Temporary Variable Allocation in the Presence of Goal-Directed Evaluation

Kenneth Walker

Department of Computer Science, The University of Arizona

1. Introduction

The maintenance of intermediate results during expression evaluation in the Icon programming language [1] is more complicated than it is for conventional languages, such as C and Pascal. Janalee O'Bagy explains this in her dissertation [2]:

> "Generators prolong the lifetime of temporary values. For example, in
>
> ```
> i = find(s1,s2)
> ```
>
> the operands of the comparison operation cannot be discarded when find produces its result. If find is resumed, the comparison is performed again with subsequent results from find(s1,s2), and the left operand must still be available."

Two basic approaches have been used in language implementations to store intermediate results. On a stack-based machine, operations implicitly use the stack to store and retrieve intermediate results. Operations are executed in such a way that only the top elements of the stack need be pushed and popped. Expressions in conventional languages translate naturally into code for a stack-based machine. This not true of Icon. The code in a stack-based implementation of Icon is forced to make a copy of part of the stack at critical points in execution, in anticipation of possible backtracking later on [2, 3]. In spite of this, earlier implementations of Icon, with the exception of a partial implementation by Thomas Christopher [4], use a stack-based virtual machine.

The alternate approach to language implementation is to use a "register-based" machine. In this implementation model, intermediate results are stored in explicit locations. These locations may or may not be actual machine registers. In any case, they may be viewed as *temporary variables*. A translator typically uses one temporary variable for several intermediate results. A *correct* allocation of temporaries to intermediate results insures that a live result (that is, one that might still be needed) is not overwritten by another result.

Temporary variable allocation consists of two problems: determining the lifetime of intermediate results, that is liveness analysis, and determining a correct and efficient mapping of intermediate results to temporary variables based on this liveness information. In a straightforward implementation of conventional languages, liveness analysis is trivial: an intermediate result is computed in one place in the generated code, is used in one place, and is live in the contiguous region between the computation and the use. In such languages, determining the lifetime of intermediate results only becomes complicated when certain optimizations are performed, such as common subexpression elimination across basic blocks and code motion [5, 6]. However, this is not true in Icon. In the presence of goal-directed evaluation, the lifetime of an intermediate result can extend beyond the point of use. Even in a straightforward implementation, liveness analysis is not trivial.

Icon programs contain many places where goal-directed evaluation is bounded; Christopher uses this fact to deduce bounds on the lifetimes of intermediate results in his temporary variable implementation of Icon. However, this approach produces a very crude estimate of these lifetimes. This report addresses the problem of "fine-

grained'' liveness analysis in the presence of goal-directed evaluation. In its most general form, liveness analysis requires iterative methods to solve. However, goal-directed evaluation imposes enough structure on the liveness problem that, at least in the absence of optimizations, iterative methods are not needed to solve it. This report presents a simple and accurate method for computing liveness information for intermediate results in Icon. The analysis is formalized in an attribute grammar.

The liveness information is used by a ''register'' allocator to make efficient use of temporary variables. How this allocation is done depends on details of the machine model, implementation conventions, and decisions of how much work is reasonable to obtain good reuse of temporary variables. This report presents a brief discussion of these allocation issues.

2. Motivation

As noted in the introduction, stack-based implementations of Icon must copy portions of the stack during program execution. To illustrate the amount of copying done, an Icon interpreter was instrumented to count the number of bytes copied and run on an Icon program that makes a moderate use of generators (this program is a prototype type inference system and was run with an Icon solution to the eight queens problems as its input). The interpreter copied 750,960 bytes of stack while executing 754,914 icode instructions (see [3] for a description of icode instructions). This copying is a small part of the cost of instruction execution in the interpreter. However, an optimizing compiler can significantly reduce other costs in program execution, increasing the significance of this copying. Therefore, a better method for handling intermediate results is desirable in such a compiler. Another motivation for changing implementation models is that a stack-based implementation limits the amount of code optimization that can be performed.

