Monitoring Events in Icon Programs

Ralph E. Griswold

Department of Computer Science, The University of Arizona

1. Introduction

Most of the events that occur during the execution of a program go unnoticed, and fortunately so — in most cases, only the end results of computations are interesting.

In order to thoroughly understand a program, however, much more detailed information is needed about events that occur during program execution. This report describes instrumentation of the interpretive implementation of the Icon programming language that provides the information needed for visualization tools that can assist in debugging, performance measurement, and program analysis. The information provided by the instrumentation can, of course, be used for other purposes.

2. Events

Many kinds of events occur during the execution of a program. Examples of such events are changes in the location in the program where execution is taking place, the values produced by expressions, procedures calls, and so forth. The kinds of events that occur depend, of course, on the semantics of the programming language and its implementation. The kinds of events that are important depend on what aspects of program execution are to be visualized and to some extent on the nature of the visualization tools.

No program is totally independent of the implementation of the programming language in which it is written. Often there is no clear dividing line between the semantics of the programming language and its implementation. Similarly, program performance depends on the computer on which it is run and the environment for execution. Consequently, there are events that are not directly related to the program semantics.

The kinds of events needed by visualization tools fall into several categories. Many events correspond to the execution of specific source-language operations, such as procedure calls and returns. Other events correspond to specific aspects of the implementation, such as garbage collection. External events, such as clock ticks, are needed also.

There typically is some overlap between language events and implementation events. While the semantics of Icon imply storage allocation of some kind, the amount and, to some extent, the time and place of allocation depend on the implementation. Garbage collection also is implied by the semantics of Icon (and the properties of real-world computer systems), but the events related to it are clearly in implementation. In the case of storage management in Icon, these aspects of the implementation are acknowledged in the language by the keywords &collections, &regions, and &storage, which provide a linguistic bridge between the implementation of Icon and the language itself.

It is worth noting that visualization of aspects of the implementation of a programming language can be useful to programmers. Icon's memory-monitoring system [1] is an example. This system uses a history of storage allocation and garbage collection events to drive visualization tools that present pictures of Icon's allocated data regions and the details of memory management. These visualization tools, despite their focus on the implementation events, have proved valuable to Icon programmers in understanding program behavior, improving program performance, and selecting appropriate programming strategies.

Another example of the overlap between language and implementation events occurs in expression evaluation in Icon. The interpretive implementation of Icon is designed around a stack-based virtual machine whose instruction set lies somewhere between the semantics of Icon and the instruction sets of real computers on which Icon runs [2]. Some virtual machine instructions correspond directly to the semantics of Icon, while others have little or no relation to the language itself. For example, there is a virtual machine instruction corresponding to each Icon operator, but there also are virtual machine instructions that manipulate the stack but which have no direct relevance to the Icon language itself.

The number of relevant events that occur during the execution of a program may be enormous. A typical Icon program, for example, allocates thousands of objects and may process them many times during garbage collections. Recording events affects program performance. Processing events by tools is time-consuming and the amount of data to be transmitted or stored may be very large. Consequently, discrimination in selecting the kinds of events to be recorded is a serious practical concern. On the other hand, failure to anticipate the significance of an event, and hence failure to record it, can lead to unsatisfactory or even misleading results. There is an essential tension here between cost and value.

It is impractical to record everything that goes on during program execution. Some events are, of course, simply unimportant. Others can be skipped because of knowledge of the language or implementation allows them to be reconstructed, if necessary. For example, when a list is created, it is not necessary to record all the allocation events; these can be derived from a knowledge of the implementation.

3. Instrumentation of the Icon Interpreter

The interpretive implementation of Icon [2,3] has been selected for the development of program visualization tools. This implementation is mature, accessible, portable, and easily modified. Event data is produced by instrumentation that has been added to this implementation. This instrumentation is modeled after the instrumentation used for the Icon memory monitoring system mentioned earlier [1]. Indeed, the event monitoring system is an extension of the memory monitoring system and subsumes it.

Event-monitoring code has been added to the implementation at appropriate points to provide information about relevant events when they occur. Compilation of this instrumentation code is conditional; it is included only if a manifest constant is defined.

