evaluation is a control regime that produces a result if it is possible to do so. For example, in the evaluation of

```
        (1 to 10) > x
```

the subexpression 1 to 10 is capable of producing the sequence 1, 2, ..., 10. Goal-directed evaluation forces 1 to 10 to produce results from its sequence until one is produced that has value greater than the value of x. If a result produced by 1 to 10 is greater than x, then the evaluation is said to *succeed* and the *outcome* of the expression is the result computed by the expression. If the subexpression 1 to 10 is exhausted before this condition is met, then the entire expression *fails* and the outcome is *failure*.

The implementation of goal-directed evaluation ensures that all possible combinations of results from subexpressions are tried in an attempt to produce a result from the evaluation of an expression. This is accomplished by implementing goal-directed evaluation as a form of control backtracking; the most recently evaluated subexpression is reactivated first for subsequent results. Goal-directed evaluation is not a form of data backtracking because variables are not necessarily restored to previous values during the reactivation process. (There are a few operations in Icon that provide limited data backtracking.) However, any temporary results present when a subexpression produces a result are restored when that subexpression is reactivated. For example, in the expression

```
        5 + (1 to 10) > x
```

the value 5 is present as a temporary result in the currently active expression instance when the subexpression 1 to 10 produces a result. The addition operation replaces both 5 and the result produced by 1 to 10 with their sum. However, if 1 to 10 produces a subsequent result, 5 must be restored to the active instance in order for evaluation to proceed properly. The information restored corresponds precisely to the information maintained as part of the expression instance for that expression.

When an expression is to be evaluated, an expression instance is created for that expression. Evaluation then proceeds within that instance. When a subexpression produces a result, a copy of the active expression instance is saved as an inactive expression instance and evaluation proceeds using the produced result. (As an implementation optimization, copies of the active expression instance are saved only when the subexpression that has produced the result has the potential of producing subsequent results. This is accomplished by making each subexpression responsible for saving the active expression instance whenever that subexpression produces a result and is capable of producing additional results.)

If failure is encountered during evaluation, the currently active expression instance is destroyed and the most recently inactivated copy of that expression instance is activated. If there are no inactive copies of that expression instance, the failure is transmitted to the enclosing instance, if any.

When evaluation of an expression produces a result (as opposed to a subexpression producing a result), the result is provided to any enclosing instance. The currently active expression instance is destroyed, along with any inactive instances of that active instance. The enclosing expression instance becomes the active expression instance.

4.1 The Two-Stack Model of Goal-Directed Evaluation

In the early implementations of Icon, two physically distinct stacks are used to implement goal-directed evaluation [9]. All expression instances that exist, but are not inactive, are maintained on a *system* stack, denoted SYSSTK. The currently active expression instance is on the top of SYSSTK. The second stack, or *control* stack, is used to store inactive expression instances. The control stack is denoted CTLSTK.

There is no need for the passive instance pointer in expression instances, since passive expression instances are maintained in proper order on SYSSTK.

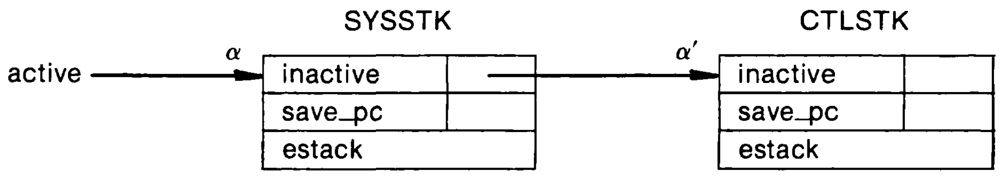
The activation address for expression instances on SYSSTK is the address to which program control is transferred whenever failure is transmitted to that instance. For instances on CTLSTK, the activation address is the address at which evaluation is to resume if the instance is reactivated.

There are three routines responsible for the implementation of goal-directed evaluation:

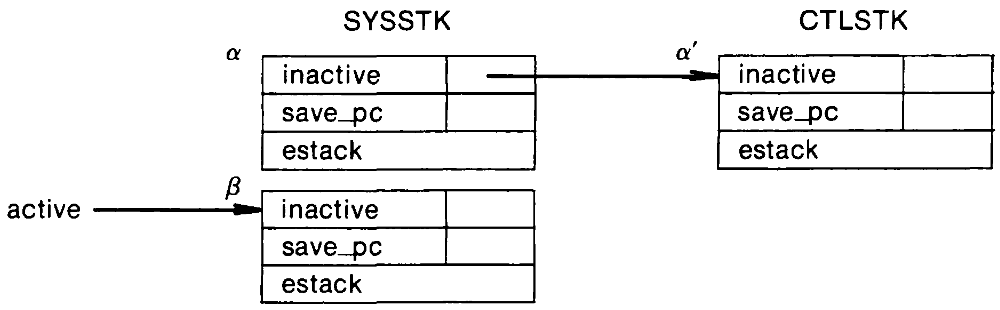
1. mark creates a new expression instance on SYSSTK and activates this instance.

```
procedure mark(failure_lab)

    push(SYSSTK, create_instance())
    top(SYSSTK).save_pc := failure_lab
    active := top(SYSSTK)

end
```

Given SYSSTK and CTLSTK prior to the call of mark



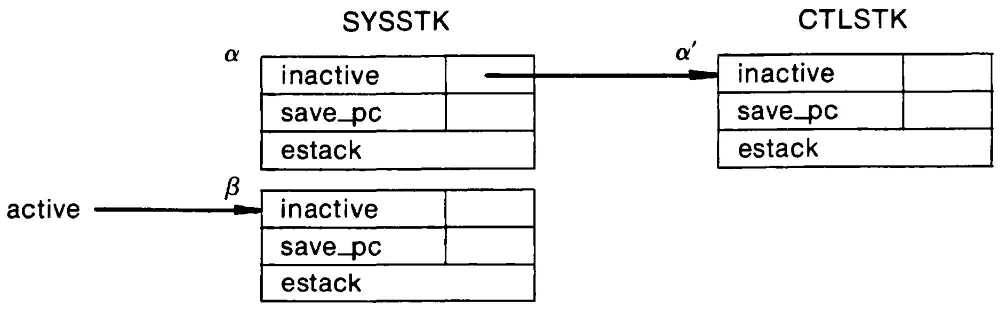
after the call they are



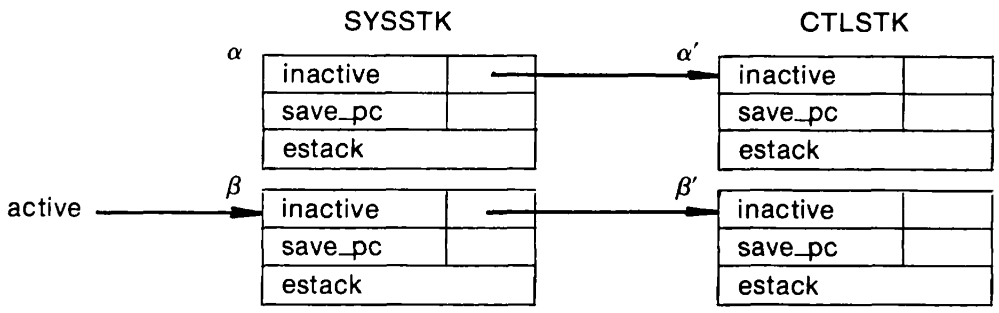
2. save saves a copy of the currently active expression instance on CTLSTK.

```
procedure save()

  push(CTLSTK, copy(active))
  top(CTLSTK).inactive := active.inactive
  top(CTLSTK).save_pc := pc
  active.inactive := top(CTLSTK)

end
```

If SYSSTK and CTLSTK before the call to save are



after the call they are

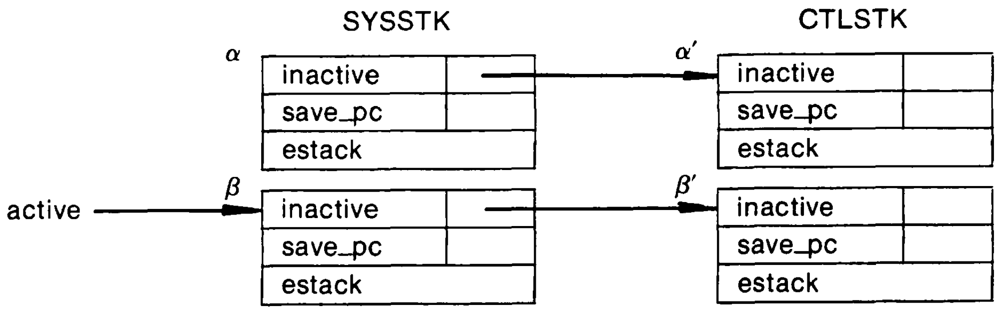


The result of the subexpression is then pushed onto the expression stack for the active instance and processing continues after the call of save().

