Porting the UNIX Implementation of Icon*

William H. Mitchell

TR 83-10d

ABSTRACT

This document explains how to port the UNIX implementaton of the Icon programming language. The Icon system is composed of a translator, a linker, and an interpreter. Procedures for porting each system component are described in detail. This document is meant to be a companion to the Icon "tour" (TR 84-11) and the source code for the system.

June 1983; Revised July 1983, January 1984, June 1984, and August 1984

Corrected January 23, 1985

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported by the National Science Foundation under Grants MCS81-01916 and DCR-8401831.

Porting the UNIX Implementation of Icon

Introduction

This document describes how to port the Version 5 Icon interpreter to a UNIX* environment. There is both an interpreter and a compiler available for Icon; this document only addresses porting the interpreter. The Icon system has three major components: a translator, a linker, and an interpreter. The translator and the linker are entirely written in C and porting them is merely a matter of setting constant values that are appropriate for the target machine. Portions of the interpreter are written in assembly language and thus must be written anew for each machine. The interpreter also contains a very small amount of C code that must be written on a per-machine basis.

The sections of this document that describe the porting of the translator and the linker are straightforward, being merely a description of a process. While porting the translator and the linker is a task of following instructions, porting the interpreter is a task of design and programming. The approach taken is to describe what function each routine must perform and how it is implemented in the VAX† version of Icon. The porter's job is to determine how to implement the various routines on the target machine.

In light of the increasing popularity of the C language and the availability of C compilers for non-UNIX environments, it is quite possible that one may desire to port Icon to a non-UNIX environment. Because the matter of porting a UNIX program to a non-UNIX environment is a problem in itself, it is not addressed in this document. Rather, this document assumes that the target environment is UNIX. This is not to say that porting Icon to a non-UNIX environment is not feasible. Icon is not strongly bound to UNIX, the primary association being that Icon is written in C. It is anticipated that most C systems that are available for a non-UNIX environment will provide most of the UNIX-independent C standard functions as part of a library. If such a library is available, it should be possible to port Icon without great difficulty.

This document is a companion document of the Icon "tour"[1] and should be studied with the source code for Version 5.9 of Icon at hand. In particular, the porter should be familiar with the information contained in the tour.

The sections of this document that describe the VAX assembly language code attempt to explain the operation of instructions when the operation is not obvious. However, this document does assume that the porter has a rudimentary familiarity with the basic concepts of the VAX-11 architecture[2].

C Compiler Requirements

Because there is no standard for the C programming language, it is difficult to say how "standard" the usage of C in the system is. The system was developed using the V7 C compiler, often referred to as the Ritchie compiler[3]. The system was later ported to the VAX using the *Portable C Compiler*[4] and no serious problems were encountered.

In addition to supporting "full" C, a few specific requirements and non-requirements are made on the C compiler:

(1) The compiler must support both assignment and call-by-value for structures.

(2) The compiler need not support bit field operations.

(3) Arguments to C functions must be stored in consecutive, ascending memory locations.

*UNIX is a trademark of AT&T Bell Laboratories.
†VAX is a trademark of Digital Equipment Corporation.

(4) There may be problems if sizeof(int) and sizeof(char *) are not the same, but no definite problems are known.

(5) It is believed that there are great, perhaps insurmountable problems, if sizeof(char *) is not equal to sizeof(int *).

System Testing

The test programs and testing procedures to be used for porting Icon are described in [5]. At various points in this document, the porter is directed to test the system component just completed. At such times, the porter should refer to [5] to determine what should be done.

1. Porting the Icon Translator

1.1 Overview

The Icon translator, known as itran, is the first logical component of the Icon system. The translator takes Icon source files as input and produces two *ucode* output files for each input file. The translator may be run by saying:

```
itran hello.icn
```

This produces two ascii files, hello.u1 and hello.u2. hello.u1 contains interpretable instructions and data in a printable format. hello.u2 contains information about global symbols and scope.