Event monitoring is inactive unless it is specifically requested. Event monitoring does not significantly affect the performance of Icon programs unless it is active. If it is active, event monitoring degrades performance somewhat but does not otherwise affect program behavior.

4. Characterizing Events

Most events have associated values. For example, the return from a procedure has an associated source-language value, and the allocation of a block of memory has an associated size and type. The allocation of a list can be viewed as an allocation event with two parameters, the type of object and the amount of space allocated or as a list allocation event with a single parameter, the amount of space allocated. The choice between such different characterizations depends partly on the use to which the event information is to be put and partly on how it is encoded. Since most events have only one associated value, this format is used uniformly in reporting events. Events with several associated values are encoded as multiple events with single values or the several values are encoded as one. Events that have no associated value are encoded with a dummy value. The overall result is uniformity at the expense of some artificiality.

5. Event Streams

The instrumentation is designed so that it can be used in different ways. For example, it can communicate directly with a program monitor. The instrumentation also can be used to write *event streams* that record the events that occur during program execution. An event stream can be piped into another process or save in a file for post-mortem analysis. Most of the remainder of this report is concerned with event streams.

The instrumentation does not produce an event stream unless a file (or pipe) to receive the event information is specified by a command-line option or an environment variable. See Appendix A.

Because of the vast amount of information produced during the execution of a program, some potential event information is discarded for practical reasons. For example, many Icon source-language values are very complex. It is impractical to produce the source-language values associated with events. Instead, source-language values are abstracted to their types in event streams.

The event description language used in event streams is an extension of the description language used for allocation history files the memory monitoring system [4]. The event description language is reasonably compact and easy to parse. An event stream consists of event tokens separated by white space (blanks, tabs, and newlines).

Except for a few special situations, white space is optional and can be used to increase readability or omitted to reduce the size of an event stream. An event token consists of a value and a code. Values are represented by nonnegative integers or, in a few cases, by strings. For example, the types of source-language values are encoded as small integers.

A value may be omitted if it is the same as the value for the last occurrence of the the same kind of event. This reduces the size of events streams considerably.

Event codes are represented by single characters excluding digits, white-space characters, and a few other characters that are reserved for special purposes. In practice, the codes are printable ASCII characters, although that restriction is not necessary and is used only for readability.

6. Event Contexts

For both conceptual and practical reasons, it is useful to identify different event contexts. For example, events such as calling a procedure, returning a value, and allocating storage occur as a direct result of the evaluation of expressions in the program. Such events can be thought of as occurring in an *evaluation context*. On the other hand, events during garbage collection occur in an essentially different context.

The choice of contexts is largely determined by the language and its implementation. For example, allocation and procedure invocation are quite different kinds of events, but they occur in the same context — expression evaluation. On the other hand, some kinds of events occur in more than one context. For example, the relocation of data during garbage collection is naturally cast as re-allocation, using the same kinds of events as for allocation in the evaluation context.

Context changes are treated as events — context beginning and ending events, where the context itself is the value of the event. Contexts can be nested.

7. Creating and Using Event Streams

Appendix A describes how to obtain event streams.

Writing tools that use event streams requires considerable knowledge of the specific contexts and events. Appendices B and C contain detailed information about these matters.

Appendix D contains an example of an event stream to illustrate the encodings used.

Facilities are available for processing event streams. These facilities use the same symbolic representation for events, values, and contexts as is used in the instrumentation that produces event streams. See Appendix E.

The instrumentation of events is evolving. New instrumentation is being added and some encodings may be changed. The use of symbolic names in support procedures minimizes the problems associated with such changes. However, the evolving and experimental nature of the instrumentation tends to make previous event streams obsolete.

Acknowledgement

Gregg Townsend collaborated in the design of event streams and the facilities for processing them.

References

1. R. E. Griswold and G. M. Townsend, *The Visualization of Dynamic Memory Management in the Icon Programming Language*, The Univ. of Arizona Tech. Rep. 89-30, 1989.
2. R. E. Griswold and M. T. Griswold, *The Implementation of the Icon Programming Language*, Princeton University Press, 1986.
3. R. E. Griswold, *Supplementary Information for the Implementation of Version 8 of Icon*, The Univ. of Arizona Icon Project Document IPD112, 1990.