Using a model that employs explicit temporary variables can eliminate the need to make copies of intermediate results. This is because operations in such a model do not implicitly consume their operands. If a potentially infinite supply of temporary variables is available and the number used has no impact on the cost of execution, then it is reasonable to allocate one for each intermediate result. However, in realistic implementations it is desirable to reuse temporary variables where possible.

Just as goal-directed evaluation complicates the use of a stack-based model of execution, it also complicates determining the length of time that an intermediate value must be retained in a temporary variable. This is because goal-directed evaluation creates implicit loops within an expression. In O'Bagy's example in the introduction, the start of the loop is the generator find and the end of the loop is the comparison that may fail. An intermediate result may be used within such a loop, but if its value is computed before the loop is entered, it is not recomputed on each iteration and the temporary variable must not be reused until the loop is exited.

The following fragment of Icon code contains a loop and is therefore analogous to code generated for goal-directed evaluation. *t1* through *t4* represent intermediate results that must be allocated to program variables.

```
t1 := &time
every 1 to 3 do {
   t2 := read()
   t3 := t1 + t2
   write(t3)
   }
t4 := 8
```

Separate variables must be allocated for *t1* and *t2* because they are both needed for the addition. Here, x is chosen for *t1* and y is chosen for *t2*.

```
x := &time
every 1 to 3 do {
   y := read()
   t3 := x + y
   write(t3)
   }
t4 := 8
```

x cannot be used to hold *t3*, because x is needed in subsequent iterations of the loop. Its lifetime must extend through the end of the loop. However, y can be used because it is recomputed in subsequent iterations. Either variable may be used to hold *t4*.

```
x := &time
every 1 to 3 do {
   y := read()
   y := x + y
   write(y)
   }
x := 8
```

Before temporary variables can be allocated, the extent of the loops created by goal-directed evaluation must be estimated. Suppose O'Bagy's example

```
i = find(s1, s2)
```

appears in the following context

```
procedure p(s1, s2, i)
   if i = find(s1, s2) then return i + *s1
   fail
end
```

The simplest and most pessimistic analysis assumes that a loop can appear anywhere within the procedure, requiring the conclusion that an intermediate result in the expression may live to the end of the procedure. Christopher's analysis notices that the expression appears within the control clause of an if expression. This is a bounded context; implicit loops cannot extend beyond the end of the control clause. His allocation scheme reuses, in subsequent expressions, temporary variables used in this control clause. However, it does not determine when temporary variables can be reused within the control clause itself.

The analysis presented here locates within the expression the operations that can fail and those that can generate results. It uses this information to accurately determine the loops within the expression and the intermediate results whose lifetimes are extended by those loops.

3. Liveness Analysis

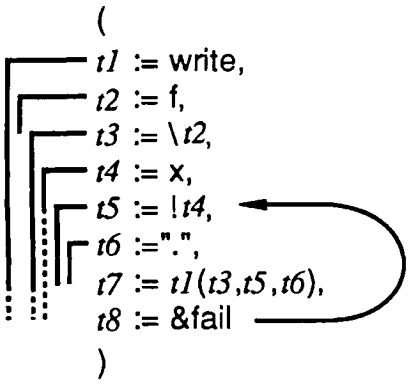
It is instructive to look at a specific example where intermediate values must be retained beyond (in a lexical sense) the point of their use. The following expression employs goal-directed evaluation to conditionally write sentences in the data structure x to an output file. If f is a file, the sentences are written to it; if f is null, the sentences are not written.

```
every write(\f, !x, ".")
```

In order to avoid control structures at this point in the discussion, the following equivalent expression is used in the analysis:

```
write(\f, !x, ".") & &fail
```