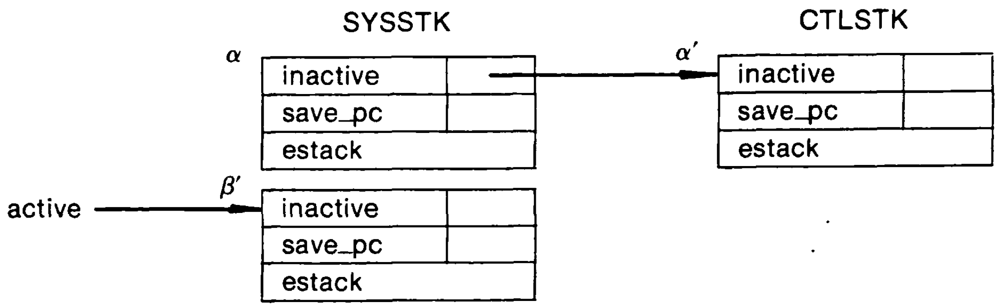
3. drive handles success or failure of expression evaluation:

```
procedure drive()

   pop(SYSSTK)
   if \failure then {
      if \active.inactive then {
         push(SYSSTK, pop(CTLSTK))
         pc := top(SYSSTK).save_pc
         failure := &null
         }
      }
   else
      popto(CTLSTK, top(SYSSTK).inactive)

   active := top(SYSSTK)

end
```

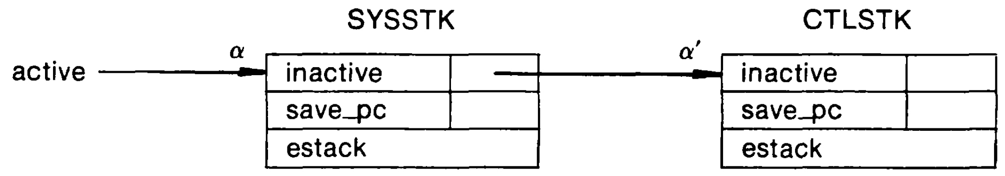
Assuming that before the call to drive the system and control stacks are



then if failure has occurred, the stacks after the call are



If there is no failure, the stacks are



Note that the value of pc after drive is different in the case of failure than it is when the expression evaluated successfully.

Calls to the routines mark and drive enclose the code for each expression requiring an expression instance. A new expression instance is created by mark upon entry to the code for an expression and is destroyed by drive upon exit from the code for that expression.

Note that drive depends upon the use of a global flag variable failure. Initially, failure has the null value. If a subexpression fails, failure is set to some non-null value and processing continues. After every subexpression that could conceivably fail, there is a test of the variable failure. If this test detects the occurrence of failure, control branches immediately to the drive at the end of the expression.

One improvement to this mechanism is to have operations directly perform the actions taken upon failure. This eliminates the need for the tests of failure after every subexpression. This enhancement is one of several presented in the following model for goal-directed evaluation.

4.2 The One-Stack Model of Goal-Directed Evaluation

Using two stacks as described above provides an effective implementation for goal-directed evaluation. Nevertheless, there are several disadvantages to using two distinct stacks. First, the use of a second stack complicates memory management in some machine architectures. Second, moving expression instances on to and off of the control stack involves additional overhead. It is possible to merge the control and system stacks into a single physical stack. The result is a more efficient implementation.

The technique is to "hide" inactive expressions instances *in place* on the system stack. Whenever a subexpression performs a save operation, a new expression instance is created containing a copy of the information necessary to continue processing (including the result being supplied from the subexpression). The new expression instance then becomes the active expression, and processing continues.

This approach has several advantages over the two-stack model. First, there is no need to copy an inactive instance back to the system stack when it is reactivated, since that instance is already on the system stack. Second, the amount of information that must be copied when an instance becomes inactive is less than that required in the two-stack model.

As an example, consider evaluation of the expression

```
5 + (1 to 10) > x
```

Just before the to operation suspends the active instance contains the value 5 as well as any temporaries formed during evaluation of 1 to 10. In the two-stack model, all of this information must be copied as part of the inactive expression instance. The one-stack model requires only copying 5 and the result produced by 1 to 10 into the new active instance, since that is all the information necessary to continue evaluation of the expression. It is assumed that the function copy_information copies the appropriate information from the currently active expression instance into the newly created expression instance.

The combination of the system and control stacks into a single stack is accomplished through a slight change in the expression marker: There is no longer any need for the inactive instance pointer. In those situations in the two-stack model where an active instance pointer points to an inactive instance, that inactive instance is now located immediately below the active instance. However, there is now a need for a passive instance pointer, since the next passive instance may not be the next instance on the stack.

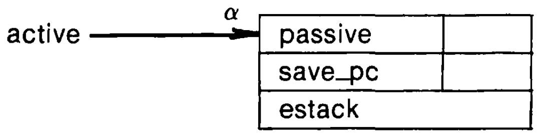
Another change to expression instances is in the use of the activation address. Whereas the activation address in the two-stack model provides the point at which control is to resume in that instance, the activation address in the one-stack model provides the location at which control is to resume in the next expression on the stack when failure occurs in the current instance. While this change is mostly cosmetic, it simplifies the implementation of some additional control structures.

These changes necessitate some modifications to the routines that control goal-directed evaluation. (In preparation for the presentation of additional language features, the global identifier active_stack is used to refer to the system stack.

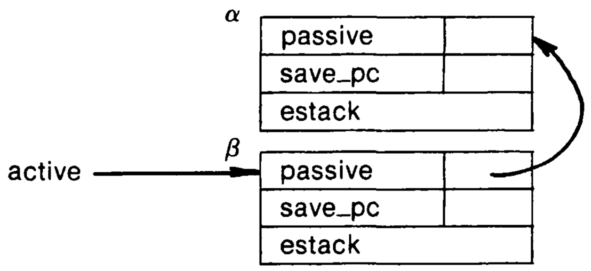
1. mark is the same as before, except that the passive instance pointer is set to point to the currently active instance.

```
procedure mark(failure_lab)

   push(active_stack, create_instance())
   top(active_stack).passive := active
   top(active_stack).save_pc := failure_lab
   active := top(active_stack)

end
```

