The Icon Analyst

In-Depth Coverage of the Icon Programming Language

February 1993
Number 16

In this issue ...

Program Visualization

Over the past few years we have become increasingly interested in ways in which program performance and behavior can be characterized in an understandable way. When we say program behavior, we don't mean the results produced by program execution but the more general characteristics of program execution as a whole. For Icon, we're interested in the places where a program spends most of its time, how deep recursion gets, how extensive goal-directed evaluation is, how much and what kinds of storage the program allocates, how much type conversion it performs, and so forth.

The problem with understanding program behavior in a high-level language like Icon is the enormous amount of computational activity that occurs during program execution. One source-language operation may produce dozens of lower-level events that are important in understanding program behavior. In raw form, such information is all but incomprehensible. Traditional methods for summarizing data, such as tables and charts, may omit or even obscure important patterns of activity.

The key to understanding program behavior is visualization. In the sciences, the word visualization is used to refer to the rendering of objects and processes in the form of images, often animated. Such images make it possible to understand things by utilizing the amazing visual-processing capabilities of the human brain and mind. (The fundamental importance of images in human life is pointed out by one of the main definitions of "to see"— to understand.)

Until a few years ago, visualization was available only to a few persons with access to expensive special-purpose equipment. The recent decline in the cost of computer systems with good graphical processing capabilities has made visualization economically practical for "ordinary mortals" like us.

Much of the focus of scientific visualization is on the physical world — models of complex molecules, the representation of fluid flow, and so forth. For this reason, scientific visualization is sometimes referred to as "visualizing reality", although if often goes beyond reality into somewhat fanciful subjects like colliding neutron stars. The term program visualization refers to the rendering of computational processes in a visual fashion.

Issues in Program Visualization

There are many intriguing and difficult issues in program visualization. In most scientific visualizations, there is an underlying physical reality and geometry. Molecules exist. Fluid flow is something we can experience. Computational processes are just as real, but they are more abstract. There often is no natural geometry in computational processes to provide an obvious basis for understandable images.

More specifically, some aspects of program behavior are not cast as images easily or naturally. For example, we still don't have a good way of representing control flow in generators and goal-directed evaluation.

Some aspects of programs, such as data structures, involve the interconnection of many components. The problem of automatically laying out large graphs in a understandable fashion is very hard and is a research topic in itself. Producing attractive graphs automatically (and aesthetics are important to understandability) is, in general, out of the question.

Then there's the "real-estate" problem. Most computer displays are relatively small. There's hardly ever enough room to present images the

way you'd like.

While we don't propose to solve such fundamental problems, we do have some new ideas about program visualization that seem interesting.

Previous Work — Memory Monitoring

Our first adventure in program visualization dates back to 1985, when Gregg Townsend set out to design a system for displaying the management of allocated data in Icon.

To get the data needed to visualize storage management, he added instrumentation code to the Icon interpreter to write out information about storage allocation and garbage collection.

The resulting tool, called MemMon, shows Icon's allocated data regions, with each allocated object appearing as it is allocated [1, 2]. Different types of data are represented by different colors on the display. Figure 1 shows a typical MemMon display. Of course, a great deal is lost in a black-and-white snapshot. If you still have the color MemMon image that came with the second issue of the Analyst, you might want to dig it out and take a look at it.

MemMon proved very useful in several ways: understanding the role of different types of data, evaluating alternative programming techniques, identifying inefficient programming techniques, locating and correcting inefficiencies in the implementation, and so forth.

The success of MemMon inspired our current work in program visualization. And the pursuit of good visualization tools has led us into several other areas.

Laying the Groundwork for Visualization

Some program visualization techniques require modifying the program to be visualized by inserting calls to display routines. We wanted to be able to visualize the execution of a program without modifying it or unduly perturbing its behavior. This led to extensive instrumentation of the Icon interpreter.

Since program visualization is a relatively new field, much work is experimental and speculative. It's difficult to justify investing months of effort in developing a new visualization tool whose usefulness cannot be assured in advance. One way to make programming easier is to use a high-level programming language — a theme we've promoted for many years. What's more natural than writing program visualization tools in Icon? But we went a step further.

MT Icon was written so that a program to be visualized could run in the same execution space as a visualization tool, enabling the tool to access