The translator is written entirely in C and is the most machine independent major system component. No serious problems should be encountered in porting it. If difficulties are encountered, it probably indicates that there are major problems with the C compiler being used.

1.2 Porting Procedure

The Icon system contains a number of instances of values that must be specified on a per-machine basis. The system also contains assembly code and, of course, such code is different on each machine. Rather than maintaining a source copy of Icon for each machine that Icon runs on, C preprocessor control statements are used to select portions of code specific to a certain machine. The source as distributed can be compiled on either a VAX and PDP-11* system by defining VAX or PDP11 respectively in h/config.h. The porting source has neither VAX or PDP11 defined; rather, PORT is defined. Where machine specific code is to appear, along with sections bracketed by #defines for VAX and PDP11, there is a skeletal section bracketed by a #define for PORT. The PORT section is to be filled out for the target machine. This convention is followed throughout and porting Icon is nothing more than filling in all the PORT sections.

The source for the translator is contained in the directory tran. Translator machine dependencies are confined to the file tran/sym.h. A pair of constants define the sizes of two data structures used during the translation process. Edit the file sym.h and search for the string PORT. The code looks something like

*PDP is a trademark of Digital Equipment Corporation.

```
#ifdef PORT
#define TSIZE          x       /* default size of parse tree space */
#define SSIZE          x       /* default size of string space */
#endif PORT
#ifdef VAX
#define TSIZE      15000       /* size of parse tree space */
#define SSIZE      15000       /* default size of string space */
#endif VAX
#ifdef PDP11
#define TSIZE       5000       /* default size of parse tree space */
#define SSIZE       5000       /* default size of string space */
#endif PDP11
```

The values of TSIZE and ssize are not critical and current values have been chosen rather arbitrarily. If you are on a large machine, use the values of TSIZE and SSIZE specified for the VAX; otherwise, use the values specified for the PDP-11.

The translator may now be compiled by issuing the *make* command without any arguments.

It should be noted that although Icon programs are used to create some of the translator source files (namely keyword.h, keyword.c optab.c, and toktab.c). These files are machine independent and do not need to be remade. If for some reason make tries to create any of these files, just touch the file in question to update the last-modified date. Similarly, parse.c is generated by *yacc* and does not need to be regenerated unless the grammar is modified.

When the translator has been successfully compiled using *make*, refer to [5] for testing.

Porting the translator may seem like a trivial task, but its successful completion is a definite milestone because it is good sign that the C compiler in use is suitable.

2. Porting the Icon Linker

2.1 Overview

The Icon linker, known as ilink, is the second logical component of the Icon system. The linker takes u1 and u2 files produced by the translator and binds them together to form an *interpretable* file. The interpretable file serves as input for the Icon interpreter. The linker is written entirely in C and is a fairly small and simple program. However, the interpretable files produced by the linker are not machine independent and because of this, porting the linker is more troublesome than porting the translator.

Interpretable files contain two distinct types of data: opcodes and associated operands that the interpreter "understands"; and data that is directly mapped into run-time data structures. By "mapping" it is meant that the data is loaded into memory and then C structure references are used to access elements of the object at a certain location in memory. The formats of the opcodes and operands must conform to what the interpreter is expecting. The data that is directly mapped must conform to the format of the C data structures used by the run-time system.

On the VAX, for example, interpreter opcodes are one byte long and operands are four bytes long. On the PDP-11, opcodes are also one byte long, but operands are only two bytes long. Opcode and operand size are fairly arbitrary, but it is important that the linker and the interpreter be coordinated.

The mapped data structures are slightly more complicated because the linker must conform to the format produced by the C compiler. This is not difficult, since the data structures involved have a regular form. All are composed of some number of *words* where each word is the same size in every structure.*

The opcodes, operands, and mapped data are accumulated in memory during the linking process. This conglomerate is referred to as the *code* section. Several routines are used to add data to the code section.

*Literature about the VAX conventionally uses the term *word* to refer to 16-bit quantities and the term *longword* to refer to 32-bit quantities. In this document, *word* in a generic context refers to the basic unit of the run-time data structures; *word* in a VAX-specific context refers to a 32-bit quantity.