If just before a call to mark the stack is



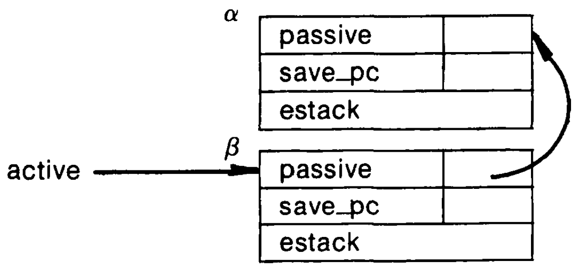
then after the call the stack is



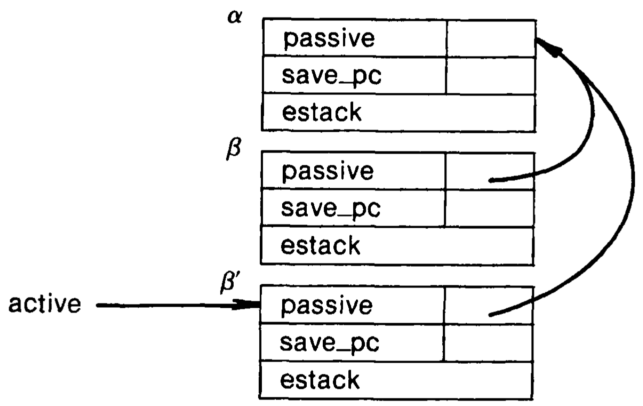
2. save hides the currently active instance in place on the stack when a subexpression suspends.

```
procedure save()

   push(active_stack, create_instance())
   copy_information(active, top(active_stack))
   top(active_stack).save_pc := pc
   active := top(active_stack)

end
```

If the stack just before a call to save is



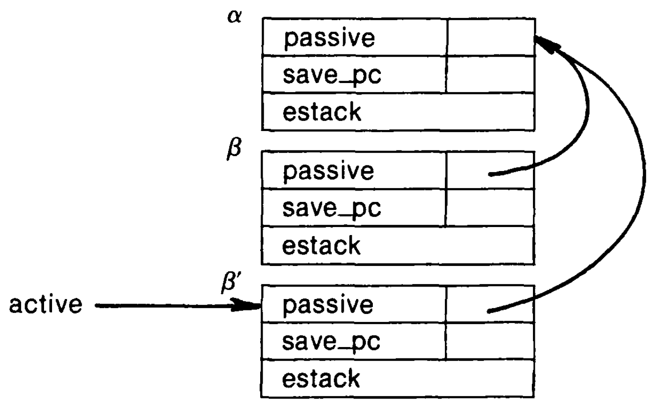
then after the call, the stack is



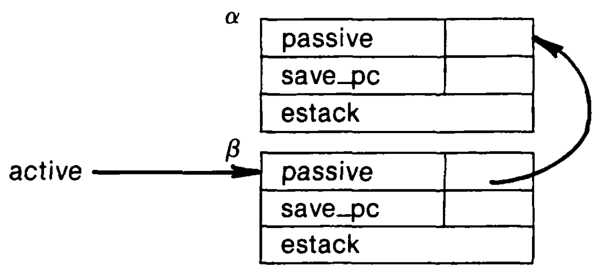
3. As stated earlier, subexpressions call the failure handling mechanism directly when they fail. The routine failure handles all failures and is invoked by the subexpression that fails. It makes no difference when failure occurs whether or not there are any inactive instances for the currently active expression. The appropriate instance to reactivate is always the next instance on the stack, and the current save_pc is the location at which execution is to continue within that instance.

```
procedure failure()

   pc := active.save_pc
   pop(active_stack)
   active := top(active_stack)

end
```

If before the call to failure the stack is



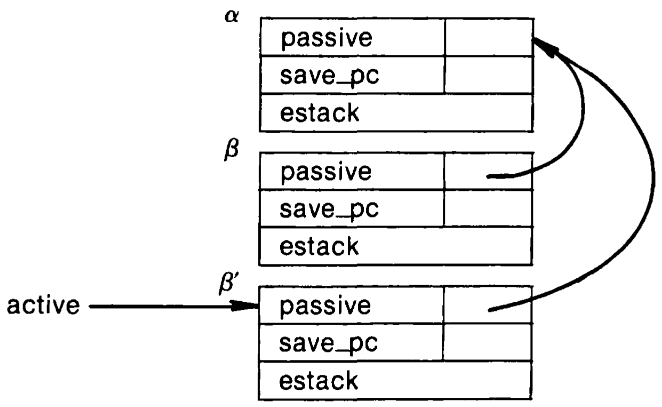
then after the call the stack is



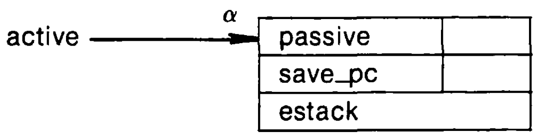
4. Because all expression failures are handled by failure, drive need only ensure that successful evaluation of an expression returns control to the next passive expression. In the one-stack model, drive has been renamed unmark, and does nothing more than pop the active stack to the passive expression instance.

```
procedure unmark()
   popto(active_stack, active.passive)
   active := top(active_stack)
end
```

If the stack before the call to unmark is



Then after unmark the stack is



4.3 Generated Code in the One-Stack Model

The code generation for a language using generators and goal-directed evaluation is straightforward. Because expression instances are created only at the points at which control decisions need to be made, most of the code production is identical to that in more conventional languages.

Control structures are another matter. The use of success or failure to control expression evaluation is directly reflected in the implementation of control structures.

Assuming a stack-based machine architecture, a simple intermediate code, called *ucode*, is used here to describe the code generated for some typical control structures based upon generators and goal-directed evaluation. These control structures have the semantics of Version 4 of Icon.

Ucode instructions that have conventional meanings are push, pop, goto, and invoke. For simplicity, it is assumed that all operators, functions, and procedures are invoked through the same mechanism. For example, the ucode produced for the expressions

```
1 + 3
1 to 10
write(3)
```

is

```
push     1
push     3
invoke   +

push     1
push     10
invoke   to
```

```
push      3
invoke    write
```

In the code shown here, comments and annotations are enclosed in braces, and labels are terminated by colons, e.g.

```
lab1:          goto       lab1       {tight infinite loop}
```

While operations that produce results do so by placing the result on the active expression stack, for descriptive convenience it is assumed that the location result also contains the result of the last operation.

The remaining ucode instructions deal exclusively with generators and goal-directed evaluation, and correspond to the procedures described in the preceding section.

1. mark *lab* is the ucode form of the procedure mark(*lab*). The universal label flab is assumed to point to an invocation of failure(). Thus, mark flab propagates failure to the first passive expression instance on failure of the marked expression instance.
2. unmark performs the same function as the procedure unmark().
3. fail corresponds to the procedure failure().
4. save() is used by a subexpression when a result is to be provided to the active expression instance. The typical use of save() is within operators and functions that are capable of producing a sequence of results. Some of the control structures require that the active expression instance provide a result to the enclosing passive instance and then become inactive. esave is used in these situations, and corresponds to the procedure

   ```
   procedure esave()
      pop(active_stack)
      push(active_stack, copy(active.passive))
      top(active_stack).save_pc := active.save_pc
      active := top(active_stack)
   end
   ```

   The approach is to replace the current active expression instance with a copy of the first passive expression instance (which then receives the result of the current active expression instance) and to continue processing.

   One of the simplest control structures is if-then-else, with Icon syntax

   if *expr0* then *expr1* else *expr2*

or

   if *expr0* then *expr1*

If there is an else clause, the generated code is

```
            mark       lab1
               {code for expr0}
            unmark
               {code for expr1}
            goto       lab2
lab1:
               {code for expr2}
lab2:
```

If the control expression produces a result, the unmark pops the active stack to the first passive expression instance, and the then clause is evaluated in that instance. If the control expression fails to produce a result, that same passive instance becomes the active instance. Control then branches via the failure mechanism to lab1, and the else clause is evaluated.