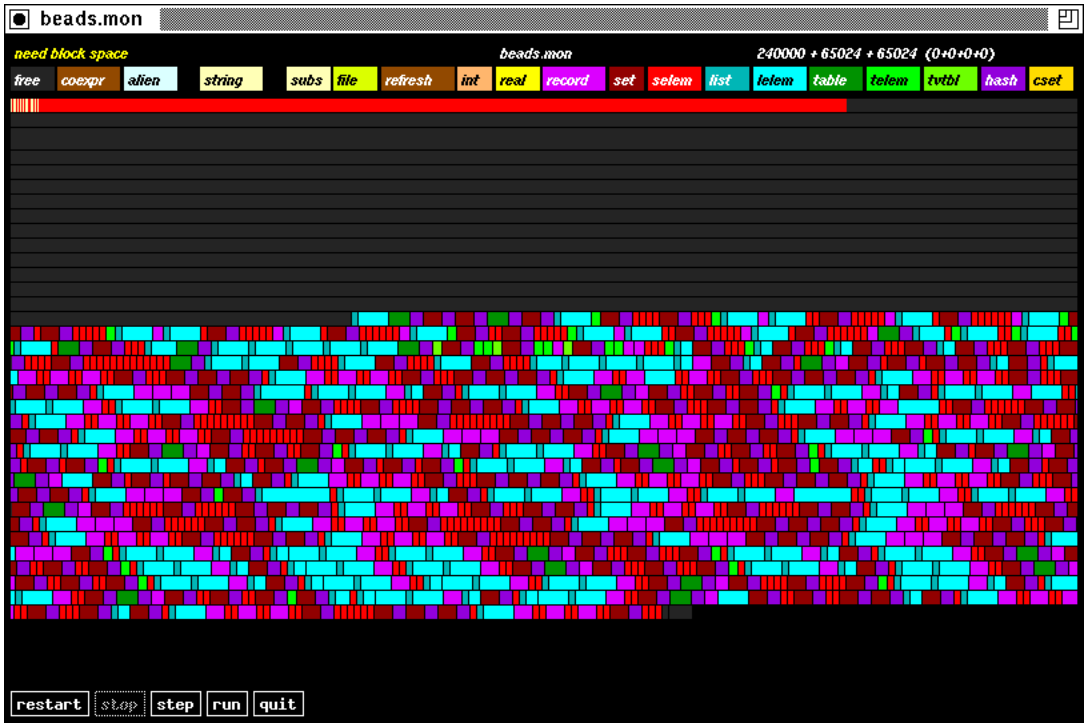


Figure 1. A MemMon Display

the program state and data of the program being visualized [3]. At the same time, the instrumentation code was changed to pass information by means of co-expression activation, instead of writing it to an output stream. This model offers the advantage that a visualization tool can deal with data from the program it is visualizing in its native format.

Graphics programming, which underlies program visualization, is still a difficult area. To engage in experimental work, we needed ways to make graphics programming easier. This led to the development of X-Icon, which adds graphic capabilities to Icon [4, 5].

It is indeed much easier to program graphics in X-Icon than it is to program them in, say, C. The pleasant surprise was to discover that graphic applications written in X-Icon run acceptably fast for program visualization on current workstations. In fact, in many cases, the speed is limited by graphic actions, not by X-Icon.

So, as is typical of such research projects, we set aside our original objectives while we developed the infrastructure to achieve them. Along the way, however, we have been able to work on issues related to program visualization and to develop some program visualization tools. Now that the mechanism to support program visualization is in place, we can get back to our original objectives and concentrate on program visualization and its applications.

Examples

Our work so far is largely exploratory. Some examples follow.

Visual Metaphors

Traditional visual representations of abstract data take the form of charts and graphs. In the case of program visualization, these usually are animated to show the course of program execution. It's rather natural for programmers to think in terms of directed graphs, tables of values, and so forth. But there's a lot of evidence that part of the human mind works more effectively on patterns and that patterns are important in characterizing program behavior.

One approach to exploiting the pattern-matching capabilities of the human mind is to use metaphorical representations in place of the more literal ones.

Last year we tackled storage management as a specific subject for such investigations. Storage management is a good subject because it is "rich" in terms of the number and variety of events that occur during program execution. The relatively literal visualization provided by MemMon also provided a basis for comparison.

So we set out to visualize storage allocation using a variety of visual metaphors. In the resulting visualization tools, different types are represented by different colors as they are in MemMon. But their geometries and animations mostly bear little relationship to those of MemMon or to the actual layout of memory.

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!arizona!icon-project

THE UNIVERSITY OF
ARIZONA
TUCSON ARIZONA

and

The Bright Forest Company
Tucson Arizona

For example, we tried visualizing storage allocation as randomly placed dots, with colors corresponding to the type allocated and areas correspond to the amounts of allocation — a "splatter painting" of storage allocation. See Figure 2.

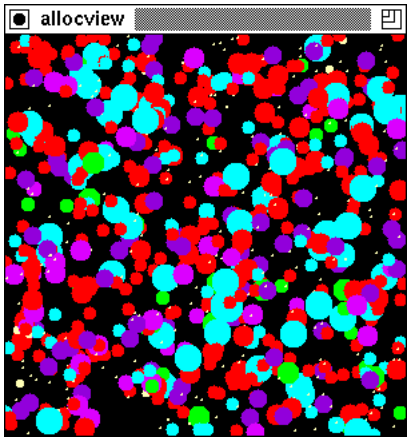


Figure 2. A "Splatter" View of Allocation

Such a representation certainly is not literal, but you can "see" things you otherwise might not. You get an overall "picture" of the frequency and amount of allocation, how the pattern changes with the course of program execution, and you often can detect distinct phases of program execution.

We went on from there to try other visual metaphors with names like pinwheel, nova, tapestry, web, flowers, haystack, cannonade, and so on.

When this coding orgy ended, we had more than 30 ways of visualizing storage allocation metaphorically. Of these, some were plainly awful — ideas that seemed good but that just didn't pan out. Some of the visualization metaphors were, not surprisingly, mundane. But a handful were interesting and useful.

Since animation is an essential component of these visualizations, it's difficult to get any real idea of what they're like from looking at snapshots. We have, however, put six of our favorite views of storage allocation at the end of this issue of the Analyst.

All the snapshots are from the same program and taken at the same point in execution. Compare the "splatter" view in Figure 2 with the corresponding color view to see the difference that color makes. In looking at the color views, you may find it helpful to know that strings are ivory, lists are cyan, record are purple, sets are red, and tables are green.

To get an idea of what kinds of things different views reveal, note the prominence of strings in the pinwheel and tapestry views, and their relative insignificance in the nova and splatter views. The pinwheel and tapestry views show only the number of allocations, not the amount allocated, while the nova and splatter views show the amount. Thus, many strings are being allocated, but they are small. This aspect of allocation in this program shown here is not easily seen in MemMon, by the way.

Incidentally, it's a testimony to the ease of graphics programming in X-Icon that all these visualization tools were written over a period of a few weeks when other business was going on. Along with developing the tools themselves, we developed infrastructure to make it easier to write new tools. By the end (when we ran out of fresh ideas and energy), it was taking from 30 minutes to an hour or two to write a new tool, depending on its geometrical complexity. When a tool is that easy to write, it is not so painful if you discover it's useless.