These routines are parameterized so that porting the linker to a new machine is merely a matter of setting the parameters correctly. Four primitive data units compose the code section. These are *opcodes*, *operands*, *words*, and *blocks*.

opcodes
: are instructions for the interpreter. An opcode may direct the interpreter to push a value on the stack, branch to a location, perform an arithmetic operation, etc.

operands
: are associated with some opcodes. For example, the goto instruction has a location to branch to as its single operand.

words
: compose mapped data structures. A word is the basic unit of the run-time data structures and should consist of sizeof(int *) bytes.

blocks
: are merely some number of bytes. For example, a cset constant is loaded into the code section as a block of 32 8-bit bytes (256 bits).

Routines in link/lcode.c are used to add a unit of data of one of the preceding types to the code section. These routines are outop, outopnd, outword, and outblock. Each routine adds the appropriate data into the code section at the current location (maintained as a pointer), and then the location pointer is advanced to the next free location.

2.2 Porting Procedure

Edit ilink.h and search for the string PORT. Define the following constants as described.

INTSIZE
: The number of bits in an int.

LOGINTSIZE
: The base 2 log of INTSIZE. That is, LOGINTSIZE answers the question "*What power of 2 is* INTSIZE?".

LONGS
: Icon has an integer data type. On the VAX and the PDP-11 the range of integer values is -2^{31} to $2^{31}-1$. On the VAX, C ints and longs are both 32 bits wide. On the PDP-11, C ints are 16 bits wide while longs are 32 bits wide. The PDP-11 Icon system makes an internal distinction between integers that "fit" in 16 bits and integers that require 32 bits. The former are stored in two-word descriptors (the actual value being in the second of the two 16-bit words), while the latter have a value descriptor that points to a block in the heap that holds the two-word, 32-bit value. On the other hand, the VAX uses two 32-bit words for descriptors and thus the second word of a descriptor can hold the largest possible integer value used by Icon. Rather than having an internal distinction between integer types on the VAX, integers are always represented by two-word integer descriptors. There are places in the code where special provisions must be made if C ints are not the same size as C longs.

  If sizeof(int) != sizeof(long) for the C compiler in use, define LONGS. (LONGS need not be given a value, #define LONGS is sufficient.) If LONGS must be defined, the minimum and maximum values that can be represented by an int must also be defined. Define MINSHORT to be the smallest value that an int can hold and define MAXSHORT to be the largest value that an int can hold.

MAXCODE
: This is the maximum size in bytes of the code that can be generated for each procedure. This value is not critical; 10,000 is used for the VAX, while 2000 is used for the PDP-11.

strchr and strrchr
: If you are on a USG UNIX system, #define index to be strchr and rindex to be strrchr.

Edit datatype.h and search for the PORT section. This section contains #defines that are used to set and test flags contained in the first word of descriptors. The basic idea in forming these constants is to set some bits at the high end of the word, and set some other bits at the low end. The number of unused bits in the

middle depends on the size of a word.

F_NQUAL, F_VAR, F_TVAR, F_PTR, F_NUM, F_INT, and F_AGGR should be set to mask values with one bit set to 1 in each. For F_NQUAL, the leftmost bit should set, for F_VAR, the next to leftmost bit should be set, and so forth. The values for the VAX and PDP-11 should be suitable for machines with 32-bit and 16-bit words, respectively.

The constants OPSIZE, OPNDSIZE, and WORDSIZE control the sizes of opcodes, operands, and words in the code section. Before setting these constants to values appropriate for the target machine, a "standard" linker should be built and tested using the supplied values (under PORT) for these constants. This allows the linker to be checked against output files that are known to be correct. The purpose of this is to attempt to discover C compiler problems. Compile the linker using make and refer to [5] for the testing procedure.

Once the "standard" linker has been checked out, the following "sizing" parameters in ilink.h should be set to values appropriate for the target machine.