In Version 4 of Icon, if the else clause is omitted and the control clause fails, the entire expression fails. Thus the code generated when the else clause is omitted is

```
        mark      flab
            {code for expr0}
        unmark
            {code for expr1}
```

The generated code for while-do loops is straightforward. The expression

```
while expr0 do expr1
```

has the generated code

```
lab:
        mark      flab
            {code for expr0}
        unmark
        mark      lab
            {code for expr1}
        unmark
        goto      lab
```

Note that while-do itself does not produce a result.

repeat loops are even simpler. repeat loops are infinite loops, relying upon an explicit break to terminate. The generated code for

```
repeat expr
```

is

```
lab:
        mark      lab
            {code for expr}
        unmark
        goto      lab
```

In this case, if *expr* fails, control branches to the same point as when it succeeds.

Another basic control structure is every-do, which forces the control clause to produce all the results it is capable of producing, and executes the do clause each time the control clause produces a result. The generated code for

```
every expr0 do expr1
```

is

```
        mark      flab
            {code for expr0}
        pop
        mark      flab
            {code for expr1}
        unmark
        fail
```

The pop after the control clause simply removes the result computed by that clause, since that result is ignored. Evaluation of the do clause takes place within a separate expression instance to limit evaluation to at most one result from that clause. A fail occurs at the end of the code sequence instead of an unmark, and is evaluated in the expression instance for the control clause. This forces any inactive instances of the control clause to be reactived, using save_pc as the reactivation address. Again, note that every-do itself does not produce a result.

If the control clause of the not control structure succeeds, then the outcome of not is failure. If the control clause fails, then the outcome of not is &null. The generated code for

```
        not expr
```

is

```
                      mark        lab
                         {code for expr}
                      unmark
                      fail
          lab:
                      push        &null
```

Alternation, with Icon syntax

```
        expr1 | expr2
```

produces the result sequence of *expr1* followed by the result sequence of *expr2*. The generated code is

```
                      mark        lab1
                         {code for expr1}
                      esave
                      push        result
                      goto        lab2
          lab1:
                         {code for expr2}
          lab2:
```

esave is used to make the expression instance for evaluating the left control expression inactive, so that failure in the surrounding expression instance reactivates the left control expression before attempting evaluation of the right control expression.

In Versions 1 through 3 of Icon, any expression enclosed in braces is limited to producing at most one result. An expression limited to at most one result is termed a *restricted* expression. The generated code for a restricted expression *expr* is

```
                      mark        flab
                         {code for expr}
                      unmark
                      push        result
```

Restricted expressions have been subsumed in Version 4 by the *limitation* control structure. A limited expression is limited to producing no more than a specified number of results. (Hence a restricted expression is identical to an expression that is limited to at most one result.) The Icon syntax for a limited expression is

```
        expr1 \ expr2
```

which limits *expr1* to at most *expr2* results. (*expr1* is termed the *limited expression.*)

The generated code for the limitation control structure is considerably more complicated than the code generated for a restricted expression, and requires the introduction of two new ucode instructions, limit and lsave.

limit has procedural form

```
procedure limit()
  if result <= 0 then failure()
end
```

limit checks the current result and succeeds if the result is positive. limit leaves this result upon the stack to function as a counter of results left to produce from the limited expression.

lsave is responsible for maintaining the count of results produced. The top of the stack of temporaries for the first passive expression instance is the count of results left to produce. If the last result is being produced, then lsave is similar to unmark. If it is not the last result, then lsave functions the same as esave.

The procedural form of lsave is

```
procedure lsave()
   top(active.passive.estack) -:= 1
   if top(active.passive.estack) > 0 then
      esave()
   else
      unmark()
end
```

The generated code for limitation is thus

```
           {code for expr2}
        limit
        mark        flab
           {code for expr1}
        lsave
        pop
        push        result
```

The last two instructions replace the count of remaining results with the result of the limited expression for subsequent processing.

As a final example of generated code, consider *repeated evaluation*, with Icon syntax

|*expr*

Repeated evaluation produces the sequence of results from its control expression, and then repeats the evaluation of the control expression. The difficulty in implementing repeated evaluation lies in the fact that if the control expression ever fails to produce any result, repeated evaluation fails, rather than attempting to evaluate the control expression anew. If this condition were removed, the generated code would be

```
lab:
        mark        lab
           {code for expr}
        esave
        push        result
```

With the failure condition, the code is

```
lab:
        mark        flab
           {code for expr}
        chfail      lab
        esave
        push        result
```

The ucode instruction chfail changes the activation address from flab to lab after the control expression has produced a result. Thus if no result is produced by the control expression, the failure is propagated to the first passive instance enclosing the repeated evaluation. If at least one result is produced, then chfail insures that subsequent failure causes the expression to be evaluated anew.

The implementation of chfail requires an additional primitive operation to gain access to the expression instance immediately following* the passive instance that is to receive the result of the limited expression. The function one_above(i) returns a pointer to the expression instance containing the activation address that is used when reactivating expression instance i.

The procedural form of chfail is then

```
procedure chfail(failure_label)
   one_above(active.passive).save_pc := failure_label
end
```

5. Co-Expressions

Forming a co-expression from an expression involves the creation of a *co-expression instance* that encapsulates the evaluation of the expression. While co-expressions can be implemented using either the one- or two-stack model of goal-directed evaluation, implementation using the one-stack model is simplest, and is used here.

A co-expression instance encapsulating an expression can be viewed as the stack used to maintain any expression instances created during evaluation of the expression, and a location counter for that expression. It is convenient to treat the location counter and stack in which program execution is initiated as a co-expression instance. The expression in which evaluation of an Icon program is initiated consists of an invocation of the procedure main. The co-expression instance in which program evaluation is currently taking place is termed the *current* co-expression, and its stack is termed the *active stack*.

The current co-expression is pointed to by the global variable current. Evaluation, or *activation*, of a co-expression is a straightforward process of switching current from one co-expression instance to another. The first co-expression instance is termed the *activator* of the second. The expression instance on top of the stack for the activator becomes a passive instance, awaiting a result from the activated co-expression. The instance on top of the activated co-expression stack becomes the active expression instance.

When the activated co-expression produces a result, that result is transmitted back to the activator, where processing continues.

The ucode instruction create *lab* is assumed to produce a co-expression instance for evaluating the co-expression whose code begins at *lab*. This co-expression instance is represented as

activator	
pc	
sstack	

and is modeled in the implementation model as a record

```
record coexpr(activator, pc, sstack)
```

where activator points to the current activator of that co-expression, pc is the location counter for that instance, and sstack is the stack of expressions instances for the co-expression.

create corresponds to the procedure create_coexpr.

* The activation address for this instance is the point at which control is to resume in the passive instance when failure occurs.

```
procedure create_coexpr(first_instr)
   push(active.estack, coexpr(&null, first_instr, stack()))
   push(top(active.estack).sstack, create_instance())
end
```

Note that an initial expression instance is built into the co-expression instance stack. This is done so that the activation process need not determine whether or not the co-expression instance stack is empty.

The procedure activate switches to a new co-expression instance. In addition, since it is reasonable to transmit a result to the new co-expression instance, activate provides the current result to the new co-expression instance*.

```
procedure activate(coexpr)
   coexpr.activator := current
   current.location_counter := location_counter
   current := coexpr
   active_stack := current.sstack
   active := top(active_stack)
   location_counter := current.location_counter
   push(active.estack, result)
end
```

The procedure coreturn provides a result from one co-expression to its activator.

```
procedure coreturn()
   current.location_counter := location_counter
   current := current.activator
   active_stack := current.sstack
   active := top(active_stack)
   location_counter := current.location_counter
   push(active.estack, result)
end
```

Note that there are very few differences between activate and coreturn. activate sets the activator field of the activated co-expression, while coreturn simply returns control to its activator.