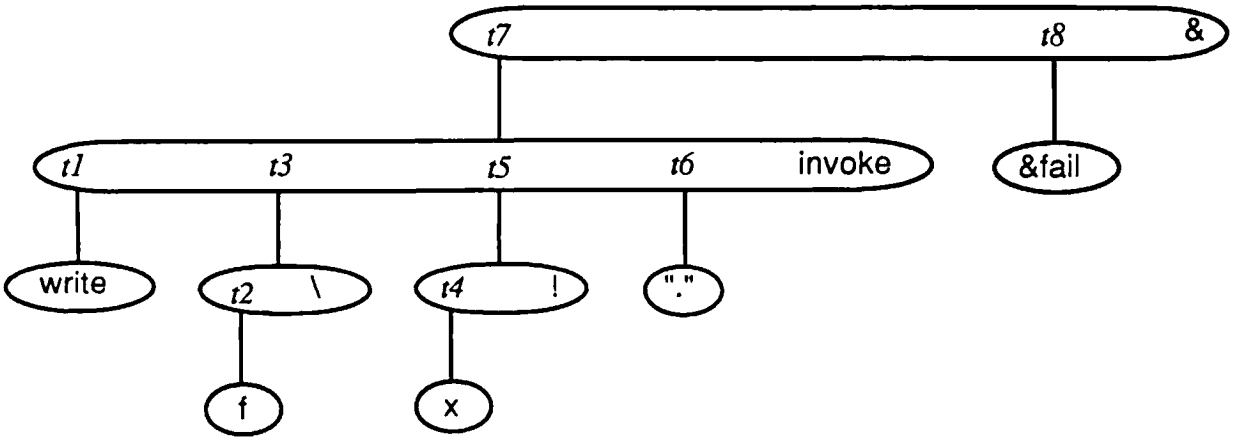
This expression can be converted into a sequence of primitive operations producing intermediate results (*t1*, *t2*, ...). For convenience, the operations are expressed in Icon:



Whether or not the program variables and constants are actually placed in temporary variables depends the machine model and implementation conventions, and clearly a temporary variable is not needed for &fail. However, temporary variables are needed if the subexpressions are more complex; intermediate results are shown for all subexpressions for explanatory purposes. (Note that Icon does not quite capture the semantics of the primitive operations; technically, *t1*, *t2*, and *t4* should be variable references, not dereferenced values.)

When the &fail is executed, the ! operation is resumed. This creates an implicit loop from the ! to the &fail, as shown by the arrow in the above diagram. The question is: What intermediate results must be retained up to the &fail? A more instructive way to phrase the question is: After &fail is executed, what intermediate values might be reused without being recomputed? From the sequence of primitive operations, the answer is clearly *t1*, *t3*, and *t4*. *t2* is not used within the loop, *t5* and *t6* are recomputed within the loop, and *t7* and *t8* are not used. The lines in the diagram to the left of the code indicate the lifetime of the intermediate results. The dotted portion of each line represents the region of the lifetime extending beyond the result's use.

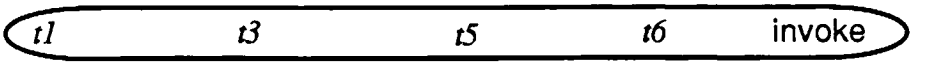
A notation that emphasizes intermediate results, subexpressions, and execution order is helpful for understanding how liveness is computed. Both postfix notation and syntax trees are inadequate. A postfix notation is good for showing execution order, but tends to obscure subexpressions. The syntax tree of an expression shows subexpressions, but execution order must be expressed in terms of a tree walk. In both representations, intermediate results are implicit. For this discussion, an intermediate representation is used. A subexpression is represented as a list of explicit intermediate results followed by the operation that uses them, all enclosed in ovals. Below each intermediate result is the subexpression that computes it. This representation is referred to as a *postfix tree*. The example above is shown as



In this notation, the execution order of operations (which includes constants and references to program variables) is left-to-right and the backtracking order is right-to-left. In this example, the backtracking order is &fail, invoke, ".", !, x, \, f, and write.

As explained above, the use of an intermediate value must appear in an implicit loop for the value to have an extended lifetime. Two events are needed to create such a loop. First, an operation must fail, initiating backtracking. Second, an operation must be resumed, causing execution to proceed forward again. This analysis computes the maximum lifetime of intermediate results in the expression, so it only needs to compute the rightmost operation (within a bounded expression) that can fail. This represents the end of the farthest reaching loop. Once execution proceeds beyond this point, no intermediate result can be reused.

The intermediate values of a subexpression are used at the end of the subexpression. For example, invoke uses the intermediate values in



In order for this use to be in a loop, backtracking must be initiated from outside; that is, beyond the subexpression (in the example, only &fail and & are beyond the subexpression).