OPSIZE

This is the size in bytes of interpreter opcodes. The interpreter treats opcodes as unsigned quantities. One byte (8 bits) is currently large enough to accommodate all opcodes and a value of 1 is recommended for OPSIZE. The outop routine in lcode.c assumes that opcodes are one byte. If a larger size is desired, outop will have to be recoded. It might be wise to use a value other than 1 for OPSIZE on machines that are not byte-addressable and have ample memory.

OPNDSIZE

This is the size in bytes of operands for interpreter instructions. For some instructions, the operand value represents an offset from the interpreter program counter and thus, the maximum possible offset is limited by the magnitude of values that can be represented in OPNDSIZE bytes. Because larger operands occupy more code space and smaller operands limit addressing "distance", a trade-off is involved. On the VAX, operands are four bytes because memory space is not very critical. On the PDP-11, operands are two bytes because of the limited memory. While it is easy to change the value of OPNDSIZE in the linker, the operand size is pervasive in the interpreter. If the target machine has a large, perhaps virtual address space, use a value such as 4 for OPNDSIZE. A value such as 2 may be appropriate for a smaller machine. A value of 1 is not advisable under any circumstances. The suggested value for OPNDSIZE is sizeof(int).

WORDSIZE

This should be set to sizeof(int *) on the target machine. The various run-time data structures are all composed of a number of words each of which contain WORDSIZE bytes. For example, the data blocks for user-defined procedures are built in the code section by a sequence of calls to outword.

The backpatch routine in lcode.c needs some machine-specific modifications. This routine backpatches forward references to ucode labels. In the *while* loop, the operand (which is OPNDSIZE bytes long) that is pointed at by q is loaded into the variable p. Then, the operand is replaced by the value of r. On the VAX, this can be expressed as:

```
p = *q;
*q = r;
```

where q is an int *. This is possible because the VAX allows word references on an arbitrary boundary. On the PDP-11, such references are illegal and the assignments must be made on a byte-wise basis. If the target machine allows word accesses on arbitrary boundaries, the VAX code may be used (assuming OPNDSIZE is equal to sizeof(int)). If not, but operands are the same size as ints, the PDP-11 code may be used. Other situations may require ingenuity. Be sure to alter the first PORT section in backpatch to contain an appropriate declaration for q (that section currently contains a declaration for q and a return).

When the linker has been compiled, refer to [5] for directions on testing.

3. Porting the Icon Interpreter

3.1 Introduction

The Icon interpreter, known as *iconx*, is the third major logical component of the system. The interpreter takes interpretable files produced by the linker and "executes" them. The interpreter is run by:

```
iconx hello
```

where hello has been produced by the linker.

Due to the stack manipulations that the interpreter performs, it is necessary for a small portion of the interpreter to be written in assembly language rather than in C. On the VAX, about 550 lines of assembly instructions are required. The coding of these assembly instructions is the most difficult part of the port.

3.2 Source File Layout

The interpreter is divided into four parts:

start-up code
the main loop
primary subroutines
support subroutines

The start-up code initializes the interpreter and passes control to the main loop. The main loop, referred to as interp, fetches interpreter instructions and executes them. An interpreter instruction may be entirely performed by interp or interp may call a *primary subroutine* to perform the operation. In turn, a primary subroutine may call a number of *support subroutines*. Each primary subroutine has a direct correspondence to a source language operation of some type or to a stack manipulation.

While the translator and linker sources files are in their own directories, the interpreter source files are segregated into several directories.

iconx The start-up code and the main interpreter loop reside in this directory. Files of particular interest are: start.s, which is entered when the interpreter is run and does some low-level initialization; init.c, which is called from start.s and completes initialization of the interpreter; and interp.s, which is the interpreter loop itself.

functions This directory contains code for the built-in procedures. For example, write.c contains the source for the write function. The source for each built-in procedure appears in a file of its own.

operators This directory contains code for the Icon operators. The routines in this directory implement the various Icon source level operators. For example, plus.c is called to perform the + (addition) operation, and bang.c is called to perform the ! (element generation) operation. As with the built-in procedures, there is one operator per file.

lib This directory contains routines that do not fit anywhere else. First of all, there is code for routines that perform actions similar in nature to those in functions and operators, but that do not have a functional or operator syntax. For example, llist.c creates a list that is specified syntactically as $[arg_0, arg_1, \ldots, arg_n]$, and field.c handles record element accesses that arise from $arg_1.arg_2$.

lib also contains routines such as esusp.s and efail.s that handle stack manipulations during expression evaluation. The routines pret.s and pfail.s handle procedure return and failure respectively.