You may think the metaphorical views are whimsical or even goofy. They may be. But at an hour or so apiece, it's cheap fun.

Coping with Limited Screen Space

Except for very expensive experimental systems, the amount of space available for visualization is very limited. This problem isn't unique to program visualization — typical graphical user interfaces attempt to create the illusion of a desktop on a screen that's less that a foot square (try working on a physical desk of that size). The problem isn't only size — the resolution of even the best computer monitors is far less than that of paper.

You know the standard ways of dealing with this problem: multiple windows, scrolling, overlapping or hidden objects, pull-down menus, various devices for zooming in and out, visual filing metaphors, icons, text in a tiny size, and so forth.

In program visualization, the problems are only worse. If you're trying to understand a program, you almost certainly want to be able to look at the text of the program itself. But even in a high-level programming language like Icon, a program may be hundreds or even thousands of lines long — certainly too long to fit on the screen in a

readable format all at once. Even the call tree of an Icon program may have hundreds of nodes and nicely occupy the entire screen, if given the opportunity.

And suppose you want to see all the data structures a program has created. A program may produce hundreds or even thousands of structures, and even one structure, like a list or table, may have thousands of elements.

So it falls to the person writing program visualization tools to find ways to use screen space to display information in a compact yet understandable way.

The standard approach to displaying a large amount of text is to scroll within a window. For non-textual displays, the ability to zoom in and out and select interesting areas can be useful.

A more unusual and promising approach to limited screen space in program visualization is to use "fish-eye" views [6]. The idea is to give a larger amount of screen space to portions of an image of greatest current interest, with material of less interest getting less space.

Figure 3 is a snapshot of a visualization tool that provides a fish-eye view of text. The type size for the text is largest near the point of interest, dwindling in size to the vanishing point for text further away. A scrollbar allows the portion of the text of greatest interest to be selected. Note that although portions of the text far from the center of attention are illegible, they still provide useful cues in the patterns of line lengths. Incidentally, this tool relies on scalable fonts that are supported by the X11R5 release of X.

An approach to the call-tree problem mentioned earlier is shown in Figure 4, which is a snapshot of a tool called Algae (to suggest the miniaturizing of a tree). In Algae, "cells" on a hexagonal grid are highlighted as procedures call other procedures (resulting in vertical growth) and as procedures suspend (resulting in horizontal growth.

The user can interact with Algae, laying down barriers that cause the monitored program to stop if procedure "growth" exceeds the specified bounds. The rectangular barriers shown in Figure 4 are produced by clicking with the left mouse button on a cell, which establishes a corner. The user also can create a barrier of arbitrary shape by clicking on individual cells with the right mouse button. If procedure call or suspension growth

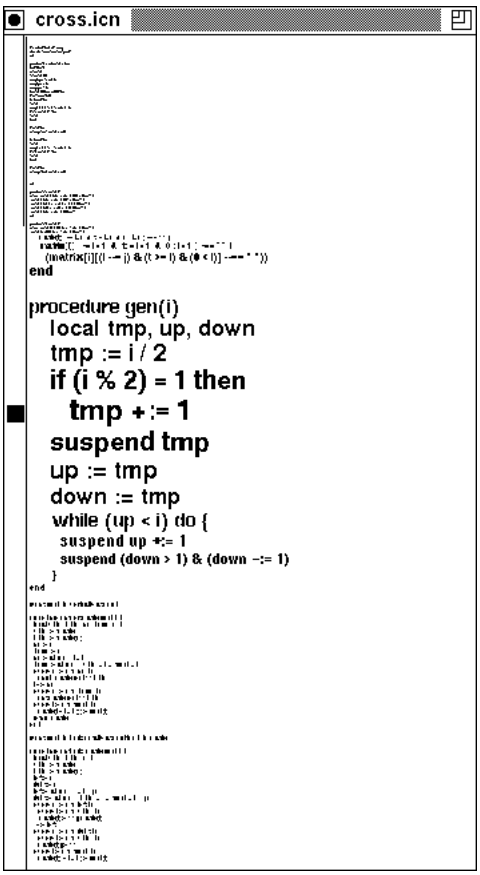


Figure 3. A Fish-Eye View of Text

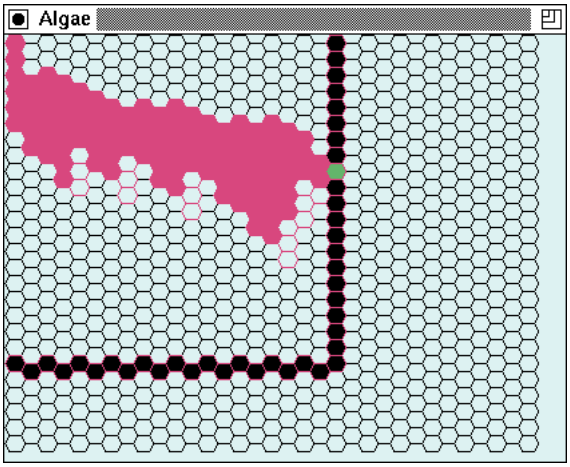


Figure 4. "Algae"

reaches the barrier, the monitored program stops, after which the user has a variety of options.

Sometimes miniature views can give an idea of a pattern of behavior that may be lost when rendered in a larger size. Figure 5 on the next page shows output from a "piano roll" visualization of program activity.

Figure 5. A "Piano-Roll" View of Execution

The vertical axis represents source-code location, with each vertical pixel position corresponding to one source line. If the program is too long for the window, the display is scaled, and more than one line is assigned to a pixel. The horizontal axis represents time, with each pixel column corresponding to one clock tick (10 milliseconds, in this case). Note the repetitive nature of the image. This shows that the program is performing a major computation repeatedly, presumably in a loop. This image illustrates an interesting aspect of program visualization: You often can learn something useful about a program using visualization without ever seeing the program itself.