In addition, for an intermediate value to have an extended lifetime, the beginning of the loop must start after the intermediate value is computed. Two conditions may create the beginning of a loop. First, the operation itself may be resumed. In this case, execution continues forward within the operation. It may reuse any of its operands and none of them are recomputed. (Whether an operation actually reuses its operands on resumption depends on its implementation, but it is conservative to assume that it does.) The operation does not have to actually generate more results. For example, reversible swap can be resumed to reuse both of its operands, but it does not generate another result.

The second way to create the beginning of a loop is for a subexpression to generate results. Execution continues forward again and any intermediate results to the left of the generative subexpression may be reused without being recomputed. Remember, backtracking is initiated from outside the expression. Suppose an expression that can fail is associated with *t6*. This creates a loop.with the generator associated with *t5*. However, this particular loop does not include invoke and does not contribute to the reuse of *t1* or *t3*.

A resumable operation and generative subexpressions are all *resumption points* within an expression. A simple rule determines which intermediate results of an expressions have extended lifetimes: If the expression can be resumed, the intermediate values with extended lifetimes consist of those to the left of the rightmost resumption point of the expression. This rule refers to the "top level" intermediate results. The rule must be applied recursively to subexpressions to determine the lifetime of lower level intermediate results.

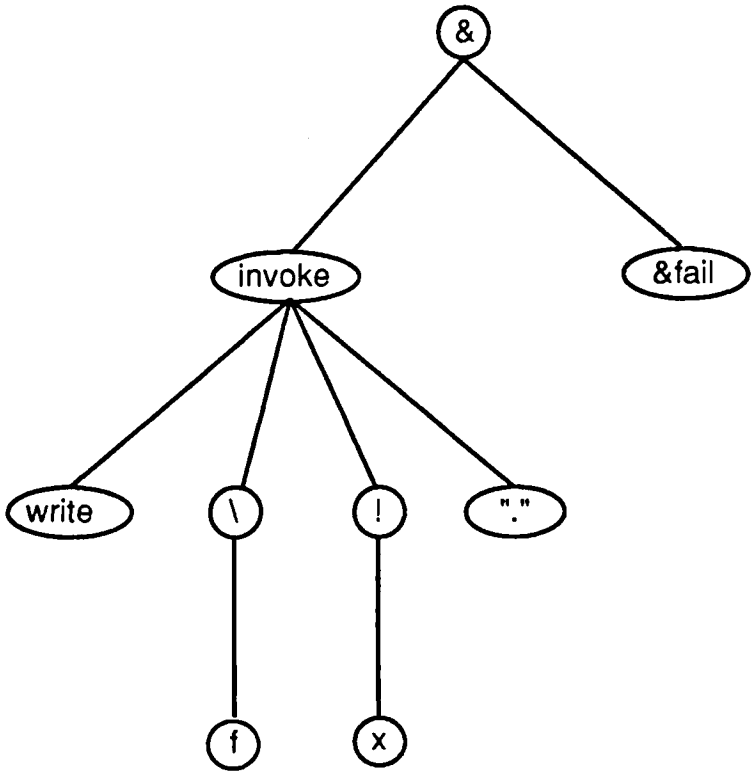
It sometimes may be necessary to make conservative estimates of what can fail and of resumption points (for liveness analysis, it is conservative to overestimate what can fail or be resumed). For example, invocation may or may not be resumable, depending on what is being invoked and, in general, it is not known until run time what is being invoked (for the purposes of this example analysis, it is assumed that the variable write is not changed anywhere in the program).

In the example, the rightmost operation that can fail is &fail. Resumption points are ! and the subexpressions corresponding to the intermediate values *t5* and *t7*.

Once the resumption points have been identified, the rule for determining extended lifetimes can be applied. If there are no resumption points in an expression, no intermediate values in that expression can be reused. Applying this rule to the postfix tree above yields *t1*, *t3*, and *t4* as the intermediate values that have extended lifetimes.