The directories functions, operators, and lib compose the primary subroutines mentioned above.

rt The support subroutines are contained in the rt directory. The primary subroutines are autonomous with respect to each other and use the rt routines for common operations. For example, cvstr.c is used to convert a value to a string, trace.c produces various types of tracing messages, and gc.c is the garbage collector.

h — This directory contains a number of header files that are #included in the other files that compose the interpreter. Of particular interest is rt.h, which defines a number of constants and data structures.

3.3 Overview of the Porting Process

The following steps are to be followed when porting the interpreter.

(1) Determination of layout of procedure, generator, and expression markers and associated frame pointers.

(2) Setting of implementation specific constants in h/rt.h and creation of h/defs.s from rt.h.

(3) Complete system compilation.

(4) Coding of a "basis" of routines for the interpreter, consisting of iconx/start.s, rt/setbound.s, lib/invoke.s, iconx/interp.s, lib/efail.s, lib/pfail.s.

(5) Testing of the basis routines for the interpreter.

(6) Coding and testing of

rt/arith.s
rt/fail.s
lib/pret.s
lib/esusp.s
lib/lsusp.s
lib/psusp.s
rt/suspend.s
functions/display.c

in an incremental fashion. Test programs are provided to test the system after adding each routine.

(7) Coding of rt/gcollect.s and rt/sweep.c. Testing of garbage collection.

(8) Complete system testing.

This document does not explain how to port the sections of the system that are related to co-expressions. The involved files are lib/coact.s, lib/cofail.s, lib/coret.s, lib/create.c, and operators/refresh.c. Icon works properly with these sections of code left unimplemented, provided no attempt is made to use co-expressions, in which case the system notes it as a fatal error.

3.4 Porting Procedure

Determination of Frame Layouts

Unfortunately, one of the most far-reaching decisions that must be made during the porting process is also one of the first decisions that must be made. The decision (actually, a number of decisions) is how to layout the procedure, generator, and expression frames and what registers should be used as frame pointers. The various frames and their usages are explained in detail in [1] and the portions of this document that describe routines that manipulate a particular frame also provide further explanations. The porter should have a good understanding of what the frames are used for before setting frame layouts as they are pervasive throughout the assembly language portions of the system.

This document is rather tightly bound to the VAX implementation of Icon. Because of this, the stack model that is used is that of the VAX. Specifically, the VAX stack starts in high memory and grows downward. Thus, when something is pushed on the stack, the stack pointer goes down. When something is removed, the stack pointer goes up. The only time that this convention is departed from is in the use of the phrase "the top of the stack". The top of the stack is the stack word that has the *lowest* memory address.

The procedure frame layout is the first to be determined. The layout is somewhat fixed by the C compiler and target machine, so the task is a combination of making a decision and also recognizing what has been pre-determined. On most machines, the task of the porter is more one of recognition than of design.

The first thing to determine is the frame layout imposed by the target machine and the C compiler. Create a file containing the following

```
f()
{
   x(1,2);
}
```

Compile the file using *cc* in such a manner as to catch the assembly code that is generated in a file. The -S option of *cc* should cause assembly code to be placed in a file. On the VAX, the code generated by x(1,2) is

```
pushl   $2
pushl   $1
calls   $2,_x
```

From this it can be seen that arguments are pushed on the stack using the pushl instruction, and that the calls instruction does the actual procedure call. The first argument to calls is the number of arguments that are on the stack. When a return is made from a procedure called with a calls instruction, the arguments are removed from the stack by the return mechanism. On some machines, the removal of arguments after a subroutine call is left to the programmer (or code generator, in this case). This is usually done by adding a value to the stack pointer or incrementing the stack pointer several times.