4. G. M. Townsend, *The Icon Memory Monitoring System*, The Univ. of Arizona Icon Project Document IPD113, 1990.

Appendix A — Producing Event Streams

The instrumentation to produce an event stream is included only if the Icon translator (icont) and executor (iconx) are compiled with the manifest constant EventMon defined. The description that follows assumes the use of an instrumented version of Icon.

An event stream is produced by the execution of an Icon program only if it is requested prior to program execution. There are two ways to request an event stream: setting an environment variable and using a command-line option.

The Environment Variable EVENTMON

If the environment variable EVENTMON is set, its value is used as the file to receive the event stream. For example, in BSD UNIX

```
setenv EVENTMON prog.mon
```

causes an event stream to be written to prog.mon.

If the value of EVENTMON begins with a vertical bar, it is interpreted as a pipe through which the event stream is sent. For example,

```
setenv EVENTMON "|grep A"
```

filters the event stream through *grep*, sending to standard output only those lines that contain the letter A. Pipes, of course, can be used only on operating systems that support the facility.

Caution: The environment variable EVENTMON causes any Icon program compiled under an instrumented version of Icon to produce an event stream. Care should be taken to unset EVENTMON after a desired event stream is produced to avoid its being overwritten by Icon programs that are run subsequently.

The −E Option

An event stream also is produced if the command-line option −E to iconx is used. The argument to this option is interpreted in the same way as for the environment variable EVENTMON. For example,

```
iconx -E prog.mon prog
```

runs prog and directs the event stream to prog.mon.

Note that this command-line option can only be used if iconx is run explicitly; it cannot be used with the direct execution of icode files that is supported by UNIX Icon. Explicit use of iconx works on UNIX systems, however.

Appendix B — Event Contexts

The Evaluation Context

The evaluation context contains most of the events that are directly related to program execution. These include procedure-level events (such as calls and returns), expression-level events (such as function and operator invocation), and some implementation events (such as virtual-machine instructions and storage allocation).

At present many such events are instrumented, and instrumentation is being added for others.

Garbage Collection Contexts

There are five contexts related to garbage collection: collection proper, verification, marking, redrawing, and refreshing. These contexts are somewhat arbitrary and are cast in a form that makes it easy to produce allocation history files from event streams.

The Symbol Table Context

Procedures and functions vary from program to program. The names of procedures and functions are needed by some visualization tools. To reduce the size of event streams, an integer is associated with each name and these integers are used in place of the names. The associations between names and integers is given in a symbol-table context that is produced at the beginning of every event stream. Symbol table information is given in event pairs, the first being the identifying integer and the second being the name. See the example event stream given in Appendix D.

Other symbol-table information may be added in the future.

Appendix C — Event Values and Codes

Event codes and some event values are represented symbolically, both as manifest constants in the C code for the instrumentation and as corresponding global variables in Icon procedures that process event streams. It is not necessary to know the actual values of these symbols; in fact they may change as the instrumentation evolves. (The event codes are defined in src/h/monitor.h in the source code for Icon.)

Event Values

Many values are simply numerical quantities such as the number of bytes allocated for an allocation event. A few kinds of events (such as symbol-table ones) have string values. String values are enclosed in quotation marks.

Two kinds of values encode fixed information: event contexts and the types of Icon source-language values.

The symbols for contexts are:

symbol	*context*
C_Collect	garbage collection
C_Eval	evaluation
C_Mark	marking allocated data
C_Redraw	redrawing allocated data information
C_Refresh	refreshing allocated data information
C_Symbols	symbol table
C_Verify	verifying allocated data information

Type codes are based on the the ones used in Icon descriptors. Since those type codes start at zero and there is no type code in a string descriptor (qualifier), the values for types in event stream are one greater than those in descriptors and the code for strings is 0. The symbols for the types are:

symbol	*type*
T_Bignum	large integer
T_Coexpr	co–expression
T_Cset	cset
T_External	external block
T_File	file
T_Integer	integer
T_Lelem	list–element block
T_List	list
T_Null	null
T_Proc	procedure
T_Real	real number
T_Record	record
T_Refresh	refresh block
T_Selem	set–element block
T_Set	set
T_Slots	hash header block
T_String	string
T_Table	table
T_Telem	table element
T_Tvsubs	substring trapped variable
T_Tvtbl	table–element trapped variable

Comments

A line beginning with a # is considered to be a comment. This is an exception to the usual value/code event syntax and is an instance where white space (the previous and following linefeeds) is required.