This kind of visualization also can be combined with a source-code listing to relate activity to specific sections of the source code, but such a visualization requires a considerable amount of screen space.

Visualizing Data

One of the most exciting aspects of our system for program visualization is the ability of a monitor to access data in a monitored program [3]. And, since the monitor and the monitored program are both written in Icon, data can be processed directly. For example, a monitor can generate elements of a list in a monitored program in the same way it would generate the elements of a list of its own.

In fact, a monitor can modify data in the monitored program or even insert its own data into the monitored program. The possibilities go far beyond visualization. Imagine how this capability could be used in a debugger.

Getting back to program visualization, there are again many ways to view the same structure. Figure 6 below shows a multi-window visualization tool in which each window shows the details of a data structure. Scrolling is available for structures that have many elements. Note that the representation of data in this tool is nonetheless essentially textual.

An example of a tool that takes a more metaphorical view, motivated by the problem of limited screen space, is shown in Figure 7 on the next page. The larger window shows all the structures in the program, color coded by type with identifying serial numbers. Clicking on a structure causes another window to pop up, showing the details of the structure in miniature. A table and a list are shown.

The really difficult problem in visualizing structures is showing the relationships among them. Since structures in Icon have pointer seman-

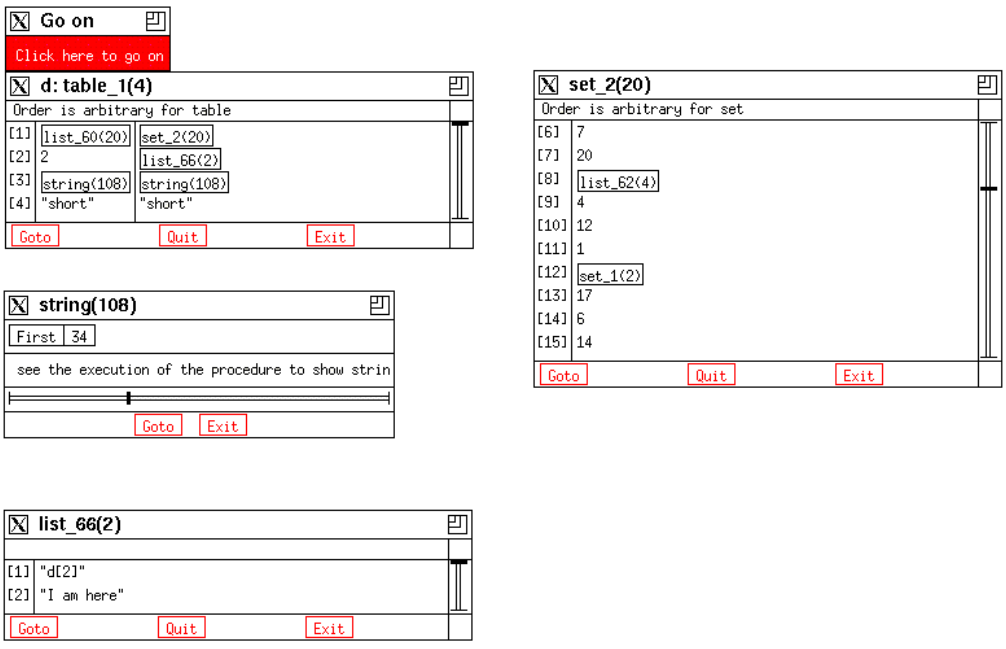


Figure 6. Visualizing Data

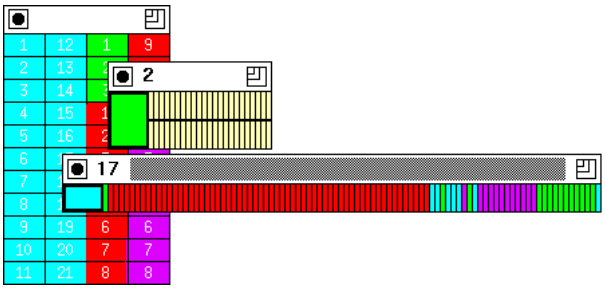


Figure 7. Another View of Data Structures

tics [7], this suggests arrows pointing from structure to structure. This sounds easier than it is. Some programs have many structures and the resulting graph may be large and impossibly complicated.

There are some obvious things that can be done, such as restricting data visualization to subsets of structures. There also are fish-eye approaches to visualizing data. We have not given much attention to such possibilities yet.

Multiple Visualizations, Multiple Views

Even a simple programming language has many features you might wish to visualize. For a sophisticated, high-level programming language like Icon, there are literally hundreds of possibilities. And, as suggested by the examples of visualizing storage allocation mentioned earlier, there also are many different possibilities for viewing the same aspect of program behavior. It's not necessarily the case that one way of viewing program behavior is better than others. The best view may depend on the situation.

In fact, a program visualization system needs to be able to support multiple, simultaneous visualizations. We have approached this problem by designing a program monitor that loads other program monitors and serves as an event multiplexer. This monitor, called Eve [8], does not perform any visualizations of its own. Instead, Eve services other monitors that perform visualizations. Eve gets events from the monitored program and distributes them to other monitors. Figure 8 on the next page shows a snapshot of multiple visualizations with Eve's window at the upper left.

Eve displays a menu listing visualization tools from which a user can make selections that may include start-up options for tool configuration. Eve provides buttons to start and stop monitoring, disable and enable specific visualizations, or "iconify" them (turning their visualization windows into icons).

You might expect multiple visualizations to be too slow to be useful. We certainly worried about this and went to some lengths to reduce overhead where we could. It was a relief to find that multiple visualizations generally run acceptably fast. Of course, speed depends on the hardware you have. We usually run on Sparc IPXs. Note also that our visualizations are comparatively simple. We have not, for example, attempted any three-dimensional views, realistic renderings, or sophisticated animations.

