The Icon Analyst

In-Depth Coverage of the Icon Programming Language

December 1991
Number 9

In this issue …

Bogus Expressions

When programmers are first learning a new programming language, they are likely to get both syntax and concepts wrong at times. This leads to what we sometimes call "bogus" expressions. The term bogus has taken on a rather broader meaning in hacker patois than you'll find in the average dictionary, but it has a nice ring. So do the derived terms "bogosities" and "bogons".

Since Icon has a richer expression-evaluation mechanism than most programming languages, it offers more opportunities for bogosities, which may strike more experienced programmers, if not their novice authors, as amusing. Here are some examples we've seen.

1. Inside-Out Expressions

This kind of thing turns up quite frequently:

```
write(every 1 to 10)
```

It's syntactically correct, but it doesn't "do" anything.

The argument of write() is an every expression. The every expression forces its argument to generate 1 through 10. But there's nothing to use these values and after the last one is produced, every fails, as all loops do when they terminate (unless by break). Since the argument of write() fails, nothing is written.

What the author of this inside-out expression presumably intended was

```
every write(1 to 10)
```

The argument that the first version "has both an every and a write()" doesn't quite get to root of the problem.

2. Confused Returns

We've probably seen more confusing usages in returns from procedure calls than in any other category of expressions. Here's one:

```
return fail
```

This one does what its author probably intended — it causes the procedure call in which it occurs to fail. But it's bogus on the face of it.

A return expression evaluates its argument. If the argument produces a value, that value is returned. If the argument fails, the procedure call fails — in other words, return produces the *outcome* of its argument. In the case above, the evaluation of fail causes the function call to fail before the evaluation of return completes.

Another example of confusion in returning is

```
return suspend
```

which presumably doesn't do what its author intended.

An omitted argument to suspend is like an omitted argument in a function call — it's equivalent to &null. In the expression above, the procedure suspends with the null value (in the middle of evaluating return). If it's resumed, the suspend expression (like every) fails. Since it's the argument of return, the procedure call then fails.

Here's another one:

```
suspend &fail
```

This expression does not suspend nor does it cause the procedure call to fail — it does nothing. A suspend expression suspends with the results generated by its argument. If its argument doesn't produce a result (as in the case here), it doesn't suspend at all, and program execution continues with the next expression after the suspend — a confusing "no-op".

3. Redundancies

Here's another kind of expression we see fairly often:

```
every suspend 1 to 10
```

The author of this expression apparently didn't know that suspend suspends with every result produced by its argument.

In fact, suspend is just like every except for the fact that it suspends with every result produced by its argument. And suspend, like every, fails when there are no more results from its argument. So, in the case above, the argument to every fails.

We've heard novice programmers say "But I just wanted to make sure." Sure.

Here's another misguided use of every:

```
every if s == !x then …
```

What was intended here presumably was to be sure !x produced all its values. It produces as many as needed as the result of goal-directed evaluation. every has nothing to do with that.

Since the expression above evaluates the if-then expression the same way, whether or not the every is hanging out front (waiting for the if-then expression to do something), the author may say, "Well, it worked", which is more than can be said for this monstrosity:

```
if every (s == !x) then …
```

We'll end with an fairly common usage that isn't wrong; it's just unnecessarily complex:

```
every s := !x do
   suspend s
```

All that's needed is

```
suspend !x
```

If you've encountered bogus or silly expressions in Icon, let us know. If we get enough more, we'll list them in a future issue of the Analyst. And, it you're the perpetrator, we'll protect your identity.

A String Evaluator

We've been asked many times for a version of Icon that would allow the user to type in Icon expressions and have them evaluated on the spot. For example, with such a facility, you could type in

```
write(repl("hello ", 3))
```

and the output would be

```
hello hello hello
```

Such a facility would be particularly handy for persons learning Icon. They could just try out various expressions and see what they do without having to edit, compile, and run a complete Icon program. And there are plenty of times when we could have used such a facility when testing new features for Icon or in trying to run down bugs.

While Icon itself doesn't have such a facility (and it wouldn't be easy to add one), it can be done with an Icon program. In fact, there are very capable programs in the Icon program library for evaluating Icon expressions represented by strings.

The key to evaluating an Icon expression represented as a string is string invocation, the feature of Icon that converts a string for a function name or operator into the actual function or operator.

The general problem of evaluating a string representing an Icon expression is fairly complicated. We'll consider a simpler problem here to illustrate what string invocation can do: a procedure to evaluate function calls with simple literal arguments.

Starting with a Special Case

Our approach to such a problem is to start with a simple special case and then see how it can be generalized.

Suppose for example, we start with a string that represents a function call with a single numeric argument, such as "sin(2.5)". We'll also assume such strings are syntactically correct, and worry about error checking later (or maybe not at all). Then it's just a matter of string scanning to get the name of the function and its argument, followed by string invocation to apply the function to its argument:

```
procedure feval(exp)
   local fnc, arg

   exp ? {
     fnc := tab(upto('('))
     move(1)
     arg := tab(upto(')'))
     }

   suspend fnc(arg)

end
```

The values of fnc and arg are, of course, strings. For example, if exp is "sin(2.5)" then the last expression in feval() is equivalent to

```
suspend "sin"("2.5")
```

String invocation takes care of getting from the string "sin" to the function sin and automatic type conversion takes care of converting the string "2.5" to the real number 2.5.