If a co-expression becomes exhausted, failure is reported to the activator. The procedure cofail is invoked when a co-expression fails.

```
procedure cofail()
   current.location_counter := location_counter
   current := current.activator
   active_stack := current.sstack
   active := top(active_stack)
   failure()
end
```

5.1 The Generated Code for Co-Expressions

The ucode operations create, coreturn, and cofail correspond to the procedural forms given earlier. Activation,

expr1 @ expr2

is like any other binary operator, and has generated code

* Any co-expression that is the activator of some other co-expression has a passive expression instance on the top of its stack If the activated co-expression is not the activator of some other co-expression, the result is ignored.

```
                    {code for expr1}
                    {code for expr2}
                 invoke      activate
```

The Icon expression

```
    create expr
```

creates a co-expression from *expr*. The code generated for co-expression creation is a bit more complex than that generated for co-expression activation. The approach is to branch around the code generated for the co-expression and then do a create with a pointer to the code for the co-expression. Hence the Icon expression

```
    create expression
```

produces the ucode instructions

```
                 goto        lab2
      lab1:
                 pop
                 mark        clab
                    {code for expression}
                 coreturn
                 fail
      lab2:
                 create      lab1
```

The pop at the beginning of the co-expression code removes the result provided by activate, since that result is ignored when the co-expression is first activated.

When a co-expression is exhausted, it fails any time that it is subsequently activated. The mark clab causes a branch to the universal label clab when the co-expression is exhausted. The code generated at clab is

```
      clab:
                 cofail
                 goto        clab
```

which repeatedly transmits failure back to any activator of the co-expression.

Finally, the fail after coreturn forces the co-expression to produce its next result the next time it is activated.

6. Actual Implementation Details

For pedagogical purposes, a ucode generator and a ucode interpreter for a small subset of Icon have been written in Icon using the routines and examples presented above. However, the use of Icon obscures a number of practical considerations that are encountered when implementation is attempted using a conventional system implementation language. A major difficulty arises because system stacks are typically addressed in terms of machine words or bytes, not expression instances as in the Icon model.

This section presents modifications to the models that are necessary to add goal-directed evaluation and co-expressions to a language based upon conventional implementation techniques. The modified models reflect the general approaches taken in the implementation of Cg and Icon, though both Cg and Icon include features and optimizations not presented here.

6.1 Goal-Directed Evaluation

Because conventional implementation languages treat the system stack as a stack of words or bytes, expression instances are represented by expression markers separating expression stack areas. In the one-stack model, it is assumed that active and any passive instance pointers point to expression markers. Expression markers are of some fixed length, while expression stack areas vary in size depending upon the number of temporary results created while that instance is active. The *modified* one-stack model assumes this more conventional layout of system stacks.

In a conventional system stack, the top of the stack is pointed to by some register, termed the *stack pointer*. In the modified one-stack model, this stack pointer corresponds to a pointer to the top of the expression stack for the active expression instance, and is represented by the global identifier sp.

When expression instances are popped from the system stack, two actions occur. First, active is changed to point to the expression marker for the new active instance. Second, sp is changed to point to the top of the new active instance.

When there are no inactive instances of the current active expression, these two operations are accomplished by setting sp to the current value of active, and setting active to the current passive instance pointer. These actions are sufficient regardless of whether or not the active instance is producing a result. They are not sufficient, however, when there are inactive instances of the current active expression.

There are two cases to consider when there are inactive instances of the current expression.

1. If the active expression has failed to produce a result, then active is changed to point to the next expression marker on the stack, and sp is changed to point to the top of the expression stack area for this new active instance.
2. If the active expression has succeeded in producing a result for some passive instance, then active is changed to point to the expression marker for that passive instance, and sp is changed to point to the top of the expression stack area for that passive instance.

Accomplishing the proper operation in both cases requires that expression markers be expanded to include two supplemental pointers. First, an inactive instance pointer is needed for resetting active during reactivation. Second, a *saved stack pointer* is needed for resetting sp to the top of the expression stack area in the enclosing passive instance.

An expression instance in the modified one-stack model has the form

passive	
inactive	
save_sp	
save_pc	
estack	

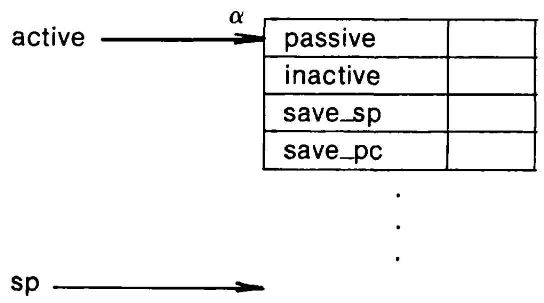
with the first four fields constituting the expression marker.

Given these changes, the primitive operations on expression instances can be rewritten assuming a conventional system stack. The assumptions are that the system stack is addressed on a word basis and that all pointers into the stack are negative offsets from the base of the stack. Hence push decrements sp and pop increments sp.

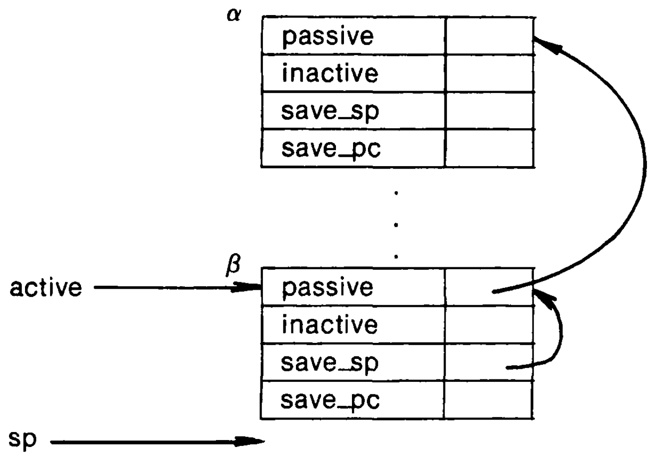
1. mark pushes a new expression marker onto the stack.

```
procedure mark(failure_lab)
local sactive

   sactive := sp
   push(active_stack, active)
   push(active_stack, &null)
   push(active_stack, sactive)
   push(active_stack, failure_lab)
   active := sactive

end
```