There's another side to this. The ability of a human mind to comprehend images, although amazingly large, is still limited. With multiple visualizations, shifts of attention from one visualization to another slow down comprehension further. It's not unusual for such visualizations to run too fast for ready comprehension; it may be necessary to delay them artificially or for the user to "take a breath", stopping monitoring altogether for a while.

Tool Interaction

It is important for users to be able to interact with visualization tools. Such interaction can be used to select different visualization options, to chose among alternate views, and so on.

More interesting possibilities lie in the visual specification of important aspects of program behavior. Algae's barriers to recursion and suspension illustrate the possibilities.

Tools also may interact with each other. For example, if the user clicks on a pixel in a piano roll display, it signals a text browsing tool, if present, to display the portion of the program surrounding the line corresponding to the selected location on the piano roll display.

Conclusions

We've come a long way from poring over computer listings of octal core dumps (yes, *core*) to interactive animated color visualization.

We've just begun to explore program visualization in Icon. There are so many possibilities that it's hard to decide what to do next.

Of course, visualization does not need to stand alone. In the back of our collective mind (not so far back, actually) is a visual programming environment for Icon.

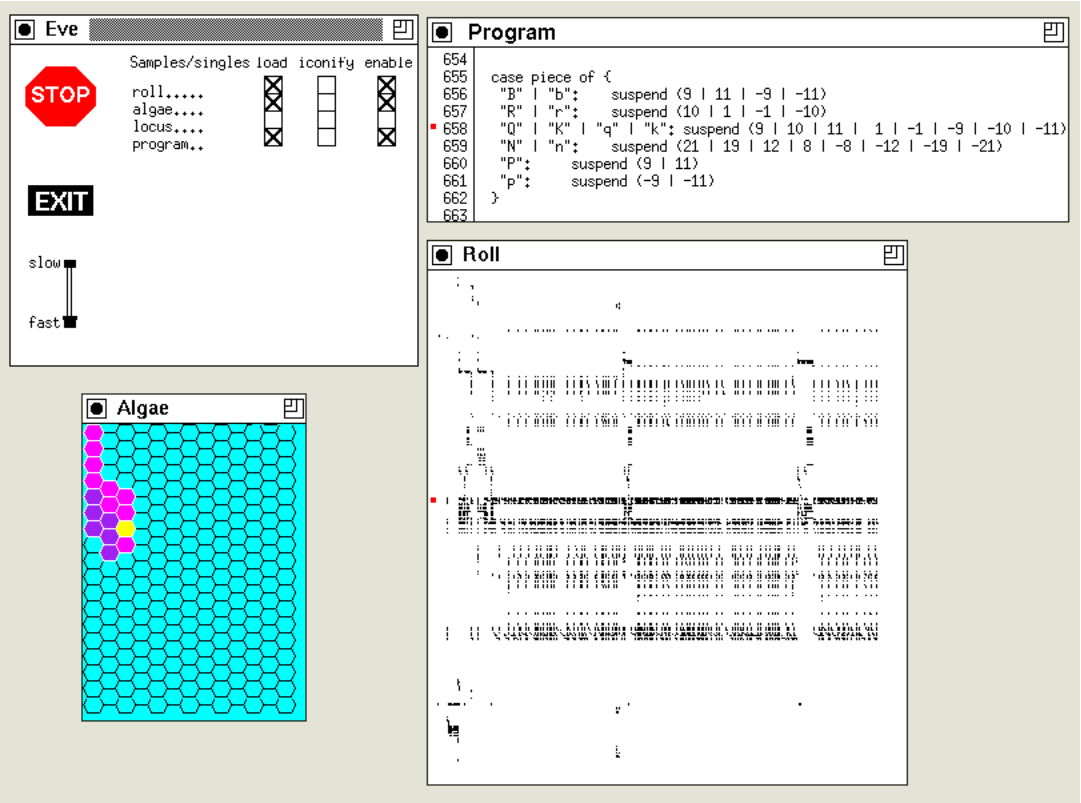


Figure 8. Multiple Visualizations Under Eve

Acknowledgments

Clint Jeffery wrote Algae, Eve, and the fish-eye tool for visualizing text. The tool for displaying Icon's data structures textually was written by Song Liang as a class project [9]. Gregg Townsend wrote MemMon and the first version of the piano-roll visualization tool.

Most of the material in this article appeared first in a conference paper [10].

References

1. *The Visualization of Dynamic Memory Management in the Icon Programming Language*, Ralph E. Griswold and Gregg M. Townsend, Technical Report TR 89-30, Department of Computer Science, The University of Arizona, 1989.

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

3. "Monitoring Icon Programs", The Icon Analyst 15, pp. 6-10.

4. *X-Icon: An Icon Windows Interface*, Clinton L. Jeffery, Technical Report TR 91-1, Department of Computer Science, The University of Arizona, 1991.

5. "An Introduction to X-Icon", The Icon Analyst 13, pp. 5 -10.

6. "Generalized Fisheye Views", George W. Furnas, *CHI'86 Proceedings; Human Factors in Computing Systems*, New York, 1986, pp. 16-23.

7. "Pointer Semantics", The Icon Analyst 6, pp. 2 -8.

8. *Eve: An Icon Monitor Coordinator*, Clinton L. Jeffery, Icon Project Document IPD179, Department of Computer Science, The University of Arizona, 1992.

9. "Icon Class Projects', *Icon Newsletter* 39, pp. 11-12.

10. "Visualizing Program Execution in Icon", *Proceedings of the Sixth International Conference on Logical and Symbolic Computing*, Madison, South Dakota, 1992, pp. 33-46.