You might ask why we used suspend instead of return. We're planning ahead. It's always safe to use suspend in place of return when its argument expression produces one value. If the argument expressions fails, control flows off the end of the procedure, causing its invocation to fail, as it should. By using suspend, feval() can be used to generate results if exp represents the call of a function that is a generator.

It's worth noting that feval() works for unary (prefix) operators if parentheses are placed around the argument. For example,

```
feval("?(10)")
```

produces a randomly selected integer in the range of 1 to 10, inclusive, and

```
every write(feval("!(487)"))
```

writes 4, 8, and 7.

feval() also works for (programmer-defined) procedures in the same manner as for (built-in) functions. String invocation handles both.

A simple use of feval() might be something like this:

```
procedure main()

   while exp := read() do
      every write(feval(exp))

end
```

Actually, some error checking is needed; we'll leave that as an exercise. *Hint:* Look at the function proc() if you're not already familiar with it.

Generalizing

Of course, feval() as given above is quite limited. If the argument to the function in exp is a quoted literal, the quotes are included as part of the argument — not exactly what you'd want. And if the argument string contains a right parenthesis, the naive string scanning used above thinks it's the closing parenthesis for the call and discards the rest of the string. You probably will see other potential problems.

There also are cases where relying on automatic type conversion for the value of arg won't produce the intended result, as in

```
feval("type(3)")
```

All of these problems can be passed off to a procedure getarg(), which gets the argument and converts it to the correct type, as well as handling omitted arguments. We'll leave that as an exercise — doing it may give you some insight into string scanning and the details of Icon's literal constants. If you don't want to bother, look in the Icon program library. We'll also postulate a procedure getfnc() to get the function name. It's not needed here, but it might be handy if feval() is further generalized.

With the "vanilla" versions of these procedures, the package becomes

```
procedure feval(exp)
   local fnc, arg

   exp ? {
      fnc := getfnc()
      move(1)
      arg := getarg()
      }

   suspend fnc(arg)

end
```

Another limitation of feval() is that it only handles function calls with one argument. It's not difficult to parse the argument list, given a procedure getarg() that returns the next argument after consuming the punctuation character following the argument. The arguments then can be accumulated in a list, so that feval() might start like this:

```
procedure feval(exp)
   local fnc, arglist

   arglist := [ ]
   exp ? {
      fnc := getfnc()
      while put(arglist, getarg()) do
                 ⋮
```

Good enough, but there's now the question of how to apply fnc to its arguments. One possibility is

```
   suspend case *arglist of {
      1:  fnc(arglist[1])
      2:  fnc(arglist[1], arglist[2])
                 ⋮
```

That's plain ugly. And even if you're willing to do the work to provide for as many arguments are you're likely to ever see, you can't make it completely general. There's no limit to the number of formal parameters that can be declared for a procedure.

There's a much easier and more elegant method, which is completely general: list invocation. All that's needed is

```
   suspend fnc ! arglist
```

The whole procedure comes down to this:

```
procedure feval(exp)
   local fnc, arglist

   arglist := [ ]

   exp ? {
      fnc := getfnc()
      while put(arglist, getarg())
      }

   suspend fnc ! arglist

end
```

What feval() now can handle by way of function calls is only limited by the capabilities of getfnc() and getarg(). For example, if getarg() encounters a function call in an argument, it could call feval() recursively, and similarly for getfnc() — it's possible for a function call to produce a function. We'll leave these possible extensions as exercises.

Alternatives

There are other ways of approaching the problem of evaluating a string that represents an Icon expression; ones that are more general and very different from the technique used here. There's material of this kind scheduled for future updates to the Icon program library and we'll have more to say on this subject in future articles in the Analyst.

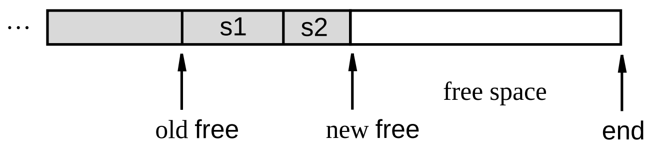
Acknowledgment

Mark Emmer called our attention to the method for handling an arbitrary number of arguments in function calls. He came up with it in a very practical context: testing the external function mechanism of ProIcon 2.0.

String Allocation

Background

Strings that are created during the execution of an Icon program are stored at a place in memory called the string region. Strings in the string region are allocated contiguously, starting at the beginning. Thus, the string region is divided into two portions: an allocated portion and a free portion. Whenever a new string is created, it is added to the end of the allocated portion of the string region, decreasing the size of the free portion. The string region can be depicted as a long sequence of characters:



where free identifies the boundary between the allocated space and the free space. Of course, there are many more characters in the string region than suggested by this diagram — typically 65K of them.

If the free portion is not large enough to hold a newly created string, a garbage collection is performed, discarding strings that are no longer needed and compressing the allocated part, making more room in the free part. See Reference 1 for more information about the details of allocation and garbage collection.