A line beginning with a ; is also considered to be a comment. Such lines result from calls of mmpause().

Event Codes

An alphabetical listing of the event code symbols follows, together with brief explanations of the events and the kinds of value they have.

symbol	*event*	*value*
E_Alien	show alien block	bytes
E_Base	show base of regions	bytes
E_Bignum	allocate large integer	bytes
E_Bsusp	suspend from built-in	code for built-in
E_Coact	activate co-expression	co-expression serial number
E_Coexpr	allocate co-expression	bytes
E_Cofail	fail from co-expression	co-expression serial number
E_Collect	start garbage collections	region number
E_Colm	change source column	column number
E_Comment	comment	comment text string
E_Coret	return from co-expression	co-expression serial number
E_Cset	allocate cset	bytes
E_Cvcset	convert to cset	input type
E_Cvint	convert to integer	input type
E_Cvnum	convert to numeric	input type
E_Cvreal	convert to real	input type
E_Cvstr	convert to string	input type
E_Ecall	call built-in	code for built-in
E_Efail	fail from built-in	none
E_End	normal program termination	none
E_Eresum	resume built-in	code for built-in
E_Eret	return from built-in	return type code
E_Error	error termination	error code
E_Esusp	suspend from built-in	return type code
E_Exit	exit termination	exit code
E_External	allocate external block	bytes
E_File	allocate file block	bytes
E_Free	show free region	bytes
E_Highlight	highlight object	highlight string
E_Lelem	allocate list-element block	bytes
E_Line	change source line	line number
E_List	allocate list block	bytes
E_Lsusp	suspend from limitation	return type code
E_Offset	note memory offset	bytes
E_Opcode	interpret virtual-machine code	code number
E_Pause	pause memory monitoring	pause text string
E_Pcall	call procedure	procedure code
E_Pfail	fail from procedure	none
E_Pid	procedure identification	procedure number
E_Presum	resume procedure	procedure code
E_Pret	return from procedure	return type code

E_Psusp	suspend from procedure	return type code
E_Pvan	vanquish procedure frame	procedure code
E_Real	allocate real number block	bytes
E_Record	allocate record block	bytes
E_Refresh	allocate refresh block	bytes
E_Region	storage region	region number
E_Selem	allocate set-element block	bytes
E_Set	allocate set block	bytes
E_Size	show region size	bytes
E_Slots	allocate hash block	bytes
E_Start	start new context	context number
E_String	allocate string	bytes
E_Sym	symbol table entry	procedure name
E_Table	allocate table block	bytes
E_Telem	allocate table-element block	bytes
E_Tick	clock tick	number of ticks*
E_Tvsubs	allocate substring block	bytes
E_Tvtbl	allocate table-element block	bytes
E_Used	show space used in region	bytes

*Clock ticks are based on the UNIX system clock, which typically ticks every 4 to 20 milliseconds; on an Sun 4, it ticks every 10 milliseconds.

Appendix D — An Example Event Stream

The event stream from a short program follows. Event streams are not intended to be readable; this example is included only to give an idea of the nature of event streams and the encodings used. See the other appendices for detailed information about event streams and for procedures to process them.

The program that produced the following event stream is:

```
    procedure main()
       every write(&features)
       every write(image(main,2.0 + 0,&lcase ++ &ucase,[]))
    end
```