4. An Attribute Grammar

To cast this approach as an attribute grammar, an expression should be thought of in terms of an abstract syntax tree. The transformation from a postfix tree to a syntax tree is trivial. It is accomplished by deleting the explicit intermediate results. A syntax tree for the example is



Several interpretations can be given to a node in a syntax tree. A node can be viewed as representing either an operation, an entire subexpression, or an intermediate result.

This analysis associates four attributes with each node (this ignores attributes needed to handle break expressions). The goal of the analysis is to produce the lifetime attribute. The other three attributes are used to propagate information needed to compute the lifetime.

- resumer is either the rightmost operation (represented as a node) that can initiate backtracking into the subexpression or it is null if the subexpression cannot be resumed.
- failer is related to resumer. It is the rightmost operation that can initiate backtracking that can continue past the subexpression. It is the same as resumer, unless the subexpression itself contains the rightmost operation that can fail.
- gen is a boolean attribute. It is true if the subexpression can generate multiple results if resumed.
- lifetime is the operation beyond which the intermediate value is no longer needed. It is either the parent node, the resumer of the parent node, or null. The lifetime is the parent node if the value is never reused after the parent operation completes. The lifetime is the resumer of the parent if the parent operation or a generative sibling to the right can be resumed. A lifetime of null is used to indicate that the intermediate value is never used. For example, the value of the control clause of an if expression is never used.

Attribute computations are associated with productions in the grammar. The attribute computations for failer and gen are always for the non-terminal on the left-hand side of the production. These values are then used at the parent production; they are effectively passed up the syntax tree. The computations for resumer and lifetime are always for the attributes of non-terminals on the right-hand side of the production. resumer is then used at the production defining these non-terminals; it is effectively passed down the syntax tree. lifetime is usually saved just for the code generator, but it is sometimes be used by child nodes.

Primary Expressions

Variables, literals, and keywords are primary expressions. They have no subexpressions, so their productions contain no computations for resumer or lifetime. The following are the attribute computations for a literal. A literal itself cannot fail, so backtracking only passes beyond it if the backtracking was initiated before (to the right of) it. A literal cannot generate multiple results.

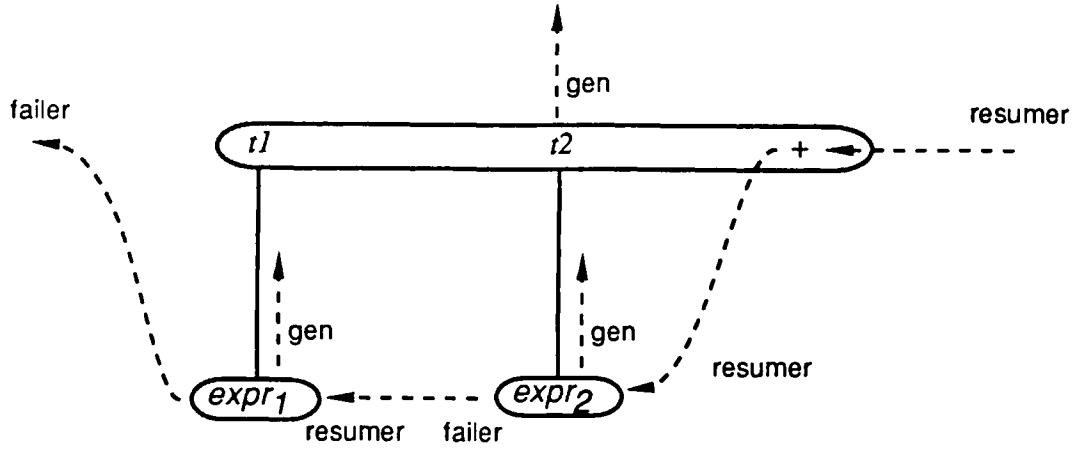
```
expr ::= literal {
        expr.failer := expr.resumer
        expr.gen := false
        }
```

Another example of a primary expression is the keyword &fail. Execution cannot continue past &fail, so it must be the rightmost operation within its bounded expression that can fail. A pre-existing attribute, node, is assumed to exist for every symbol in the grammar. It is the node in the syntax tree that corresponds to the symbol.