Strings created during program execution, called allocated strings, are not null-terminated as they are in C and some other programming languages. Null-termination, which adds a character with code zero to the end of a string, is used as a means of finding the end of a string. Icon accomplishes this by keeping both a pointer to the first character of a string and its length in an Icon string value.

This method makes computation of the length of a string fast, and it also allows null characters to appear in Icon strings. Perhaps more important, a substring of an existing string can be formed by changing only the pointer to the first character and the length, whereas with null termination, it's generally necessary to copy a portion of the existing string, since a terminating null character would overwrite a character in the string of which it's a part. A substring operation in Icon such as

```
s[i:j]
```

does not allocate any space in the string region.

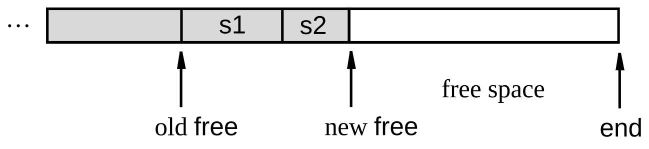
Several Icon operations do allocate space in the string region, the most notable being reading and concatenation.

Concatenation

Concatenation works by copying its two argument strings to the end of the allocated portion of the string region. The result of

```
s1 || s2
```

can be depicted as follows:



In general, such a concatenation allocates space for

```
*s1 + *s2
```

characters and copies that many characters.

One of the problems with concatenating in this fashion is illustrated by the following program segment that builds up a string piece by piece:

```
result := ""

while s := read() do
   result := result || s
```

In the loop, space is allocated for the string that is read, which then is assigned to s. The concatenation then allocates space for result and s, and the resulting string is assigned back to result. Note that all new string data comes from reading. If the successive strings read are indicated by subscripts, the pattern of allocation is

$s_1\ s_1\ s_2\ s_1\ s_2\ s_3\ s_1\ s_2\ s_3\ s_4\ s_1\ s_2\ s_3\ s_4\ \ldots$

Every previously read string is copied to perform the next concatenation. Although the previous values of result are discarded when a garbage collection occurs, the amount of space allocated is clearly much larger than is needed for the final result. In addition, garbage collection can be expensive, and the cost of garbage collection may depend on other things that have gone on during program execution, not just on the concatenation loop.

If you look at the final result of the concatenation loop, you'll see it's just

$s_1\ s_2\ s_3\ s_4\ \ldots$

The rest is "garbage". Is all the intermediate allocation necessary? That's the kind of question that the implementation of Icon attempts to address.

Optimizations

Short of changing the way concatenation, and hence string allocation, is done, the question becomes a more general one: "Are there situations in which it's not necessary to allocate all of the space needed in the most general case?"

One situation is when one or both of the arguments of concatenation is the empty string, in which case no concatenation or allocation is necessary. We'll come back to this case later.

There are two other situations that lend themselves to optimizations.

Optimization 1: The code segment discussed above illustrates a situation in which an optimization is possible. As the loop is evaluated repeatedly, result is the last string allocated when another string is read in and assigned to s by the next iteration of the loop. After reading, but before concatenating, the last two allocated strings are result and s. But that's exactly what's needed for the concatenation!

In other words, because of the order in which strings are allocated, result and s are already concatenated. And it doesn't take much to check for this situation, since an Icon string value contains a pointer to its first character and the length. For the concatenation

```
s1 || s2
```

pseudo-code for the check looks like this:

```
if loc(s1) + len(s1) = loc(s2) then … # done
```

If this test succeeds, no allocation is done, the new value points to s1, and its length is the sum of the lengths of s1 and s2.

With this optimization, the pattern of allocation for the loop given earlier is

$s_1\ s_2\ s_3\ s_4 \ldots$

which is exactly what's needed for the final result, with no extra allocation.

Note that the test above is more general, and applies anywhere in the allocated portion of the string region, although the chance of its succeeding anywhere but for the last two allocated strings is small.

Optimization 2: There's another situation in which part of the allocation for concatenation can be avoided — when the first argument of the concatenation is the last allocated string and hence at the end of the allocated portion of the string region:

```
if loc(s1) + len(s1) = free then … # don't copy
```

In this case, it's only necessary to append the second argument of the concatenation to the end of the string region.

A situation in which Optimization 2 applies is shown by

```
result := ""

while s := read() do
   result := s || result
```

In the absence of Optimization 2, the pattern of allocation for the loop is

$s_1\ s_1\ s_2\ s_2\ s_1\ s_3\ s_3\ s_2\ s_1\ s_4\ s_4\ s_3\ s_2\ s_1 \ldots$

With Optimization 2, the allocation pattern is

$s_1\ s_2\ s_1\ s_3\ s_2\ s_1\ s_4\ s_3\ s_2\ s_1 \ldots$

The savings aren't as great as for Optimization 1; you can't expect to reverse the order of strings, which is what is happening here, without some copying.

Other Optimizations

Earlier we skipped over the situation in which one or both of the arguments of concatenation is the empty string. This sounds like a situation worth checking, since when it occurs, no allocation is necessary.

There's a kicker, however: The test has to be applied for every concatenation. It turns out that the cumulative cost of checking for this situation takes more time on the average than it saves (which is not true for Optimizations 1 and 2).

There's a moral here: An optimization may sound good, but the cost of testing for a special case may outweigh the savings when it does apply.