Sparse Arrays

In Issue 14 of the Analyst [1], we showed various ways in which multi-dimensional arrays can be implemented using lists of lists. As mentioned in that article, there are limitations on the size of an array that can be implemented in this way — it may require too much memory.

Very often, however, large arrays are "sparse", and only a small percentage of their elements are actually referenced.

Consider, for example, a monitor that follows the location of program execution in a monitored program, counting the number of times each program token is evaluated. The natural representation of the program is a two-dimensional array with a row for each program line and with columns corresponding to characters of the lines. Execution does not occur on white space in a program, and only one location is needed for each token. Furthermore, since some lines usually are longer than others, the use of a rectangular array has the effect of adding white space to the ends of some lines. For typical programs, only a few percent of the elements in such a location array are ever referenced.

This suggests a sparse array in which space for all elements in not allocated when the array is created. Such an implementation thus requires a mechanism for adding to the array depending on the elements that actually are referenced.

Tables of Tables

Icon's table data type is the obvious candidate for implementing sparse arrays. Tables start out empty and an element (and the space for it) is allocated only when the element is referenced the first time. What could be more natural for sparse arrays? For example, a one-dimensional sparse array can be created by

```
A := table()
```

and indexed as

```
A[i] := x
```

If this is the first time A is subscripted with i, an element is created for the index (key) i.

Implementing sparse arrays using tables is not all easy going, however. The problems are inherent in the deferred allocation. If, for example, a two-dimensional array is implemented as a table of tables, it might start out the same way as a one-dimensional array:

```
A := table()
```

Now consider an array reference:

```
A[i][j] := x
```

As shown above, the first subscript, i, is not a problem. But A[i] is null, the default value given when A was created. If nothing else is done, there's a run-time error when an attempt is made to subscript this null value with j.

The apparently obvious thing to do is to create A as

```
A := table(table())
```

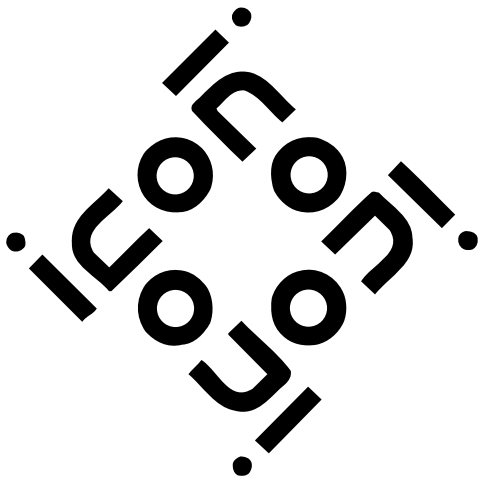
so that the default value of an element of A is itself a table. This is a nice idea, but it doesn't work. The double subscript works, but the values in the array don't come out as expected.

The results may be baffling. In fact, this is one of the commonest and most frustrating problems for novice (and sometimes experienced) Icon programmers. They often even think there's something wrong with the implementation of Icon.

The problem is that there is only *one* table that serves as the default value for *all* newly referenced elements of A. We've mentioned this problem and its relatives before [2]. Despite what you might like to have happen, the expression above is equivalent to

```
T := table()
A := table(T)
```

which illustrates that there are only two tables associated with the array. No wonder things don't come out right!



No, there isn't any way around this, like

```
A := table(create |table())
```

This just gives A a default value that's a co-expression. There's no way to have this co-expression automatically activated to produce new tables as needed. And if you think about it, you'll realize there can't be and still have consistent semantics for expression evaluation and tables.

What's required instead is an explicit test for a new key. If a key is new, a new table needs to be assigned to it. Going back to the original formulation, this initialization of new elements of A can be done as follows:

```
A := table()
     ⋮
/A[i] := table()
A[i][j] := x
```

so that if A[i] is null, and hence i is a new key, a table for "column" i is created.

For a two-dimensional sparse array, this initialization, although annoying, is not all that bad. And it can be hidden in a procedure such as

```
procedure ref_sparse(A, i, j)
  /A[i] := table()
  return A[i][j]
end
```

Notice that since the value returned is a reference to an element of a structure, it is a variable and assignment can be made to the procedure call, as in

```
ref_sparse(A, 1, 5) := x
```

Sparse arrays with many dimensions are more of a problem. If the number of dimensions is large (or unlimited), a recursive approach based on the procedure above can be used [3].

Subscript Encoding

There's another technique that can be used to implement sparse arrays of arbitrarily high dimensionality using a single table. It isn't pretty, but it works and avoids the complexity of the tables-of-tables approach.

The idea is to encode multiple subscripts as a single string. For example, instead of using

```
A[i][j] := x
```

you might use

```
A[i || ":" || j] := x
```

where the colon is used to separate the integers.

We told you it wasn't pretty … . The ugliness can be hidden in a procedure, however:

```
procedure ref_sparse(A, i, j)
  return A[i || ":" || j]
end
```

This encoding device works with any number of subscripts, so a more general procedure is

```
procedure ref_sparse(A, subscripts[])
  local s
  s := ""
  every s ||:= !subscripts || ":"
  return A[s]
end
```

Note that the trailing colon produced by the every loop causes no problem and it's easier, simpler, and faster to leave it at the end of the key.

With this formulation, it's possible to write expressions like

```
ref_sparse(A, 1, 5, 100, 200000, 4) := x
```

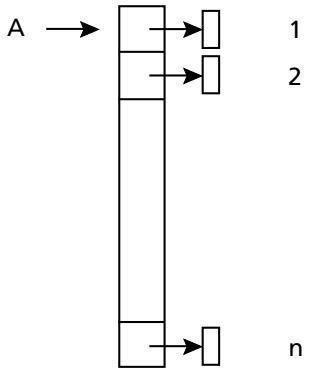
A List of Tables

There are situations in which sparse arrays can be implemented with a combination of lists and tables. If a two-dimensional array is not very sparse and if it's relatively short and of fixed extent in one dimension, the list-of-lists approach given in Reference 1 can be usefully recast as a list of tables:

```
A := list(n)
every !A := table()
```

Thus, there is a fixed number of rows, n, but the number of columns is unlimited. The resulting

structure is shown below. In this form, a sparse array consists of one list and n tables:



Note that this representation fits the situation of the program tokens mentioned at the beginning of this article.

