The MT Icon Interpreter [*]

Clinton L. Jeffery

November 2, 1993

Abstract

MT Icon is an Icon language interpreter that supports multiple tasks, where a task is the execution state of a program within the Icon virtual machine. MT Icon includes language extensions that allow Icon programs to load, execute, communicate with, and control one another, all within a single instantiation of the Icon interpreter. This document describes the language extensions and provides examples of their use.

1 Introduction

MT (Multi-Tasking) Icon is an enhanced version of the Icon interpreter that allows Icon programs to load and execute other Icon programs within the same interpreter space. MT Icon is *not* a concurrent programming language nor does it include special support for multiprocessor hardware. Its domain is that of high-level language support for programs that benefit from or require a tighter coupling than that provided by inter-process communication; that is, programs that access each other's state extensively.

Icon co-expressions provide the MT interpreter's program execution model, and co-expression activation serves as the communication mechanism. The extensions are general enough to be useful in a wide variety of contexts. For example, programs that use the multi-tasking interface can communicate directly without resorting to external files or pipes.

Icon programs do not need to be aware of the MT Icon extensions. Version 8.10 [Gris93] Icon programs run in the multi-tasking interpreter exactly as they do in the standard Icon interpreter. Their interaction with other programs is entirely passive. In addition to its general multi-tasking execution model, MT Icon has features specific to the control and monitoring of loaded tasks by the *parent* task that loads them. These monitoring features are described in [Gris92].

[*]This work was supported in part by the National Science Foundation under Grant CCR-8713690 and a grant from the AT&T Research Foundation.

At the language level, the extension involves the addition of several built-in functions and keywords, but no new types, declarations, or control structures. Several existing functions have been extended to offer additional support for the multi-tasking environment. The interpreter creates separate memory allocation regions for each task. The remainder of this paper describes these extensions and their use.

2 MT Icon preliminary terminology

Before describing the MT Icon task model, a few definitions are needed. These definitions pertain to regions of memory referenced by programs during execution.

Name spaces

A *name space* is a mapping from a set of program source-code identifiers to a set of associated memory locations [Abel85]. Icon programs have a global name space shared across the entire program and various name spaces associated with procedures. Procedures each have a static name space consisting of memory locations shared by all invocations of the procedure and local name spaces private to each individual invocation of the procedure.

When a co-expression is created, a new local name space is allocated for the currently executing procedure, and the current values of the local variables are copied into the new name space for subsequent use by the co-expression.

Program and co-expression state

An Icon program has an associated *program state* consisting of the memory associated with global and static name spaces, keywords, and dynamic memory regions. Similarly, a co-expression has an associated *co-expression state* consisting of an evaluation stack that contains the memory used to implement one or more local name spaces. Co-expressions in an Icon program share access to the program state and can use it to communicate.

3 Tasks: an extended co-expression model

The central concept in MT Icon is the *task*; a task is the execution state of a program within the Icon virtual machine [Gris86]. A single task called the *root* is created when the interpreter starts execution. Additional tasks can be created dynamically as needed.

A task consists of a main co-expression and zero or more child co-expressions that share a program state. At the source-language level, tasks are loaded, referenced, and activated solely in terms of one of their member co-expressions; the task itself is implicit.

This definition of tasks is related to the concept of the same name commonly used in operating systems and concurrent programming languages. It differs, however, in certain fundamental respects. Icon is a sequential language; co-expressions in Icon provide a synchronous coroutine execution model, not a concurrent execution model with implicit task switching and scheduling. Another way to view this is that unlike other languages such as Ada, MT Icon provides the task model as a mechanism for multi-tasking, but does not predefine the policy; matters such as the scheduling algorithm used and whether multi-tasking is co-operative or pre-emptive are programmable at the user level.