Some interesting examples of this occur in an implementation of SNOBOL4 that included some clever "heuristics" that turned out to be unfortunate in practice [2].

The trouble is that optimizations usually can't be evaluated analytically. And it may be very time-consuming and expensive to evaluate them in practice. The natural tendency is to rely on an intuitive feeling of the usefulness of an optimization. Such intuition often is faulty.

Other Allocation Strategies

If you think about how Icon allocates string space, you'll notice that the same string may occur in many different places in Icon's allocated string region.

Occasionally someone suggests that before allocating a new string, there should be a search to see if it already exists. This certainly is a bad idea — it's very expensive to perform character comparisons just on the chance of finding a copy of a string that can be "re-used".

A different idea would be to put all strings in a hash table [3], so that each different string would be allocated only once. Although there are some fast and clever hashing techniques, they eventually come down to character comparison, which is quite expensive compared to an occasional garbage collection to remove unused strings. Furthermore, in a hashing scheme, the sharing of characters among substrings is not possible.

But again, the only way to be sure about this is to actually try it (or possibly simulate it, although a simulation in a case like this is difficult and error-prone). This would be a *major* effort, and one hardly worth undertaking, especially when the expectation clearly is negative.

The Whole Truth

Optimization 2 has a long standing. It was first used, to our knowledge, in the SPITBOL implementation of SNOBOL4 [4] and was incorporated in the first implementation of Icon.

Although we've given Optimization 1 priority in this article because it is so effective, this optimization didn't occur to us until we started to write this article and, in the process of showing that Optimization 2 did not apply to appending to an evolving string, discovered there was a better optimization that did apply.

Optimization 1 was not included in Icon until Version 8.4, so if you're running an earlier version, your programs won't benefit from this optimization.

Taking Advantage of the Optimizations

In most cases, the optimizations for string concatenation, like other aspects of the implementation of Icon, are "just there" and you needn't think about them when you program.

On the other hand, if you like to hone your programs for maximum performance, you may want to give a little thought to taking advantage of the concatenation optimizations.

One thing to watch out for is intermediate allocation that defeats the optimizations. For example, in

```
result := ""

while s := read() do {
   write(repl("=", *s))
   result := result || s
   }
```

the allocation for

```
repl("=", *s)
```

follows the allocation for

```
read()
```

and defeats Optimization 1 that otherwise would apply.

On the other hand, other strings can be appended to the concatenation without additional allocation, as in

```
result := ""

while s := read() do
   result := result || s || ","
```

Here, Optimization 1 applies to the first (left) concatenation, while Optimzation 2 applies to the second (right) concatenation.

It's worth knowing that literal strings are contained in the code produced by compiling a program, and space for using them is not allocated in the string region unless they have to be copied in concatenation or some other operation that allocates space in the string region [5]. For example,

```
marker := "+"
```

does not allocate space, but

```
marker := marker || "+"
```

does.

One situation in which Optimization 1 may apply at a place other than at the end of the string region occurs in string scanning. For example, in

```
text ? {
   if t := tab(many(&letters)) then
      t := t || tab(upto(' ') | 0)
   }
```

tab(many(&letters)) and tab(upto(' ') | 0) are adjacent in text, which need not be at the end of the allocated portion of the string region.

Output as a Weak Form of Concatenation

Jim Gimpel likes to call output a "weak form of concatenation". His point is that when a string is written to a file sequentially, it is automatically appended to (concatenated onto) the last string written.

As shown above, the concatenation of strings in a program requires the allocation of space (which may eventually result in a garbage collection) and also the copying of characters. This is costly compared to, say, arithmetic.

In many programs, most strings that are built up by concatenation are eventually written to a file. If you can arrange to write the strings as they are computed and in the order they need to be output, you can save the costs involved in concatenation.

As a very simple example, it's considerably more efficient to use

```
write(s1, s2)
```

than to use

```
write(s1 || s2)
```

It's often possible to avoid actual concatenation altogether in programs that transform input data. Sometimes all it takes to use output instead of concatenation is looking at the problem in the right way. In fact, it can be fun to see how far you can carry this. We'll have an example of such a program in the next issue of the Analyst.

References

1. "Memory Monitoring", The Icon Analyst 2, October 1990, pp. 5-9.

2. "Performance of Storage Management in an Implementation of SNOBOL4", David G. Ripley, Ralph E. Griswold, and David R. Hanson, *IEEE Transactions on Software Engineering*, Vol. SE-4, No. 2 (1978), pp. 130-137.

3. *The Macro Implementation of SNOBOL4; A Case Study of Machine-Independent Software Development*, Ralph E. Griswold, W. H. Freeman, San Francisco, California, 1972.

4. “MACRO SPITBOL — A SNOBOL4 Compiler”, Robert B. K. Dewar and Anthony P. McCann, *Software — Practice & Experience*, Vol. 7 (1977), pp. 95-113.

5. “An Imaginary Icon Computer”, The Icon Analyst 8, October 1991, pp. 2-6.

Type Inference in the Icon Compiler

The Icon compiler performs several kinds of optimizations in order to produce faster and smaller code. The subject of optimization is, in general, a complex one. If you’re interested in the details of what goes on inside the Icon compiler in this regard, you may wish to get a copy of Ken Walker’s doctoral dissertation [1].