If before the call to mark, the stack is



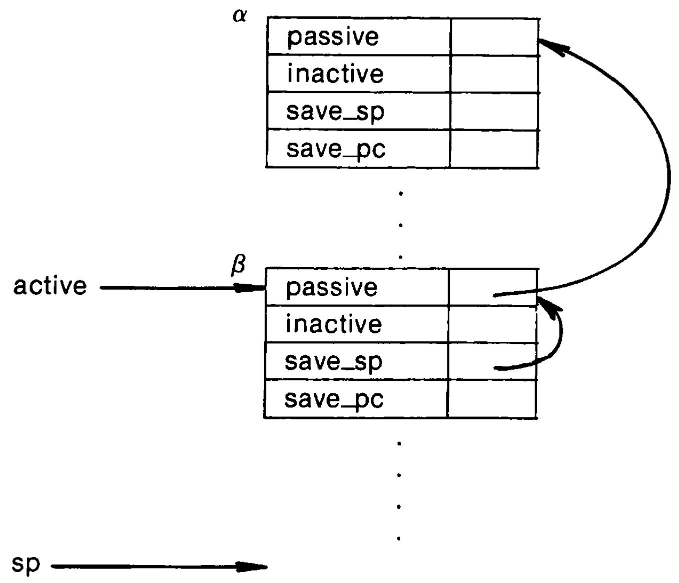
after the call the stack is



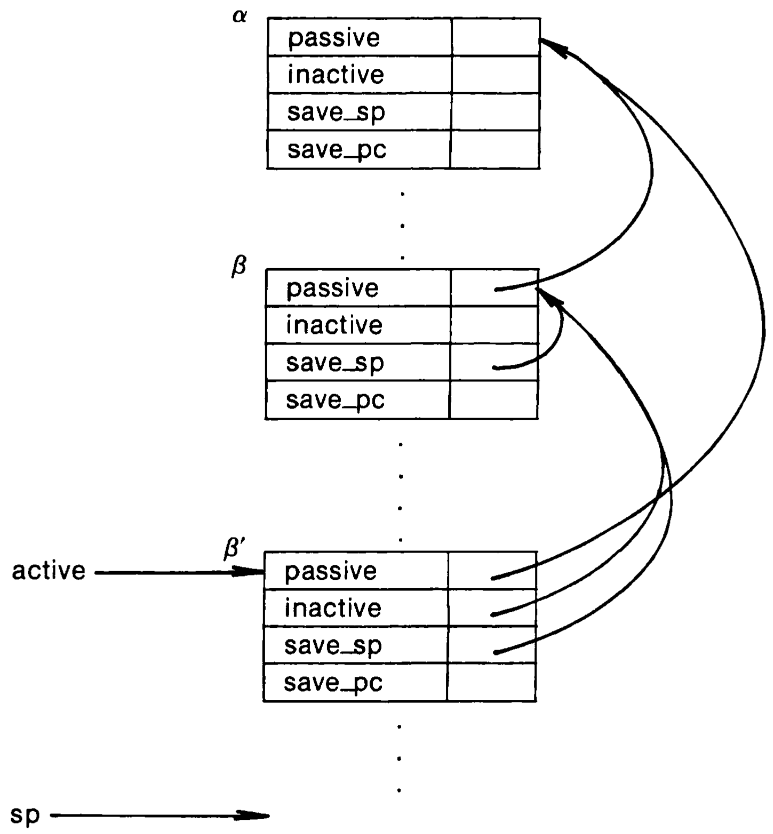
2. save must update inactive and save_sp as it "hides" the current active instance. Note that the routine copy_information is replaced by a simple every loop.

```
procedure save()
local sactive

   sactive := sp
   push(active_stack, active_stack[active])
   push(active_stack, active)
   push(active_stack, active_stack[active-2])
   push(active_stack, pc)
   every
      push(active_stack, active_stack[active-4 to sactive+1 by -1])
   active := sactive

end
```

If the stack before a call to save is



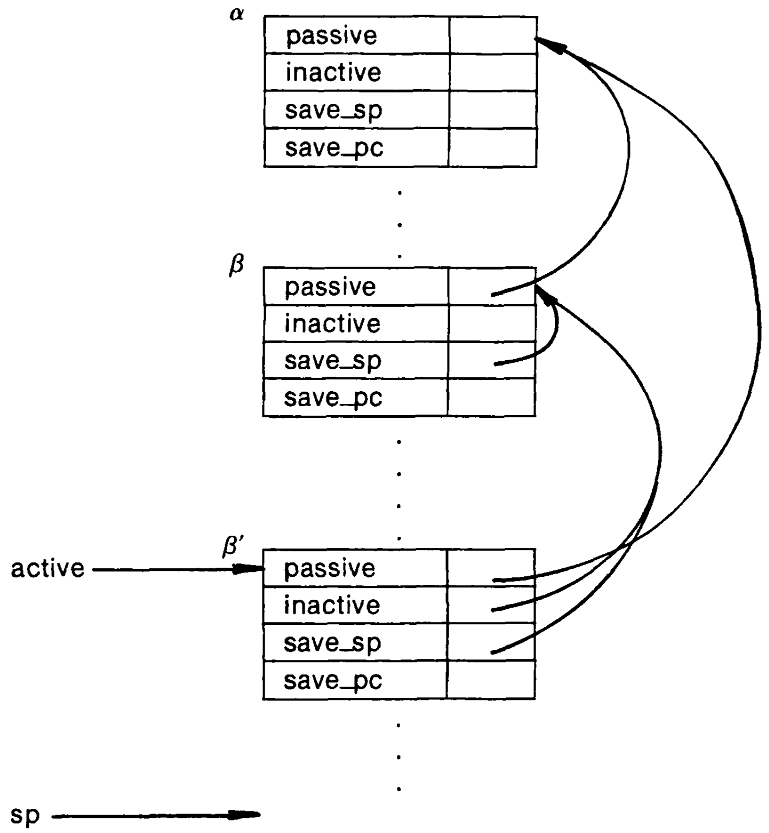
then after the call the stack is



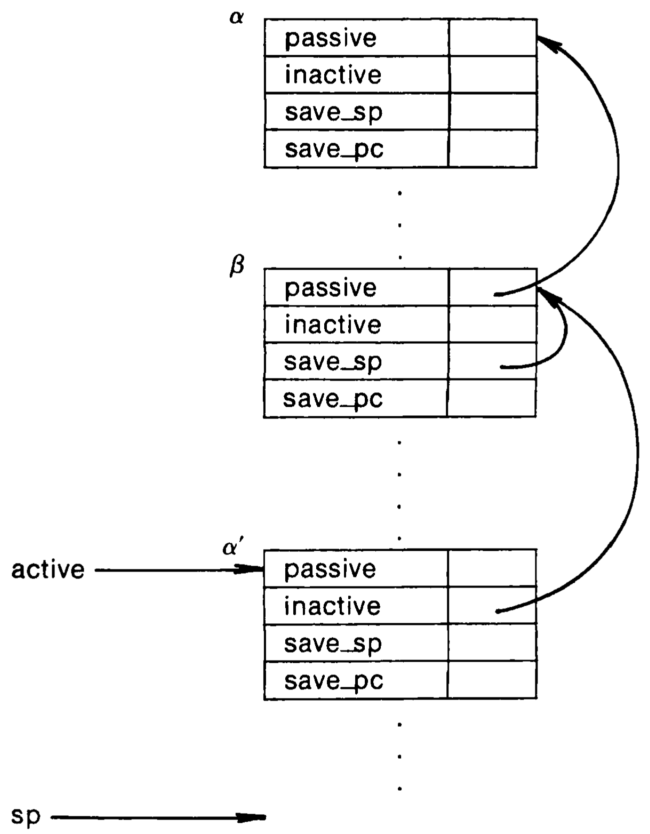
3. The changes to esave are similar to those required by save. esave must ensure that inactive points to the next expression marker on the stack.

```
procedure esave()
local passive, inactive, ssp, spc

   passive := active_stack[active]
   inactive := active_stack[active-1]
   ssp := active_stack[active-2]
   spc := active_stack[active-3]
   sp := active
   push(active_stack, active_stack[passive])
   push(active_stack, \inactive|passive)
   push(active_stack, active_stack[passive-2])
   push(active_stack, spc)
   every
      push(active_stack, active_stack[passive-4 to ssp+1 by -1])

end
```