One nice property of this representation is that no special initialization of new elements is needed — all the necessary tables are created initially. For example,

```
A[i][j] := x
```

works as it stands.

Comments on Using Tables to Implement Arrays

Tables offer great flexibility. But with this flexibility may come problems. For example, a table can be subscripted with *any* value. Unlike lists, there are no bounds checks for tables, so there is no automatic check of an out-of-bounds array reference. That must be provided in, say, an array-referencing procedure.

There are more subtle problems. There is no type conversion of table subscripts, so, for example `T[1]` and `T["1"]` reference different elements of `T`. It's easy for errors to occur as the result of this, and these errors may be hard to find. Again, the remedy is to provide explicit conversion in a procedure. Note that a table subscript need not even be numeric, which means that provision has to be made for handling erroneous types as array indexes.

On the other hand, there are situations in which having a sparse "array" that can be subscripted by non-numeric "indexes" can be very handy. However, it's probably best not to think of such a structure as an array. Of course, if indexes can be non-numeric, the encoding of multiple indexes as strings may need to be modified, or it may not work at all.

We also have glossed over issues like providing an initial value for all the elements of a sparse array. In the list-of-tables approach, handling this is easy. For example,

```
A := list(n)
every !A := table(0)
```

creates a sparse array with elements whose initial values are zero.

Choosing an Array Representation

We've shown four ways of representing arrays: as a list of lists, as a table of tables, as one table with encoded subscripts, and as a list of tables. Which is the best representation to use? This depends on the situation. If an array is not sparse or if it is not too big, the list-of-lists representation is the simplest and most efficient one. If a sparse representation is needed, there still are several choices, and the best one depends on details.

The factor to consider first is space. Although the use of tables to allow referenced elements to be added to the original structure saves space for elements that are never referenced, tables are larger than lists and table elements are larger than list elements. Hence, if an array is not sufficiently sparse, a sparse representation can actually take more space than the list-of-lists representation. Even where a sparse representation is clearly appropriate, the relative merits of the different sparse-array representations depend both on the degree and nature of the sparseness.

Referencing a table also is somewhat slower than referencing a list, although the difference is not as much as you might think, and it ordinarily is a secondary consideration.

While it is not possible to give precise guidelines for the choice of an array representation, a example may help in understanding what the factors are and how important they are.

A Comparative Example

We tried all of the representations with the program monitor described at the beginning of this article. The program we used for comparing the representations was of modest size: 102 lines, with the longest line having 73 characters. Thus the conceptual 73×102 array consisted of 7,446 elements in all. When the program was run, there

were 58,001 evaluations of tokens, but only 187 different tokens, so the array was about 97.5% sparse — definitely a candidate for a sparse representation. However, the test program was small enough that the list-of-lists representation was practical and it therefore could be compared to the sparse representations.

The results are shown at the bottom of the page. The figures for space are for bytes allocated for the structures used for the arrays.

It's actually possible to work out the approximate space requirements using the formulas cited in Reference 4 if you have one additional piece of information about the sparse array: the number of different columns that have referenced elements, which is 42 in this example. (This information is needed for the table-of-tables representation.)

The relative speeds should not be surprising, although the penalty for the sparse representations may not be as much as you'd expect.

On the other hand, the space used by the sparse representations shows that they really are worthwhile. However, before concluding that a table with encoded subscripts uses the least space, you need to know that the strings used to encode the subscripts require a lot of space. In fact, in this example, 283,173 bytes were allocated for strings. Many of these were duplicates, however, and all but 22,521 bytes could be reclaimed by garbage collection. So a fairer figure for encoded subscripts is 28,149 bytes.

It's also worth mentioning that in the table-of-tables representation, the creation of new tables was done in-line. If it's done with a procedure, the time increases to 50.1 seconds.

Which one to choose? In this case, we prefer the list-of-tables representation for its simplicity, economical use of memory, and speed. You may feel differently. And, of course, it all depends on the amount and nature of sparseness. For example, if all the tokens evaluated had been in a few columns, the table-of-tables representation would be more appealing.

	list of lists	table of tables	list of tables	table with encoded subscripts
bytes	63,774	10,964	13,500	5,628
seconds	41.6	45.6	42.6	46.0

Comparison of Array Representations

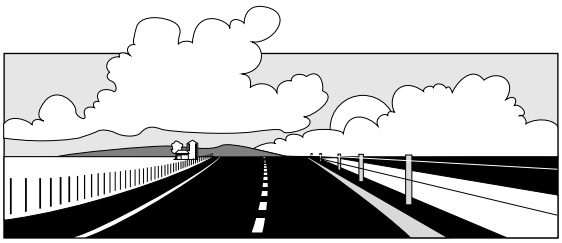
References

1. "Arrays", The Icon Analyst 14, pp. 2-4.

2. "Idiomatic Programming", The Icon Analyst 14, pp. 4-8.

3. "Programming Tips", The Icon Analyst 13, pp. 10-12.

4. "Memory Utilization", The Icon Analyst 4, pp. 7-10.