One of the most important optimizations involves type inference. Type inference refers to the process of determining, by analysis of a program, what set of types the operands of operations may have at different places in the program.

Types in Icon

As you know, unlike most programming languages, Icon has no type declarations. In fact, a variable in Icon can and usually does take on values of different types during program execution. Another way of looking at this is to say that Icon’s variables are not typed, but its values are. An Icon value contains within it information that identifies its type.

There are several reasons why Icon does not have type declarations and hence a compile-time type system:

• It saves programmers from having to write a lot of tedious type declarations.

• It allows useful programming techniques, such as the ability to write a procedure that takes arguments of different types and returns values of different types.

• It allows heterogeneous structures, such as lists that contain both integers and real numbers.

Even if Icon had a compile-time type system, it couldn’t be complete because of Icon’s pointer semantics — it’s possible to have a list of lists, a list of lists of lists, and so on.

Despite the lack of type declarations, Icon has a strong run-time type system in the sense that all operations check the types of their arguments to be sure that they are correct. Along with this run-time type checking, inappropriate types are automatically converted (“coerced”) to the appropriate ones if possible.

The problem with this is that type checking takes time. And in the Icon interpreter, it’s done for all operations repeatedly, whether or not the types are correct. This adds significantly to execution time.

Type Usage

Although Icon does not dictate what types of values can be assigned to variables, most Icon programmers use most variables in a type-consistent way. For example, if a variable is used as a counter, it likely to be assigned only integer values. This type consistency is a by-product of good programming style. It makes sense to associate a particular use with a particular variable and hence for it to be assigned values of only one type throughout program execution. Similarly, the operand of an operation usually always has the same type.

Of course, there are exceptions to this. These exceptions may be for very good reasons, as in heterogenous structures, they may be accidental, or they may be the result of “sloppy” programming — we’ve been known to write things like

```
x := sort(x)
```

to convert a set into a sorted list.

Earlier we said variables usually have values of different types during program execution. This seems to contradict our claim that most variables are used in a type-consistent way. The reason most variables have different types during program execution is that all variables have the null value initially. But this really is a special case that does not affect subsequent type usage.

Determining Type Usage

The observable fact that most operands of operations are type-consistent suggests that an analysis of the program can detect where type checking is not needed and generate code without the type checking. This is, in fact, one of the most important kinds of optimizations the Icon compiler makes.

Both the theory and the implementation of type inference for Icon are complicated. We again refer you to Reference 1 for the details. A few simple examples indicate what’s possible.

Consider the following program segment:

```
text := ""

while line := read() do
   text := text || "," || line
```

It’s obvious that the arguments of the two concatenations are strings. text is a string prior to the loop by virtue of the assignment to it. In the loop, text is always assigned the result of concatenation, which again is a string. The literal is of course a string, and line is a string because it is assigned a string as the result of read() in the control expression of the while loop; if read() fails, the expression in the do clause is not evaluated. It’s clear, therefore, that the concatenations do not need to check the types of their arguments.

Note also that at the termination of the loop, text is a string, although line may not be (if it wasn’t a string before the loop and read() failed the first time through the loop). If line had some other type before the loop, it may be either that type or a string after the loop.

This suggests how type inference can be performed. Some operations are known to produce values of specific types. The types of values an expression may have depend on the path of execution. For example, if there are alternative paths, a variable may have values of different types at different times at a particular point in the program.

Now consider what happens in the example above if an inappropriate type is used:

```
text := ""

while line := read() do
   text := text || ',' || line
```

One of the arguments of concatenation now is a cset, which must be converted to a string. Since concatenation, like most infix operations, is left-associative, the arguments group as

```
(text || ',') || line
```

which means that the first concatenation must convert its right argument to a string, but the second concatenation does not have to convert either of its arguments.

Of course, a smart compiler could replace the cset literal in this situation by the equivalent string literal and avoid type checking code for the first concatenation. It's not clear, however, that this kind of thing happens in real programs often enough to make it worthwhile for the compiler to check.

There are several reasons why the possible types an expression may have can be uncertain. One of these is because some Icon expressions can fail, as illustrated above. This is true even in simple assignment expressions. For example, after evaluating

```
loc := find(header)
```

The value of loc can either be an integer (if find() succeeds) or whatever it was before (if find() fails). Note that all variables always have some value; null if nothing else. Thus, assignments that can fail tend to add to the possible types a variable can have. On the other hand, an assignment such as

```
line := repl("=", 10)
```

always results in line having a value of type string at the place in the program after this expression. However, it's not true that an assignment that can't fail always leaves the variable to which the value is assigned with only one possible type. In

```
x2 := copy(x1)
```

the type of x2 after the assignment depends entirely on the type of x1.

Similarly, in

```
n3 := n1 + n2
```

the type assigned to n3 depends on the types of n1 and n2, but it can be only an integer or a real number.

So you can see the situation is complicated and that the types an operation may produce depend not only on program flow but also on the properties of the operation.

The Inferencing Process

In order to perform type inference, the compiler must be able to model program execution to the extent necessary to determine the set of possible types that expressions may produce.