Another useful comparison can be made between Icon tasks and Smalltalk processes. Both provide pseudo-concurrency within the context of a sequential virtual machine. Since Icon tasks have their own dynamic memory regions, their presence affects each other less than Smalltalk processes affect each other. For example, if one task is exhibiting thrashing heap behavior in which garbage collections are frequent, the other tasks in the system can execute at full speed during the portion of time in which they are running, since they do not allocate memory out of the thrashing task's (full) heap. This minimal effect of tasks on each others' behavior is especially important in the domain of execution monitoring.

4 Task creation

In MT Icon, a task can create other tasks. The MT Icon function

```
load(s, L, f1, f2, f3, i1, i2, i3)
```

loads an *icode* file [Gris86] specified by the file name s, creates a task for it and returns a co-expression corresponding to the invocation of the procedure main(L) in the loaded icode file. L defaults to the empty list. Unlike conventional Icon command-line argument lists, the argument list passed to load() can contain values of any type, such as procedures, lists, and tables in the calling task.

The task being loaded is termed the *child* task, while the task calling load() is termed the *parent*. The collection of all tasks forms a tree of parent-child relationships.

f1, f2, and f3 are three file arguments to use as &input, &output, and &error in the loaded task; &input, &output, and &error default to those of the loading task. i1, i2, and i3 are three integer arguments that supply initial region sizes for the task's block, string, and stack memory areas, respectively. i1 and i2 default to 65000, while i3 defaults to 20000 (the defaults may be changed by the environment variables BLKSIZE, STRSIZE, and MSTKSIZE).

5 Running other programs

A co-expression created by load() is activated like any other co-expression. When activated with the @ operator, the child task begins executing its main procedure. Unless it suspends or activates &source, the child task runs to completion, after which control is returned to the parent. Chapter 5 presents an alternative means of executing a child with which the parent retains control over the child as it executes.

An example

This default behavior is illustrated by the program seqload, which loads and executes each of its arguments (string names of executable Icon programs) in turn. In this program the variable arguments is a list of strings passed into the Icon program from the operating system. Each of these strings (extracted from the list using the element-generation operator, !) is passed in turn to load(). load() reads the code for each argument and creates a task in which to execute the loaded program; the tasks are then executed one-by-one by the co-expression activation operator, @. This is ordinary Icon code; there is nothing special about this example except the semantics of the load() function and the independent execution environment (separate global variables, heaps, and so forth), that load() provides to each task.

```
# seqload.icn
procedure main(arguments)
    every @load(!arguments)
end
```

For example, if three Icon programs whose executable files are named translate, assemble, and link are to be run in succession, the command

```
seqload translate assemble link
```

executes the three programs without reloading the interpreter for each program.

6 Data access

Although tasks have separate program states, they reside in the same address space and can share data; values can be transmitted from task to task through main()'s argument list, through co-expression activation, or by use of event monitoring facilities described in [Gris92]. In the following pair of programs, the parent receives a list value from the child and writes its elements out in reverse order.

```
# parent.icn
procedure main()
    L := @ load("child")
    while write(pull(L))
end

# child.icn
procedure main()
    L := [ ]
    while put(L, read())
    return L
end
```

This data access applies to all first-class data objects in Icon, such as procedures and co-expressions.

7 Shared icode libraries

Programs that are written to take advantage of the multi-tasking environment gain in space efficiency and modularity. Code sharing is one natural way to achieve space efficiency in a collection of programs. Since procedures are first-class data values in Icon, code sharing can be implemented via data sharing. Programs executing in a single invocation of the interpreter can share code easily if the code is not required to produce side effects on global variables in the *calling* task's program state. If side-effects to the calling task's program state are required, the shared code must generally be written with care to explicitly reference the calling task's state. Side effects in the client task can also be achieved through the parameters passed in and results obtained by calling the shared procedure.

7.1 Loading shared code

Consider a collection of applications that make extensive use of procedures found in the Icon program library (IPL) [Gris90]. If those applications are run using MT Icon, the IPL routines need be loaded only once, after which they may be shared.

In order to reference shared code from a loaded task, two additional considerations must be satisfied: the shared code must be loaded, and the client tasks must be able to *dynamically link* shared routines into their generated code.