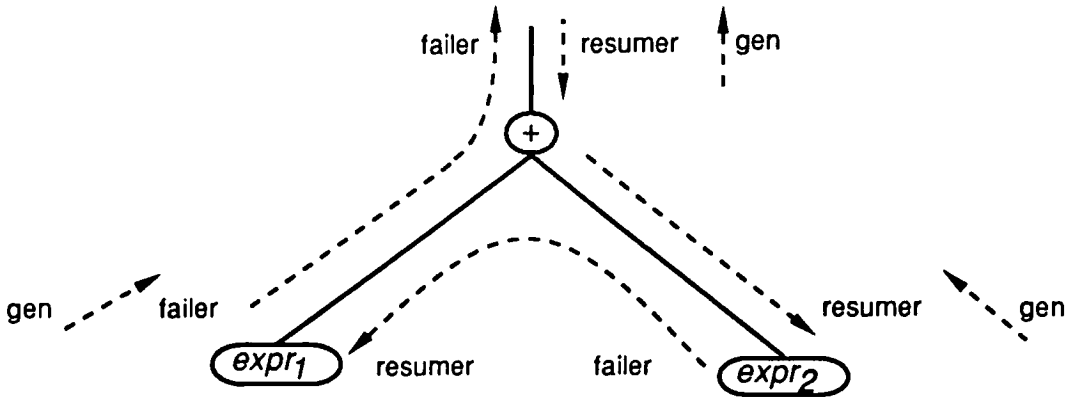
```
expr ::= &fail {
        expr.failer := expr.node
        expr.gen := false
        }
```

Operations with Subexpressions

Addition provides an example of the attribute computations involving subexpressions. The following diagram shows how resumer, failer, and gen information would be passed through the postfix tree.



This information would then be used to compute lifetime information for *t1* and *t2*. The next diagram shows how the attribute information is actually passed through the syntax tree.



The lifetime attributes are computed for the roots of the subtrees for $expr_1$ and $expr_2$.

The details of the attribute computations depend, in part, on the characteristics of the individual operation. Addition does not fail, so the rightmost resumer, if there is one, of $expr_2$ is the rightmost resumer of the entire expression. The rightmost resumer of $expr_1$ is the rightmost operation that can initiate backtracking that continues past $expr_2$. Addition does not suspend, so the lifetime of the value produced by $expr_2$ only extends through the operation (that is, it always is recomputed in the presence of goal-directed evaluation). If $expr_2$ is a generator, then the result of $expr_1$ must be retained for as long as $expr_2$ might be resumed. Otherwise, it need only be retained until the addition is performed. $expr_1$ is the first thing executed in the expression, so its failer is the failer for the entire expression. The expression is a generator if either $expr_1$ or $expr_2$ is a generator (note that the operation | is logical *or*, not Icon's alternation control structure):

```
expr ::= expr1 + expr2 {
        expr2.resumer := expr.resumer
        expr2.lifetime := expr.node
        expr1.resumer := expr2.failer
        if expr2.gen & expr.resumer ≠ null then
           expr1.lifetime := expr.resumer
        else
           expr1.lifetime := expr.node
        expr.failer := expr1.failer
        expr.gen := (expr1.gen | expr2.gen)
        }
```

/expr provides an example of an operation that can fail. If there is no rightmost resumer of the entire expression, it is the rightmost resumer of the operand. The lifetime of the operand is simply the operation, by the same argument used for $expr_2$ of addition. The computation of failer is also analogous to that of addition. The expression is a generator if the operand is a generator:

```
expr ::= /expr1 {
        if expr.resumer = null then
           expr1.resumer := expr.node
        else
           expr1.resumer := expr.resumer
        expr1.lifetime := expr.node
        expr.failer := expr1.failer
        expr.gen := expr1.gen
        }
```

!expr differs from /expr in that it can generate multiple results. If it can be resumed, the result of the operand must be retained through the rightmost resumer:

```
expr ::= !expr1 {
        if expr.resumer = null then {
           expr1.resumer := expr.node
           expr1.lifetime := expr.node
           }
        else {
           expr1.resumer := expr.resumer
           expr1.lifetime := expr.resumer
           }
        expr.failer := expr1.failer
        expr.gen := true
        }
```