To do this, the compiler builds a representation of the program and performs what's called *abstract interpretation*. In order for this process to be practical and so that termination can be assured, only certain aspects of program execution are handled. For example, only types, not actual values, are considered. This may, of course, result in less perfect type inference than is theoretically possible; this is a compromise with reality. In practice, fairly crude type inferencing methods yield reasonable results and the fairly sophisticated one used by the Icon compiler usually comes close to what is theoretically possible.

In an addition to the simple kinds of situations discussed above, to be effective, type inference must be able to deal with two other aspects of Icon: structures and functions.

As mentioned above, a structure can be heterogeneous and have elements of different types. Each element of a structure is a variable, of course, but it's not practical to treat each one separately. It may not even be possible to determine the size of a list. Therefore, if a list contains both integers and real numbers, each list element is treated as if it could be an integer or a real.

Furthermore, most structures in Icon can grow and shrink. Thus, if a string is pushed on a list that formerly contained only integers and real numbers, every element of the list is now treated as if it could be a string, integer, or real number.

While there are situations where this treatment of structures leads to poor type inferencing, in most cases it does quite well, primarily because of the way most Icon programs are written.

It's also not possible to keep track of individual structures — in fact, there is no way to know how many there will be, since that can, for example, depend on the data on which the program operates. The type inferencing system handles this by assigning a distinct structure type to each *location* in the program where a structure can be created. For example, in

```
tags := ["first", "second", "third"]
counts := [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

two different list types are associated with the two list-creation operations. Lists created at the first site are initially lists of strings, while those at the second site are initially lists of integers. It’s important to understand that the type inferencing system treats the lists created at the two sites as two different types, although in Icon they are the same type. Thus, the type inferencing system adds types of its own to assist in the inferencing process.

Functions also require separate handling. In order for the compiler to do a good job of type inference, it needs to use information about functions: what types of arguments they require, what types of values they produce, and whether or not they can fail.

In most Icon programs, functions are what they appear to be: reverse(x) is a function that expects a string argument and produces a string value. However, reverse is a variable, and it’s possible to make an assignment to it, as in

```
reverse := pop
```

so that reverse(x) may mean something entirely different from what it appears to mean.

Changing the value of an identifier that initially is a function is rarely done intentionally, but the compiler must take the possibility into account in order to do correct type inferencing. The compiler does this by first checking for function-valued identifiers that may have values assigned to them and by treating each of these as a separate type in subsequent type inference.

Procedures are handled the same way as functions: There’s no difference to the type inferencing system between the two.

Using Type Information

The compiler uses information derived from type inference to eliminate unnecessary type conversions.

In the worst case, in which nothing can be inferred about the arguments of an operation, a general form of the operation, with complete type checking and coercion, is used, and it usually is handled by a call to a routine in the run-time library. This most general form of the operation is essentially the same as the one used in all situations by the interpreter.

If type inference provides more specific information about the possible types for one or more arguments, the most general routine is tailored by pruning out type checking and coercion code. The result may be sufficiently small that it can be placed in-line in the generated code.

The Benefits of Type Inference

As we mentioned earlier, Icon programmers tend to use types in a consistent way, despite the freedom Icon allows to do otherwise. This is an empirical observation — if it weren’t true, there would be little point in type inference.

Type consistency varies considerably from program to program. In the Icon program library, the range is from about 60% to 100%, with the average being about 80%. In some cases a low percentage is inherent in what the program does. In other cases, it is the result of how the program was written. These figures are, incidentally, from the type inference system in the compiler. The actual percentages may be somewhat higher, since the type inference system isn’t perfect — it can’t be, actually. However, it does a better job than a human being can do by hand. Our experience has been that when we’ve suspected a defect in the type inference system, we’ve found out instead that we were wrong and it was right. We think the present inferencing system is good enough that work to improve it is not justified. In any event, the figures given for type consistency above are what’s determined by the compiler, which is what counts.

The Icon Analyst

Madge T. Griswold and Ralph E. Griswold
Editors

The Icon Analyst is published six times a year. A one-year subscription is $25 in the United States, Canada, and Mexico and $35 elsewhere. To subscribe, contact

Icon Project
Department of Computer Science
Gould-Simpson Building
The University of Arizona
Tucson, Arizona 85721
U.S.A.

voice: (602) 621-8448

fax: (602) 621-4246

Electronic mail may be sent to:

icon-project@cs.arizona.edu

or

...{uunet,allegra,noao}!arizona!icon-project

THE UNIVERSITY OF
ARIZONA
TUCSON ARIZONA

and

The Bright Forest Company
Tucson Arizona

© 1991 by Madge T. Griswold and Ralph E. Griswold
All rights reserved.

There are two ways to measure the effectiveness of type inference in reducing execution time: (1) the effect it has on the time it takes to evaluate expressions in isolation and (2) how much time it saves in the execution of typical programs.

The problem here is that some programs spend much of their execution time in operations like table lookup and storage management, which have nothing directly to do with expression evaluation and which are unaffected by optimizations resulting from type inference. The bottom line, of course, is how much type inference increases the speed of program execution. It does no good to optimize something that consumes an insignificant amount of time.

Just considering individual expressions, it is possible to get dramatic improvements in execution speed. For example,