Examine the assembly code produced on the target machine by the given C statements. Determine what actions are taken by the machine when the appropriate call instruction is performed. It is important to completely and totally understand what the target machine does when a call is performed. Next, determine what sort of procedure frame is used by C routines. Compile the following C function using -S.

```
f(a,b,c)
int a; char b; char *c;
{
   int x,y;

   x = a;
   a = 1;
   y = 2;
}
```

Look at the generated code and try to get a feel for what is going on. The things that need to be determined are:

how arguments are accessed
the format of the C call frame
register saving and restoring conventions

For example, on the VAX, the following code is generated for the test procedure.

```
        .word   L12             register save mask, filled in later
        jbr     L14             jump to end to make stack space
L15:
        movl    4(ap),-4(fp)    x = a
        movl    $1,4(ap)        a = 1
        movl    $2,-8(fp)       y = 2
        ret                     return
        .set    L12,0x0         set register mask
L14:
        subl2   $8,sp           make room for two local variables
                                   of four bytes each
        jbr     L15             jump to start of routine
```

Several inferences can be made. First of all, arguments are accessed relative to ap, the argument pointer.

Secondly, local variables are accessed relative to fp, the frame pointer. On the VAX, because of the hardware register save and restoration based on the entry mask (the first word of the routine), no subroutine calls are required to save registers.

The Icon procedure frame must have the following attributes:

(1) The values on the stack at the time of call to the procedure appear as arguments to the procedure. Furthermore, the values must be accessible in a deterministic fashion.

(2) Register values are saved in the frame and can be accessed deterministically.

(3) _line and _file appear in the procedure frame just below the last word pushed on the stack as part of the C procedure calling protocol.

(4) The region for local variables begins at the lower end of the "constant" portion of the frame. Local variables must be be accessible via deterministic means.

(5) The procedure frame created by a C procedure call must be a subset of the procedure frame selected. That is, the Icon procedure frame must be an augmentation of the C procedure frame.

The VAX uses this procedure frame layout:

		arguments
	4	number of arguments (nargs)
ap →	0	number of words in argument list (nwords)
		saved r11 (efp)
		saved r10 (gfp)
		...
		last saved register
	16	saved pc
	12	saved fp
	8	saved ap
	4	program status word and register mask
fp →	0	0 (condition handler address)
	-4	saved value of _line
	-8	saved value of _file
sp →		Icon local variables

Actually, on the VAX, most of the decisions are predetermined by the VAX architecture. The arguments are present on the stack, so they are the high end of the frame. The registers are saved on the stack by the calls instruction. The values of _line and _file naturally fit after the saved registers. The locals then appear on the lower end and extend for a variable distance (on a per-procedure basis). Note that the first local is at -16(fp) and the *last* argument is at 8(ap).

The VAX hardware takes care of saving and restoring registers upon subroutine entry and exit. It is quite possible that the target machine will not have this capability and the task must be delegated to software. This is usually evidenced by a call to a routine with a name such as csave as the very first thing in the routine and a call to a routine with a name such as crestore at the end of a routine. If this is the case, the actions of the saving and restoring routines must be taken into account when determining the procedure frame layout.

In addition to determining the procedure frame layout, a procedure frame pointer must also be selected. On the VAX, the fp stays constant throughout execution of a C procedure; it is used as the procedure frame pointer. For the target machine, there should be some register on which references to local variables (and perhaps parameters) are based. That register should be used as the procedure frame pointer (sometimes referred to as the pfp). The pfp need not point at the lowest word pushed on the stack as part of the procedure call; it only needs to be constant while a procedure is executing. Of course, the pfp changes while the program is executing; by "pointing at" a particular word, it is meant that the pfp always references a certain word in the procedure frame marker. An rt.h constant, FRAMELIMIT, is dependent on the number of words between the lowest word of the procedure marker and the word that the pfp points to. Setting FRAMELIMIT is described below.