If the stack prior to a call of esave is



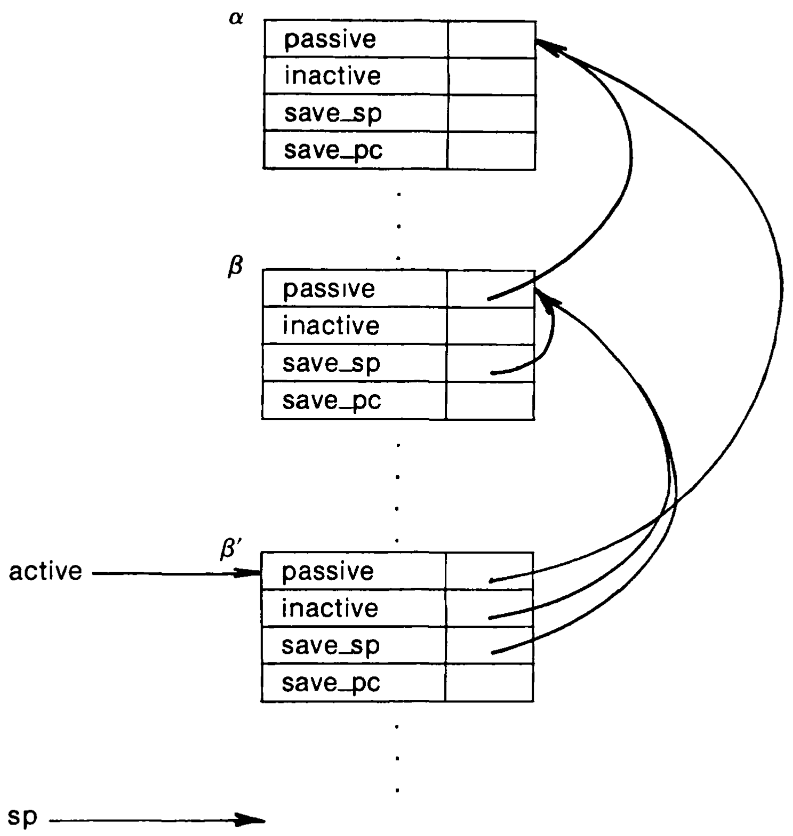
then after the call it is



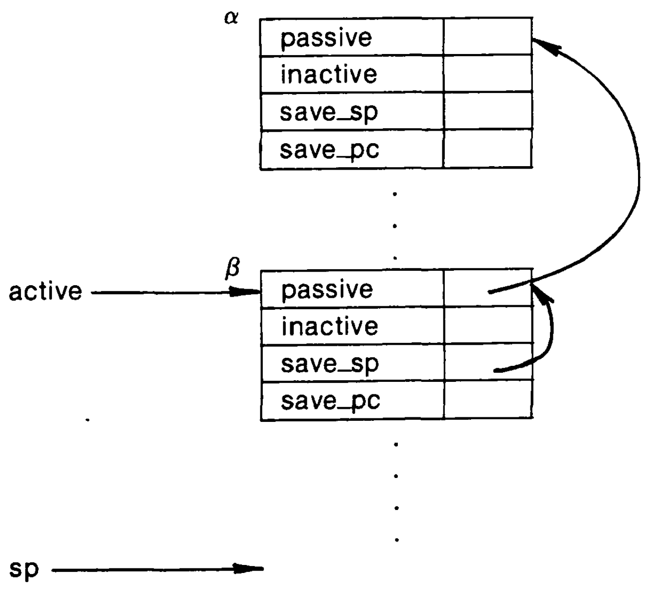
4. failure determines whether there is an inactive instance to reactivate. If there one, it is reactivated. Otherwise, failure is propagated to the enclosing passive instance.

```
procedure failure()

   pc := active_stack[active-3]
   sp := active
   if \active_stack[active-1] then
      active := active_stack[active-1]
   else
      active := active_stack[active]

end
```

If the stack before a call to failure is



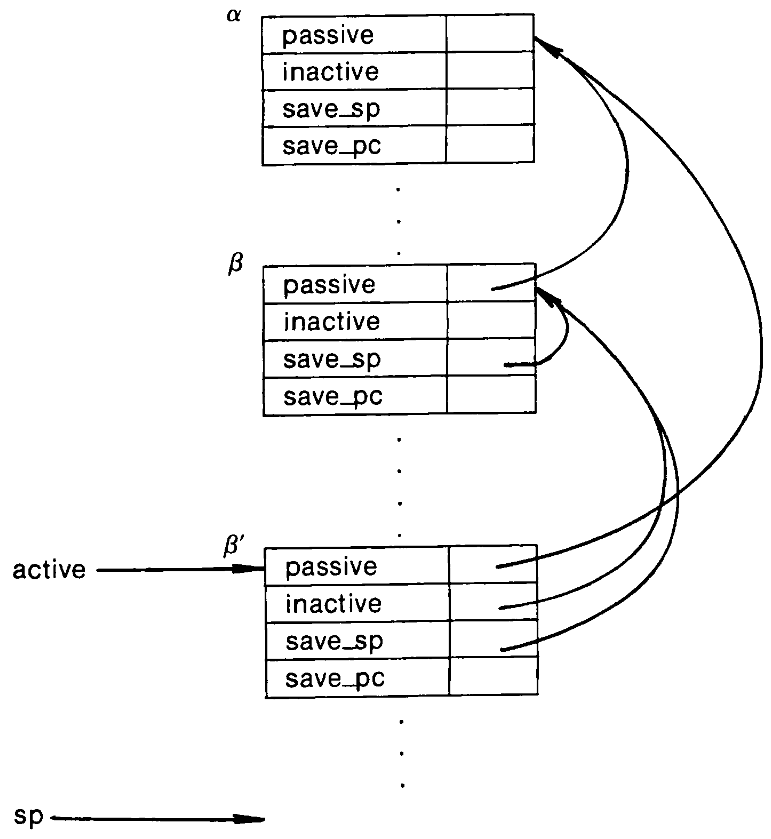
then after the call the stack is



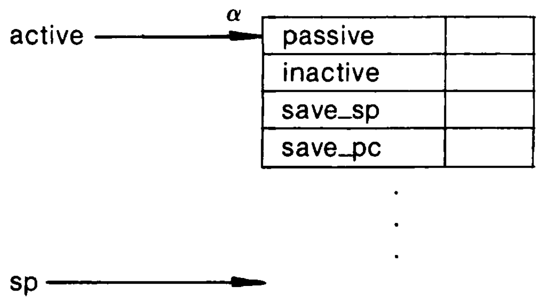
4. unmark pops all inactive instances of the active instance from the stack.

```
procedure unmark()

   sp := active_stack[active-2]
   active := active_stack[active]

end
```

If the stack before a call to unmark is



then after the call the stack is



5. lsave works as in the original one-stack model.

```
procedure lsave()
local top_passive

   top_passive := active_stack[active-2]
   if (active_stack[top_passive+1] -:= 1) > 0 then
      esave()
   else
      unmark()

end
```

6. Finally, chfail is able to access the appropriate activation address directly.

```
procedure chfail(failure_lab)
local one_above

   one_above := active_stack[active-2]
   active_stack[one_above-3] := failure_lab

end
```

These are all the changes needed to implement the one-stack model of goal-directed evaluation using a conventional system stack. However, interfacing goal-directed evaluation with other language features may require additional modifications. The most notable example in Version 4 of Icon involves storage reclamation.

The storage reclamation algorithm must locate all valid data items. To do so requires that the system stack be *tended* (searched for valid data) [8].

In the modified one-stack model, all expression stack areas contain valid data and must be tended. This is not difficult in itself; the pointers active and sp as well as the pointers in the expression markers are sufficient to locate all the expression stack areas. The problem is that in the implementation of Version 4, not all of the information within an expression stack area is necessarily valid Icon data. Inactive instances may contain information left by run-time support routines. This information must be skipped over during tending.

Fortunately, the information to be ignored is always at the top of the expression stack area, and an additional pointer can be associated with inactive instances to give the separation point between valid Icon data and information left by any run-time support routines. This pointer is called the expression area *boundary* [4].

Besides assisting in the storage reclamation process, the boundary helps distinguish functions and operators from user defined procedures. (An inactive instance with information above its boundary is an instance for evaluating a suspended function or operator. An instance with no information above its boundary is an instance for the evaluation of a procedure.) This information is useful to Icon's tracing mechanism, which must trace procedure reactivation, but not function or operator reactivation.

6.2 Co-Expressions

Co-expressions may be implemented as described previously, with the exception that active and sp must be preserved with each co-expression. The simplest solution is to push active and sp onto the co-expression stack each time that co-expression activates some other co-expression, and get the new active and sp from the top of the activated co-expression. A particular machine architecture may cause severe problems with the implementation of co-expressions. An example is in the implementation of Version 4 of Icon on the PDP-11/70 machine.