Both of these problems can be solved entirely at the source level: In order to introduce a shared Icon procedure into the name space, a global variable of the same name must be declared. Managing the loading of shared libraries is itself a natural task to assign to an Icon procedure that uses a table to map strings to the pointers to the procedures in question.

7.2 Code sharing example

The following collection of three programs illustrate one schema that allows code sharing. Other conventions can certainly be devised, and much of the sharing infrastructure presented here can be automatically generated. Program calc.icn consists of a shared library procedure named calc() and a main procedure that exports a reference to calc() for sharing:

```
# calc.icn
procedure calc(args...)
   # code for calc
   # (may call other routines in calc.icn if there are any)
end

procedure main()
   # initialization code, if any
   return calc
end
```

Note that a module exporting shared procedures can also have global variables (possibly initialized from other command-line arguments). Shared modules can export other values besides procedures using the same mechanism.

The parent task that loads the various shared library clients implements a procedural encapsulation (loadlib() in this example) of an Icon table to store references to shared routines. The parent passes this procedure to clients. Each client calls the procedure for each shared routine. Routines that are already loaded are returned to requesting tasks after a simple Icon table lookup. Whenever a routine is requested that has not been loaded, the load() function is called and the shared library activated.

```
procedure main(arguments)
    @load("client",put(arguments,loadlib))
end
procedure loadlib(s, C)
    static sharedlib
    initial sharedlib := table()
    /sharedlib[s] := @load(s)
    variable(s, C) := sharedlib[s]
end
```

A client of calc declares a global variable named calc, and assigns its value after inspecting its argument list to find the shared library loader:

```
global loadlib
global calc
procedure main(arguments)
    if /loadlib then stop("no shared libraries present")
    loadlib("calc", &current)
    # ... remainder of program may call shared calc
end
```

7.3 Sharing procedure collections

The primary deficiency of the previous example is that it requires one shared library procedure per Icon module, that is, separate compilation. In practice it is more convenient to have a collection of related procedures in a given Icon compilation unit. Shared libraries can employ such a mechanism by resorting to a simple database that maps procedure names to load modules.

8 Extended and new Icon functions

Several of Icon's standard functions are extended in MT Icon. In each case the extension consists of the addition of an optional co-expression argument to specify the task to which the function is to apply. The co-expression argument in all cases defaults to &current.

In addition to these extensions, MT Icon provides some entirely new functions for accessing simple aspects of an Icon program's state. These features are useful in program execution monitors such as debuggers.

cofail(C) : n — transmit failure

cofail(C) activates C and transmits a failure to it. It returns the transmitted value from the current co-expression's next activator, similar to @C.

Default:	C	&source
Error:	118	C not co-expression

display(i,f,C) : n — display variables

display(i,f,C) writes the image of the supplied co-expression and the values of the local variables of the i most recent procedure activations within C to file f.

Defaults:	i	&level
	f	&errout
	C	&current
Error:	118	C not co-expression

fieldnames(R) : s — generate field names

fieldnames(R) generates the names of the fields in record R in the order in which they appear in R's record declaration.

Error:	107	R not record

globalnames(C) : s — generate global variable names

globalnames(C) generates the names of global identifiers in the task that contains co-expression C.

Default:	C	&current
Error:	118	C not co-expression

keyword(s, C) : s — produce keyword

keyword(s, C) produces the keyword named s in the task that contains C. The string name does *not* include the ampersand character (&). This function is supported for keywords whose values vary from task to task, summarized in the following table:

&allocated	&errornumber	&eventcode	&input	&pos	&source
&collections	&errortext	&eventsource	&line	&progname	&storage
&column	&errorvalue	&eventvalue	&main	&random	&subject
&error	&errout	&file	&output	&regions	&trace

On systems with X-Icon facilities, the function also supports &window, &col, &row, &x, and &y. Note that &level and &time are not yet supported; this is a bug. keyword() fails on keywords such as &cset that are constants or are the same in all tasks. keyword() produces a variable if the corresponding keyword may be assigned to.