A point about terminology should be stressed. The procedure frame marker is bounded by arguments on one end and the Icon local variables on the other. A procedure marker, the arguments, the Icon local

variables, and the stack below the local variables compose a procedure frame.

Determining the procedure frame layout is by no means a deterministic process. It takes work, but once it's successfully set, the single hardest task of the port is complete.

Once the procedure frame has been set, the generator frame layout follows rather easily. A generator frame is merely an augmented procedure frame. The generator frame has two additional pieces of information, a saved value of _k_level, and a saved value for the boundary. It is recommended that the generator frame be identical to a procedure frame except that the two extra words required be located between the lowest word that is pushed on the stack by the procedure call mechanism and the saved value of _line. Thus, on the VAX, the generator frame *marker* is

		saved r11
		saved r10
		...
		last saved register
	20	reactivation address
	16	saved fp
	12	saved ap
	8	program status word and register mask
	4	0 (condition handler address)
gfp →	0	saved value of the boundary
	-4	saved value of _k_level
	-8	saved value of _line
	-12	saved value of _file

Note that instead of a saved pc value, the generator frame marker holds a reactivation address. Control passes to this address when the generator is reactivated. Reactivation is fully explained in later sections.

A generator frame pointer (gfp) is associated with a generator frame. On the VAX, r10 is the gfp. The choice of a gfp is indirectly determined by the machine architecture and is intertwined with the selection of an expression frame pointer. The selection of the register to use as the gfp is discussed below. It is recommended that the gfp point at the word containing the saved boundary value.

The third type of frame marker is the expression frame marker. Expression frame markers are totally machine independent and contain three pieces of information: a saved expression marker address, a saved generator marker address, and a failure label that is to be given control in certain circumstances. On the VAX, the expression marker layout is

efp →	0	saved efp value
	-4	saved gfp value
	-8	failure address

This same format should be used on the target machine and there is no apparent reason for needing an alternative format. The expression frame pointer (efp) should point at the high word of the expression marker.

The registers that should be used for the gfp and efp are indirectly dependent on the procedure call mechanism. The primary requirement for the registers used as the efp and gfp is that they are saved across procedure calls. The secondary requirement is that the gfp and efp always be saved in a procedure frame. If the target machine has two general purpose registers that are always saved in a procedure frame, those two registers are quite suitable for the gfp and efp.

If the procedure call mechanism does not always save a pair of general purpose registers, the problem is more complicated. There are stack manipulations that are performed that *require* saved values of efp and gfp to be present in procedure and generator frames. For built-in procedures and Icon procedures this is no problem because invoke creates the procedure frame for them and can insure that the registers are saved. On the VAX, for the C routines that are directly called from interp, no such assurances can be made because the VAX C compiler directs only the registers used in a routine to be saved in the C procedure frame. This creates a problem because Icon counts on the registers being saved. The problem is countered by making the C compiler think that certain registers are used in certain routines. Specifically, declarations for a pair of register int variables are placed at the start of appropriate routines. On the VAX, the first two local variables declared

in a C routine *always* get allocated to r10 and r11. Thus, r10 and r11 are used for the gfp and the efp respectively. If the target machine is like the VAX in that it doesn't always save certain registers, a similar tactic may need to be used. If this is the case, try compiling a routine with a pair of register int variables declared and see what the compiler does. If the compiler saves the two registers assigned to the variables, use those registers for the gfp and the efp. It is wise to attempt to be sure that the compiler is deterministic in making its choice of registers to allocate to the variables. Routines that require this ruse to be employed have a line containing the string DclSave as the first line of the declarations. DclSave is defined in rt.h and should be set to an appropriate value. It may be the case that no registers need to be saved. If so, define DclSave, but specify no value. This is done for the PDP-11.

It is also necessary to select a register to use as the interpreter program counter (ipc). Any general register that is preserved across procedure calls is suitable. The VAX uses r9 for the ipc.