Control Structures

Other operations follow the general pattern of the ones presented above. However, control structures require unique attribute computations. In particular, several control structures bound subexpressions, limiting backtracking. For example, not bounds its argument and discards the value. If it has no resumer, then it is the rightmost operation that can fail. The expression is not a generator:

```
expr ::= not expr1 {
        expr1.resumer := null
        expr1.lifetime := null
        if expr.resumer = null then
           expr.failer :=  expr.node
        else
           expr.failer :=  expr.resumer
        expr.gen := false
        }
```

$expr_1$; $expr_2$ bounds $expr_1$ and discards the result. Because the result of $expr_1$ is the result of the entire expression, the code generator makes their result locations synonymous. This is reflected in the lifetime computations. Indeed, all the attributes of $expr_2$ and those of the expression as a whole are the same:

```
expr ::= expr1 ; expr2 {
        expr1.resumer := null
        expr1.lifetime := null
        expr2.resumer := expr.resumer
        expr2.lifetime := expr.lifetime
        expr.failer := expr2.failer
        expr.gen := expr2.gen
        }
```

A reasonable implementation of alternation places the result of each subexpression into the same location: the location associated with the expression as a whole. This is reflected in the lifetime computations. The resumer of the entire expression is also the resumer of each subexpression. Backtracking out of the entire expression occurs when backtracking out of $expr_2$ occurs. This expression is a generator:

```
expr ::= expr1 | expr2 {
        expr2.resumer:= expr.resumer
        expr2.lifetime := expr.lifetime
        expr1.resumer := expr.resumer
        expr1.lifetime := expr.lifetime
        expr.failer := expr2.failer
        expr.gen := true
        }
```

The first operand of an if expression is bounded and its result is discarded. The other two operands are treated similar to those of alternation. Because there are two independent execution paths, the rightmost resumer may not be well-defined. However, it is always conservative to treat the resumer as if it is farther right than it really is; this just means that an intermediate value is kept around longer than needed. If there is no resumer beyond the if expression, but at least one of the branches can fail, the failure is treated as if it came from the end of the if expression (represented by the node for the expression). Because backtracking out of an if expression is rare, this inaccuracy is of no practical consequence. The if expression is a generator if either branch is a generator:

```
expr ::= if expr1 then expr2 else expr3 {
        expr3.resumer := expr.resumer
        expr3.lifetime := expr.lifetime
        expr2.resumer := expr.resumer
        expr2.lifetime := expr.lifetime
        expr1.resumer := null
        expr1.lifetime := null
        if expr.resumer = null & (expr1.failer ≠ null | expr2.failer ≠ null) then
           expr.failer := expr.node
        else
           expr.failer = expr.resumer
        expr.gen := (expr2.gen | expr3.gen)
        }
```

The do clause of every is bounded and its result discarded. The control clause is always resumed at the end of the loop and can never be resumed by anything else. The value of the control clause is discarded. Ignoring break expressions, the loop always fails:

```
expr ::= every expr1 do expr2 {
        expr2.resumer := null
        expr2.lifetime := null
        expr1.resumer := expr.node
        expr1.lifetime := null
        expr.failer := expr.node
        expr.gen := false
```

Handling break expressions requires some stack-like attributes. These are similar to the ones described in [2, 7].

The attributes presented here can be computed with one walk of the syntax tree. At a node, subtrees are processed in reverse execution order: first the resumer and lifetime attributes of a subtree are computed, then the subtree is walked. Next the failer and gen attributes for the node itself are computed, and the walk moves back up to the parent node.

5. Using Liveness Information

An allocation of intermediate results to temporary variables must meet several constraints, some of which may be dictated by the implementation model. The implementation model includes such things as the memory available along with its usage conventions. It also includes primitive instructions and invocation conventions. To avoid the complexities of classical register allocation, which involves allocating from a fixed finite pool of "cheap" memory locations along with an unbounded pool of "expensive" memory locations, this discussion assumes an unbounded

pool of uniform cost. This can be implemented by placing the temporary variables for each procedure in its procedure frame.