Default:	C	&current
Error:	118	C not co-expression

load(s,L,f1,f2,f3,i1,i2,i3) : C — load task

load(s,L,f1,f2,f3,i1,i2,i3) loads an icode file specified by file name s, creates a task for it, and returns a co-expression corresponding to the invocation of the procedure main(L) in the loaded task. f1, f2, and f3 are the child task standard input, output, and error files. If f1, f2, or f3 is closed by the parent or the child and subsequently used in the other task, dire consequences will result as the closure is not reflected in the other task. i1, i2, and i3 are the child task block region, string region, and stack sizes.

Defaults:	L	[ ]
	f1	caller's &input
	f2	caller's &output
	f3	caller's &error
	i1	65000 or BLKSIZE
	i2	65000 or STRSIZE
	i3	10000 or MSTKSIZE
Errors:	101	i1, i2, or i3 not integer
	103	s not string
	105	f1, f2, or f3 not file
	108	L not list

localnames(x, i) : s — generate local variable names

localnames(p) generates the names of the local variables of procedure p. localnames(C, i) generates the local identifiers i levels above the currently active procedure in co-expression C. localnames() fails if i is larger than the level of the current procedure activation within C.

Defaults:	C	&current
	i	0
Errors:	118	C not co-expression
	101	i not integer
	205	i less than zero

name(x,C) — produce variable name

name(x,C) produces the name of variable x. name() fails if x is not a variable from the task that contains co-expression C.

Default:	C	&current
Error:	118	C not co-expression

paramnames(x, i) : s — generate parameter names

paramnames(p) generates the names of the parameters of procedure p. paramnames(C, i) generates the names of the parameters i levels above the currently active procedure in co-expression C. paramnames() fails if i is larger than the level of the current procedure activation within C.

Defaults:	C	&current
	i	0
Errors:	118	C not co-expression
	101	i not integer
	205	i less than zero

parent(C) : C — produce parent's &main

parent(C) produces the parent's &main with respect to the task that contains co-expression C. parent(C) fails if C is part of the root task.

Default:	C	&current
Error:	118	C not co-expression

proc(x, i, C) : p — convert to procedure

proc(x, i, C) produces the procedure named s within the task that contains C.

Default:	C	&current
Errors:	101	i not integer
	118	C not co-expression

---

staticnames(x, i): s — generate static variable names

staticnames(p) generates the names of the static variables of procedure p. staticnames(C, i) generates the names of all the static variables in the currently active procedure in the co-expression argument C. staticnames() fails if i is larger than the level of the current procedure activation within C.

Defaults:	C	&current
	i	0
Errors:	118	C not co-expression
	101	i not integer
	205	i less than zero

---

variable(s, C, i): x — produce variable

variable(s, C, i) produces the variable for the identifier or keyword named s within the task that created the co-expression argument C, examining local and static scopes i levels above the currently active procedure in C, and then checking for a global variable or keyword if no local or static variable by that name is found. variable() fails if i is larger than the level of the current procedure activation within C.

Defaults:	C	&current
	i	0
Errors:	118	C not co-expression
	101	i not integer
	205	i less than zero

9 &column

A new keyword, &column, produces the current source column at which execution is taking place. &column is analogous to &line.

References

[Abel85] Abelson, H. and Sussman, G. J. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Massachusetts, 1985.

[Gris86] Griswold, R. E. and Griswold, M. T. *The Implementation of the Icon Programming Language*. Princeton University Press, Princeton, New Jersey, 1986.

[Gris90] Griswold, R. E. *The Icon Program Library; Version 8.1*. Icon Project Document 172, Department of Computer Science, University of Arizona, 1990.

[Gris92] Griswold, R. E. and Jeffery, C. L. *Writing Execution Monitors for Icon Programs*. Icon Project Document 192, Department of Computer Science, University of Arizona, 1992.

[Gris93] Griswold, R. E., Jeffery, C. L., and Townsend, G. M. *Version 8.10 of the Icon Programming Language*. Icon Project Document 212, Department of Computer Science, University of Arizona, 1993.