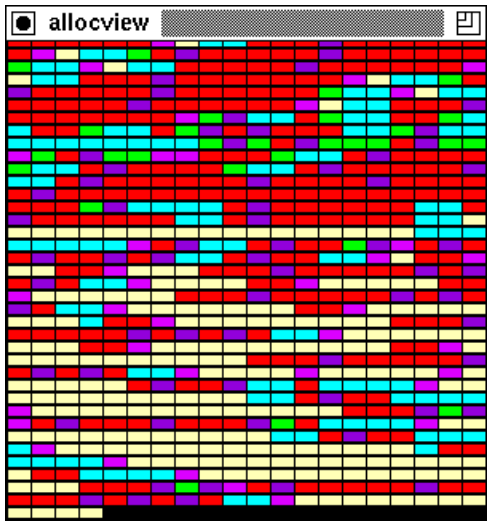
What's Coming Up

Over the years there have been a few programming languages closely associated with Icon that have become "lost" — languages that no longer exist or at least are not in general use.

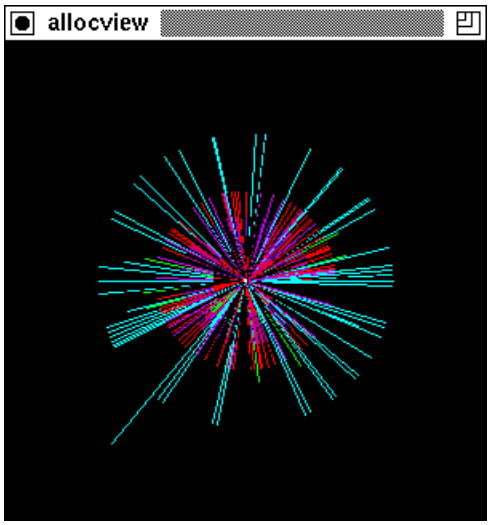
Four SNOBOL languages preceded Icon. We don't count these as lost. Despite substantial differences among the SNOBOL languages, the first three are really earlier versions of SNOBOL4, and SNOBOL4 is hardly dead — it's still available and in use on several contemporary platforms.

There are three Icon-related languages that we *do* count as lost — SL5, Rebus, and Seque. SL5 was a fully developed programming language in its own right that immediately preceded Icon. It was, in fact, abandoned to make room for Icon. Rebus, on the other hand, was an experiment in casting SNOBOL4 in a syntax that resembles Icon. Seque was something entirely different — an experiment in elevating the concept of sequences of values to first-class data types.

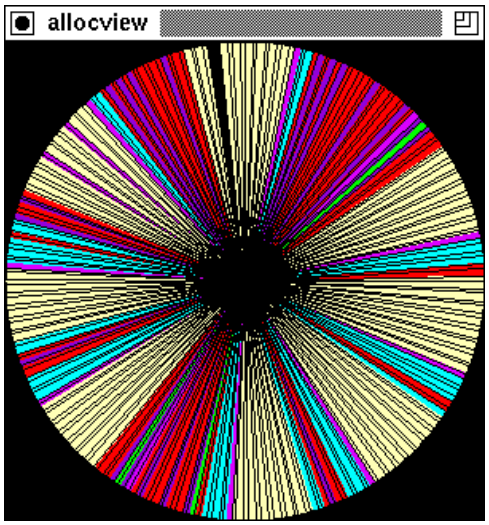
Starting in the next issue of the Analyst, we'll describe these languages and indicate how they relate to Icon.



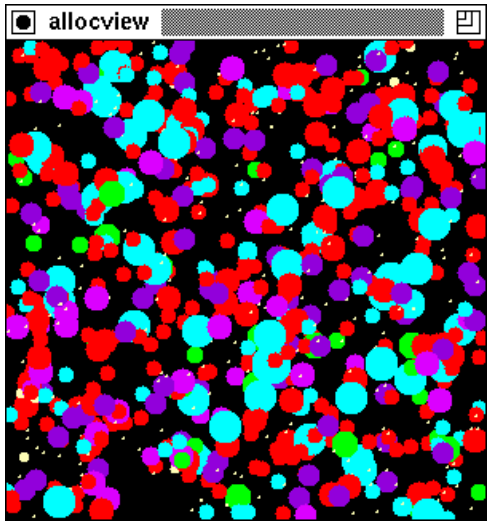
mosaic



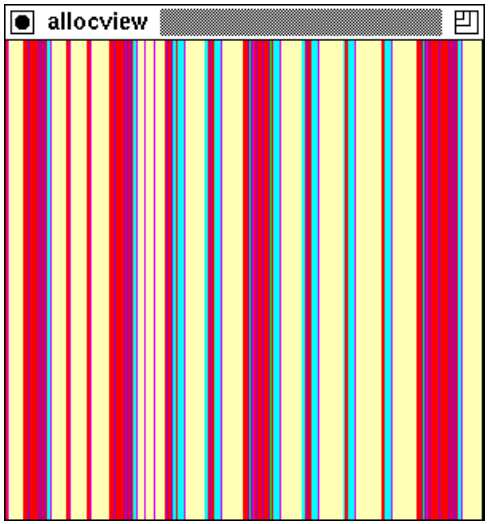
nova



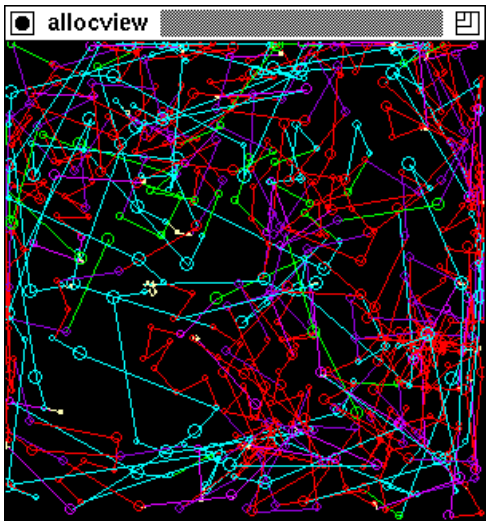
pinwheel



splatter



tapestry



web

Views of Storage Allocation in Icon