```
1 + 2 + 3 + 4
```

executes about 5.8 times faster with optimizations resulting from type inference than without them. If you have the Icon compiler and the latest version of the Icon program library, you can try other expressions yourself — empg.icn can be used to time expressions in either the compiler or the interpreter.

Of course, what really counts is not individual expressions, but complete programs, which brings in the issue of execution time spent where the compiler can't reach.

Overall, the improvement in execution speed of compiled code over interpreted code ranges from nearly zero to a factor of seven or eight, with something between three and four being typical. But that includes all optimizations. What does type inference contribute? That can be determined by turning off type inference and leaving other optimizations. For most programs, type inference contributes about one-half the total speed improvement. So you might expect type inference to contribute a factor of perhaps two to execution speed over the interpreter.

The Cost of Type Inference

Type inference is a complex process and it can require considerable computational resources, both in time and memory.

A worst-case analysis shows that type inference has time complexity of O(n^7), where *n* is the size of the program. (In this context, program size refers roughly to the number of variables and operations.) In other words, the amount of time it possibly can take for type inference is proportional to the seventh power of the size of the program. If that were really the case in practice, it would be truly alarming — the time for type inference in a 100-line program could be 10,000,000 times longer than for a 10-line program.

While it's possible to contrive programs that achieve the O(n^7) figure, it's very difficult to do and nothing like this kind of time complexity occurs in "real" programs. We've timed type inference on a large number of Icon programs, including some real monsters. The observed time complexity from these tests is only O(n^2).

From a practical point of view, type inference is very fast for small programs. For large ones, type inference can be slow, but it's not impossibly slow.

The amount of space required for type inference also is a concern. A worst-case analysis gives a space complexity of O(n^3). In practice, it seems to be more like O(n^2). This definitely can be a limiting factor for a large program compiled on a platform with a small amount of memory. It's one thing to have to wait a long time for an Icon program to compile, but it's quite another to have it fail to compile because of lack of memory.

Fortunately, if type inference requires too much in the way of resources, it can be disabled by a command-line option.

Open Questions

It seems obvious that you can get more out of type inference by taking care in the way you program, although there are pitfalls as noted below.

The questions are how much benefit you can get and what you need to do to get it.

At present, these matters are somewhat murky. If a program already has a high degree of type consistency, increasing its type consistency probably isn't going to help much, unless a type inconsistency at a critical place is having a big effect (the figures given above are static ones and don't take into account how often different expressions are evaluated). But suppose you have a program with a type consistency of 60% that could be improved to 100% (such improvements are not always possible — in fact, they rarely are). You'd expect to get a noticeable improvement in execution speed.

We've worked on several programs with mixed results. One problem is that type inference is complicated and even knowing what type inference discovers on a per-expression basis doesn't always lead to obvious improvements. It's also easy to out-smart yourself — increasing type consistency actually can slow a program down if what you do introduces additional computations into the program.

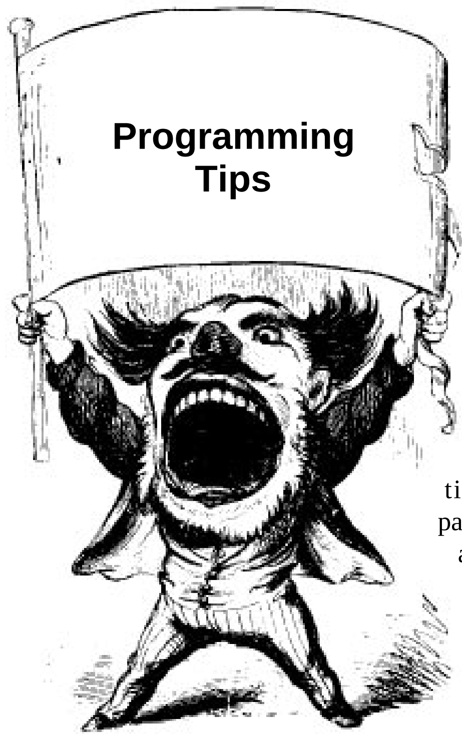
We hope to develop some programming guidelines that will be easy to follow and that will lead to improved program performance without requiring an advanced degree in type inference. Better yet, we're looking at the possibility of developing tools that work in conjunction with type inferencing to assist programmers in improving the performance of their programs. Program visualization, which is one of the focuses of our current research, seems particularly apt here.

It's worth remembering, however, that execution speed isn't everything. There are plenty of programs in which execution speed isn't important at all. And one of the virtues of Icon is that it makes programming easy and even fun.

That's not compatible with excessive concern for performance. But in those cases where performance must be a serious consideration, help is on its way.

Reference

1. *The Implementation of an Optimizing Compiler for Icon*, Kenneth Walker, technical report TR 91-16, Department of Computer Science, The University of Arizona, 1991.



Building Lists

Icon often provides several ways of doing the same thing. Such alternatives add richness to the language, but they also mean you have to choose among them. Sometimes stylistic concerns are paramount, but there are times when questions of speed and storage requirements need consideration. Worse, if you happen to do something that's inappropriate and don't think of another, better way, the impact on program performance can be serious.

List construction is a case in point. There are two basic ways of building up a list one element at a time: list concatenation and using the deque ("double-ended queue") functions put() and push().

For example, in building up a list of lines from a file, you might do it either of the two following ways:

```
lines := [ ]
while lines := lines ||| [read()]
```

or

```
lines := [ ]
while put(lines, read())
```

The difference in performance between the two methods is dramatic — list concatenation is much slower than using put(), and the performance of list concatenation becomes increasingly worse as the list becomes large.

Why is this? It partly has to do with how lists are implemented, but there's also a basic difference between the two language features. Having to create a separate list for each line of input in list concatenation should be a warning.

In list concatenation, both arguments must be lists; hence the need for creating a one-element list. A new list is created with enough space for the elements of both lists to be concatenated, and then the elements of the two lists are copied into the new list. In the case here, the result is then assigned back to lines, replacing its former value. Note that the two lists used in the concatenation are no longer needed once the assignment is made. They are garbage collected if necessary, but the net effect is a lot of allocation and copying.

This points up another problem with using list concatenation to add an element to a list. If there already are *n* elements in the list, there must be enough memory for this list and for the new list of *n*+1 elements; the space from the first list is not available for re-use until after the assignment is made back to lines. If *n* is large …

Why should the use of put() be better? In good part, that's because the implementation of lists takes into account the possibility that elements may be added by deque functions. Space is provided in advance, and the addition of a new element does not require the allocation of more space until what's there is used up. If more space is needed, another block is allocated, but all the elements that are already in the list are unaffected.

The implementation of lists is rather sophisticated in this regard. See References 1 and 2 for the details.

You might well argue that you shouldn't have to be an expert on the implementation of Icon to use it efficiently. That's true, and if you look at the Icon language book [3], you won't get much help either. The examples given there for list concatenation illustrate the operation but also use a bad programming practice — the result of trying to provide examples that are easy to understand without giving enough attention to their appropriateness in practice. In the book's defense, the subsequent examples of building lists use deque functions where they are the better choice.

Downloading Icon Material

Most implementations of Icon are available for downloading electronically:

BBS: (602) 621-2283

FTP: cs.arizona.edu (cd /icon)

In any event, the Icon language book does not attempt to say much about efficiency — it's a complicated and advanced topic, and one that depends on implementation details that may change.

If you want a simple rule that is appropriate in most situations, it would be to use list concatenation only when you already have two lists to concatenate, and especially when they both are large. Don't form a one-element list just so you can add an element to another list by list concatenation.

When you have concerns about efficiency, and especially when you're trying to chose between alternative programming techniques, we recommend benchmarking [4-5]. The program `empg.icn` in the Icon program library makes this easy. Sometimes it takes a little thought to find simple expressions to compare performance. Here are the ones we used as input to `empg` for comparing list concatenation and the deque method:

```
:lines := [ ]
lines |||:= [" "]
```

and

```
:lines := [ ]
put(lines, " ")
```

The first line in each case creates a list for subsequent use, but evaluates it only once (the colon indicates this). The second expression, on the other hand, is evaluated repeatedly (1,000 times is the default, but that can be changed when you run the benchmarking programs).

In our benchmark tests, when run with 1,000 iterations, list concatenation takes more than 33 times longer than the deque method. List concatenation also performs 73 garbage collections for the default 65KB block region, while the deque method doesn't do a single garbage collection. For 5,000 iterations, list concatenation takes more than 388 times longer than the deque method and performs a rather amazing 2,995 garbage collections, while the deque method still doesn't do a single garbage collection.

In addition, since the list concatenation method requires two large lists to be in memory at the same time, the block region expands to over 120KB. This only works on platforms that have enough memory. For example, you can't create a 5,000-element list one element at a time by using list concatenation on an standard MS-DOS system.

Back Issues

Back issues of The Icon Analyst are available for $5 each.

This price includes shipping in the United States, Canada, and Mexico. Add $2 per order for airmail postage to other countries.

References

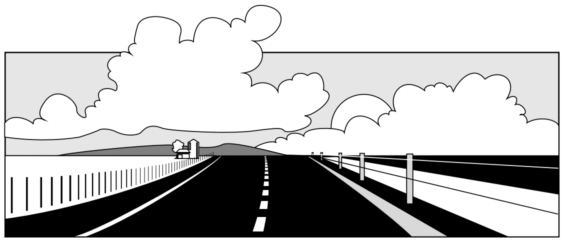
1. *The Implementation of the Icon Programming Language*, Ralph E. Griswold and Madge T. Griswold, Princeton University Press, Princeton, New Jersey, 1986.

2. *Supplementary Information for the Implementation of Version 8 of Icon*, Ralph E. Griswold, Icon Project Document 112, Department of Computer Science, The University of Arizona, 1990.

3. *The Icon Programming Language*, second edition, Ralph E. Griswold and Madge T. Griswold, Prentice Hall, Englewood Cliffs, New Jersey, 1990.

4. "Benchmarking Expressions", The Icon Analyst 1, August 1990, pp. 10-12.

5. "Benchmarking Expressions", The Icon Analyst 2, October 1990, pp. 10-11.



What's Coming Up

In the next issue of the Analyst, we'll have an article on how to get to operating-system facilities from inside an Icon program.

We'll also have an article on encapsulating expressions in procedures and the advantages this provides.

Another feature of the next issue will be an analysis of a complete program from problem specification through the details of coding.