The PDP-11 does not have stack-based addressing for stack operations. Rather, pointers into the stack reference absolute memory locations within the user's data region. This makes relocation of stacks during Icon's storage reclamation process difficult, as all pointers into each stack must be tended.

Tending the pointers within the expression markers is possible, since they are known to be pointers into the stack. However, the fact that inactive instances may contain information above their boundary, and that this information may contain pointers into the stack that are unknown to the storage reclamation process makes it impossible to relocate co-expression stacks. The problem of identifying unknown pointers also makes it impossible to copy a co-expression with the Icon function copy.

In the implementation of co-expressions used in Version 4 of Icon, a fixed-sized space is allocated for each created co-expression to serve as the co-expression stack. This space is never relocated during the storage reclamation process, but is tended.

A final difficulty with co-expressions comes from their use as data objects. As a data object, the lifetime of a co-expression may exceed the lifetime of the procedure in which it is created. Variables that are local to the procedure and that are referenced within the co-expression must exist as long as the co-expression exists. In Version 4 of Icon, a co-expression is provided copies of all the current local variables when that co-expression is created. These copies are maintained with the co-expression, freeing the scope of the co-expression from the scope of the creating procedure and eliminating any problem similar to the FUNARG problem in Lisp [10]

7. Impact and Performance of the Implementation

It is difficult to measure the impact and performance of the implementations given in this report. Programs written using goal-directed evaluation or co-expressions differ greatly in style and approach from similar programs written without these language features. A few observations are possible, however.

7.1 Goal-Directed Evaluation

In situations in which there are no inactive instances, the system stack differs little in appearance from the system stacks for conventional stack-based languages. Only a few extra words (the expression marker) are added to separate expression instances. The fact that expression instances are only needed at points of program flow control helps minimize the number of these expression markers. Some of the information within the expression marker is needed only for inactive instances, and can be removed from other expression markers, reducing the number of words per expression marker.

Finally, both mark and unmark are simple operations, and can be implemented with a few machine instructions. Thus the impact on the efficiency of other language features should be slight.

It is when an expression instance becomes inactive that the two major sources of inefficiency in the performance of generators occur. First, there is the overhead involved in hiding that instance on the stack. The fact that only a portion of an expression instance needs to be copied reduces this overhead slightly. The implementation in Version 4 of Icon differs from that of Version 3 in that Version 3 copies the entire expression instance where Version 4 copies the minimum information necessary to continue processing. This change appears to have resulted in a 10-15% improvement in overall efficiency.

Second, an instance cannot suspend with a variable pointing to information contained within the expression stack area for that expression, since that area may not exist by the time the variable is referenced. Such a variable must be dereferenced when the instance suspends. Since the same situation occurs when a procedure returns a result, this problem is not endemic to generators.

Unmarking, reactivating, and propagating failure are all efficient operations amounting to little more than resetting the system stack pointer to the appropriate level. Note that reactivation in the one-stack model is thus considerably more efficient than reactivation in the two-stack model, which must copy the reactivated instance from the control stack back onto the system stack.

7.2 Co-Expressions

Co-expression creation is a fairly expensive, though relatively infrequent, operation. Space for the co-expression stack must be allocated and, at least in the case of Version 4 of Icon, the local variables for the current procedure must be copied. However, activation of a co-expression is a simple operation, accomplished in a few machine instructions. As with goal-directed evaluation, the major source of inefficiency with co-expression activation is that variables pointing to values within the activating co-expression must be dereferenced.

The impact that co-expressions have on other language features depends upon the sophistication of the underlying machine architecture. A source of the impact is in detecting stack overflow of the co-expression stack. Again, the PDP-11/70 provides a case in point.

The hardware of the 11/70 provides stack overflow checking for the primary system stack by detecting when a push operation causes the stack pointer to cross into the user's data region. However, the stack spaces for additional co-expressions lie entirely within the user's data region, and the hardware does not detect overflow on these stacks. Adequate stack overflow detection of co-expression stacks on the 11/70 requires that software checks be inserted into the code. Since it cannot be determined whether a section of code will be executed in the main system stack or another co-expression stack, these checks must be inserted throughout the code, degrading the performance of all language features. This is not a problem on machines such as the DEC-10 and Computer Automation 495, which permit the user to specify an upper bound for each stack.

8. Conclusion

Generators and co-expressions are language features capable of adding a great deal of expressiveness to programming languages. The use of expression instances as a descriptive device shows that an efficient implementation for generators is possible in a stack-based language, and that the implementation of co-expressions can be accomplished with little more than the capability to manage multiple system stacks.

The implementation of co-expressions is similar to implementation of coroutines, and can be accomplished without an implementation of generators and goal-directed evaluation. Much of the expressiveness available with co-expressions would be lost in this case, since the capability of co-expressions to free the use of generators from any lexical constraints is a powerful programming tool [11]. For example, a label generator for the code-generation phase of a compiler can be written using co-expressions and goal-directed evaluation as

```
new_label := create "L" || (0 to 9) || (0 to 9) || (0 to 9) || (0 to 9)
```

The activation of new_label could then be used wherever needed to generate labels of the form L*nnnn*. Without generators, a more conventional approach must be used.

Acknowledgement

I am indebted to Tim Budd, Cary Coutant, Ralph Griswold, and Dave Hanson for their discussions on goal-directed evaluation and co-expressions. Walt Hansen, Dave Hanson and Tim Korb developed the two-stack implementation of goal-directed evaluation. Cary Coutant is responsible for the actual implementation of the one-stack model and co-expressions on the PDP-11/70, the development of the concept of the boundary to resolve problems with interfacing to the storage reclamation process, and the elimination of the need for a failure signal to drive expression failure.

References

1. Budd, Timothy A. *An Implementation of Generators in C.* Technical Report TR 81-5, Department of Computer Science, The University of Arizona. *To appear.*
2. Coutant, Cary A., Ralph E. Griswold, and Stephen B. Wampler. *Reference Manual for the Icon Programming Language; Version 3.* Technical Report TR 80-2, Department of Computer Science, The University of Arizona. May 1980.
3. Coutant, Cary A., Ralph E. Griswold, and Stephen B. Wampler. *Reference Manual for the Icon Programming Language; Version 4.* Technical Report TR 81-4, Department of Computer Science, The University of Arizona. *To appear.*
4. Coutant, Cary A. and Stephen B. Wampler. *A Tour Through the C Implementation of Icon.* Technical Report TR 80-9, Department of Computer Science, The University of Arizona. June 1980.
5. Griswold, Ralph E. *Expression Evaluation in Icon.* Technical Report TR 80-21, Department of Computer Science, The University of Arizona. August 1980.
6. Griswold, Ralph E., David R. Hanson, and John T. Korb. "The Icon Programming Language; An Overview", *SIGPLAN Notices*, Vol. 14, No. 4 (April 1979). pp. 18-31.
7. Griswold, Ralph E., David R. Hanson, and John T. Korb. "Generators in Icon", *ACM Transactions on Programming Languages and Systems*, Vol. 3, No. 2 (April 1981). pp. 144-161.
8. Hanson, David R. "Storage Management for an Implementation of SNOBOL4." *Software—Practice and Experience* Vol. 7, No. 2 (March 1977). pp. 179-192.
9. 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.
10. Moses, J. "The Function of FUNCTION in LISP", *SIGSAM Bulletin, No. 4 (July 1970). pp. 13-27.*
11. Wampler, Stephen B. *New Control Structures in Icon.* Technical Report TR 81-1, Department of Computer Science, The University of Arizona. January 1981.
12. Wampler, Stephen B. *Sequences and Expression Evaluation in Icon.* Technical Report TR 81-2, Department of Computer Science, The University of Arizona. March 1981.