The resulting event stream is:

```
## Icon event stream, Version 8.4.000
#
# program: example
# date:    Sun Aug 25 08:04:48 1991
2(
1."main"T
2."write"T
3."image"T
2)
1(
6(259772<60000=60000>319808<0=65024>384832<0=65024>6)
3(
2+10000x10004+202z10208+8z10218+2z10222+z10226+4z10232+2z10236+2050z12288+2712Z
3)
7(259772<60000=60000>319808<0=65024>384832<0=65024>7)
12288+256z5k23m1.61O1C64O2_108O11|67O85O84O64O3_108O16|62O0b64O108O
15|61O61c12546+34z0r70O53Of0ub64O108O61Ocr70O53Ofub64O108O61Ocr70O53Of
ub64O108O61Ocr70O53Ofub64O108O61Ocr70O53Ofub64O108O61Ocr70O53Ofub64O108O
61Ocr70O53Ofub64O108O61Ocr70O53Ofub64O108O61Ocr70O53Ofub64O108O61Ocr70O
53Ofub64O108O61Ocr70O53Ofub64O108O61Ocr70O53Ofub64O108O61Ocr70O53Ofub64O
108O61Ocr70O53Ofub64O108O61Ocr70O53Ofub64O108O61Ocr70O53Ofuff67O85O84O
OO69O75O60O64O4_108O31|30O30c4d4r69O64O108O35|62O5r64O108O45|62Or64O108O
42|42O42c10er69O64O108O52|65Okm9r64O108O21|61O61c10s4s0r64O108O15|61Oc
r70O53Off69O64O5_108O1|68O1Ff86O0X
7(259772<60000=60000>319808<14=65024>384832<280=65024>7)
1)
# Normal Exit
```

Appendix E — Processing Event Streams

The facilities here are designed to simplify the processing of event streams. These facilities consist of an Icon procedure and a set of functions that is built into versions of iconx that are compiled with event monitoring enabled.

There also is a package of Icon procedures that can be used in place of the built-in functions.

Efficiency in processing event streams is a prime concern in visualization. The built-in functions are much faster than the Icon procedures and should be used where possible.

Values related to event streams are communicated via (Icon) global identifiers for efficiency. See Appendix C for a list of the global identifier names.

Initialization

The package is initialized by the Icon procedure EvInit(f), where f is an Icon file that specifies the event stream. It may be a file that has been opened as a pipe. The default is standard input.

EvInit() sets the values of global identifiers corresponding to the names of event codes and contexts. EvInit() also creates two tables:

- ProcName, which maps integer values that encode procedures and functions to the names of the corresponding procedures and functions.
- AllocMap, which maps allocation event codes to the corresponding MemMon allocation codes.

The procedure EvInit() is in a library file that can be linked as follows:

```
link evinit
```

If the procedure package is used in place of the built-in functions, it can be linked as follows:

```
link evprocs
```

The ucode files for evinit and evprocs must be accessible through IPATH.

Programs that link evinit must be compiled with –SI200.

Context Selection

The speed of processing an event stream can be improved by limiting processing to selected contexts, which is done by

```
EvSelect(i1, i2, ..., in)
```

where i1, i2, ..., in are the desired contexts. Contexts should be specified by global identifiers corresponding to the context names that are provided by EvInit().

If EvSelect() is called with no arguments, all contexts are selected.

EvSelect() can be called as needed to change the selected contexts.

Producing Events

The function

```
EvGet(c)
```

reads the event stream, processing the currently selected contexts and returns when an event whose code is in the cset c is encountered. The characters in c correspond to the event codes of interest. If c is omitted, all events are processed.

EvGet() sets the values of four (Icon) global variables before it returns:

EvCode	The event code (a one-character string)

EvContext	The current event context (a positive integer)
EvValue	The event value (an integer or string)
EvGivenValue	The event value as it actually appears in the event stream; it is null if the value for the event is not explicitly specified in the event stream.

EvGet() returns the value of EvCode but fails when the event stream is exhausted.

EvGet() can be called with different arguments as desired; the argument applies only to the current call.

An Example

The following tool prints a summary of procedure events:

```
link evinit

procedure main()
   local codes, ProAct

   EvInit()

   ProAct := table(0)

   EvSelect(C_Eval)                    # procedure events only occur in C_Eval

   codes := cset(E_Pcall ++ E_Presum ++ E_Pfail ++ E_Pret ++
      E_Psusp ++ E_Pvan)

   while EvGet(codes) do
      ProAct[EvCode] +:= 1

   write("procedure calls:        ",   right(ProAct[E_Pcall],5))
   write("procedure resumptions: ",    right(ProAct[E_Presum],5))
   write("procedure returns:      ",   right(ProAct[E_Pret],5))
   write("procedure suspensions: ",    right(ProAct[E_Psusp],5))
   write("procedure failures:     ",   right(ProAct[E_Pfail],5))
   write("procedure removals:     ",   right(ProAct[E_Pvan],5))

end
```