The most fundamental allocation constraint is that two intermeditate results may not be assigned to the same temporary variable if there is some program point at which they are both live. Another constraint is that, as noted above, operations such as alternation should have the results of subexpressions placed in the same temporary variable. An example of a model-dependent constraint is the requirement that the arguments of some operations be stored in consecutive locations.

A simple intuitive approach to temporary variable allocation satisfies the first constraint. This is accomplished by one pass over the program in execution order. At each point, an intermediate result is produced, an unused temporary variable is allocated for the result and marked *in use*. When the end of the lifetime of an intermediate result is reached, the corresponding temporary variable is marked *free*. The works because a value is live during a contiguous region of the program and the liveness calculations insure that there is a unique end-of-lifetime.

However, the other two constraints demonstrate that variables for different intermediate results cannot always be allocated independently. These constraints can be met if temporary variables are allocated for all operands when an expression is encountered. This has the undesirable effect of tying up temporary variables before they actually contain values. This problem can be reduced by marking the variable *reserved* at the start of the expression and marking it *in use* when the operand has used it. A subexpression can allocate a variable that is already marked *reserved*, as long as its intermediate result does not live beyond the original need for the variable. As this original need must occur after the current subexpression is completed, it is conservative to only assign reserved variables to intermediate results that do not outlive the subexpression.

There is an important difference between the stack model with copying and the temporary variable model. In the stack model, locations containing operands can be used as work areas (for dereferencing, type conversions, etc). This is because the original values have been saved if they might need to be reused. In the temporary variable model, values are not saved. Separate work locations are sometimes be needed. These locations may be allocated using the same mechanism used for intermediate results or another mechanism may be used.

The fact that dereferencing and type conversion cannot necessarily be performed in place, potentially undermines the ability of the temporary variable model to eliminate the copying of intermediate results. If one of these conversions actually does work, for instance converting a string to an integer, then it is no more expensive to compute the result into a work area than it is to compute it in place. However, if it turns out that no conversion is actually needed, then the work area conversion must still copy the value, while the in-place conversion is finished as soon as it completes the type check. To minimize this copying, liveness information should be used to determine when in-place conversions can be safely generated. Copying can be further reduced by eliminating unnecessary conversions. This can be done with information produced by type inference [7].

6. Conclusions

A stack-based model of expression evaluation is not natural for Icon with its goal-directed evaluation. A model employing explicitly temporary variables can eliminate the large amount of copying of intermediate results that is inherent in a stack model. The potential drawback of the temporary variable model is the necessity of determining, at compile time, the lifetimes of intermediate results. This report presents an effective and efficient method of computing those lifetimes.

If classical register allocation is not done, a simple algorithm can be used to allocate temporary variables. However, if the goal of reducing data movement is to be realized, care must be taken to avoid unnecessary copying of intermediate results into work areas for dereferencing and other conversions.

Acknowledgements

Ralph Griswold served as the research advisor for this project. Kelvin Nilsen applied the liveness analysis presented in this report to an implementation model slightly different from the one it was developed for, providing insight into dependencies on execution models. Ralph Griswold, Kelvin Nilsen, and Janalee O'Bagy reviewed this report, providing helpful suggestion on the material in the report and its presentation.

References

1. R. E. Griswold and M. T. Griswold, *The Icon Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, second edition, 1990.
2. J. O'Bagy, *The Implementation of Generators and Goal-Directed Evaluation in Icon*, The Univ. of Arizona Tech. Rep. 88-31, 1988.
3. R. E. Griswold and M. T. Griswold, *The Implementation of the Icon Programming Language*, Princeton University Press, 1986.
4. T. W. Christopher, *Efficient Evaluation of Expressions in Icon*, Unpublished Draft, Illinois Institute of Technology, 1985.
5. A. V. Aho, R. Sethi and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison Wesley, Reading, MA, 1985.
6. S. S. Muchnick and N. D. Jones, *Program Flow Analysis: Theory and Applications*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1981.
7. K. Walker, *A Type Inference System for Icon*, The Univ. of Arizona Tech. Rep. 88-25, 1988.