3.5 Machine and System Specific Values

Edit h/rt.h and search for the first PORT section. Define the various constant values as outlined below.

MAXHEAPSIZE

The size of the heap storage region in bytes. The VAX uses 50k and the PDP-11 uses 10k. If you have a small machine, use 10k. Larger machines should use larger values, such as that for VAX.

MAXSTRSPACE

The size of the string storage region in bytes. As with MAXHEAPSIZE, this value is somewhat arbitrary. A value similar to that used for the heap size should be used.

STACKSIZE

The size of co-expression stacks in words. Use 1000 for smaller machines, 2000 for larger ones.

MAXSTACKS

The number of co-expression stacks initially allocated. Use 2 for smaller machines, 4 for larger ones.

NUMBUF

The number of i/o buffers available. When a file is opened, a buffer is assigned to the file if one is available. A value from 5 to 10 is recommended.

INTSIZE

LOGINTSIZE

LONGS

MINSHORT

MAXSHORT

These constants must be set to the values they were given (if any) in link/ilink.h.

MINLONG

The smallest value that can be represented in a long.

MAXLONG

The largest value that can be represented in a long.

LGHUGE

The highest base-10 exponent plus 1 representable by a float. For example, on the VAX, the highest number representable by a float is about 1.7×10^{38}. Thus, LGHUGE is 39 on the VAX.

FRAMELIMIT

As discussed above, set FRAMELIMIT to the number of words between the low word of the procedure frame marker and the word that the procedure frame pointer references.

STKBASE

This value represents the approximate base of the stack when execution begins. On machines such as the VAX, where the stack grows down from high memory, STKBASE should have a high value, where on machines where the stack grows up from low memory, STKBASE should have a low value. The *man* page for *exec(2)* usually specifies the initial value for the stack pointer when program execution begins. If uncertain, be extreme with the value.

GRANSIZE

The granularity of memory allocations. Calls to *brk(2)* are used to expand the main memory that is being used. When *brk* is given an address to expand to, it rounds it to a multiple of a certain number. That number should be used for GRANSIZE. The *man* page for *brk(2)* should state what value is used on a particular system.

DclSave

Give DclSave the value needed as previously described.

EntryPoint(x)

EntryPoint is a macro that is used to yield the address of the first instruction of the C routine x that is past any procedure entry protocol instructions. On the VAX, the register mask is two bytes long and thus the first executable instruction of a routine x is at (char *)x + 2. On the PDP-11, there is a four-byte instruction at the start of each routine that calls the routine csv to save registers and establish the procedure frame. Thus for the PDP-11, EntryPoint(x) is (char *)x + 4. Values calculated by EntryPoint are used in invoke.

DummyFcn(name)

Initially, each of the assembly language portions of the system that must be filled in consist of a single line of the form DummyFcn(name). DummyFcn should be defined to generate *assembly* language statements that form a dummy routine with the label name. This can be as simple as a label and a global declaration. It is advisable to include as part of the definition something that will cause a program abort. A halt instruction usually does the job. Thus, the system can be built and will function normally unless an incomplete routine is called.

DummyDcl(x)

A macro that should expand into an assembly language declaration that allocates a word of storage for a variable named x.

DummyRef(x)

A macro that should expand into an assembly language reference to the variable x.

Global(x)

A macro that should expand into an assembly language declaration of x as a global symbol.

fp, efp, gfp, ipc

It is advisable to use #defines for these registers rather than explicitly name them in the code.

cset_display

This is a rather complicated macro that is used to initialize the values of csets such as &cset and &lcase. If the target machine has ints with 32 or 16 bits, then the VAX or PDP-11 definition (respectively) of cset_display may be used. If this is not the case, cset_display will have to be hand-crafted and the various uses of it will have to be altered for the machine in question. Briefly, a cset_display specifies which of the 256 bits that comprise a cset are to be set to 1. For example, the cset_display for &cset has all the bits set to 1, while &ascii has the first 128 bits set to 1. csets are accessed using the setb and tstb macros which are also defined in rt.h. cset_displays appear in iconx/init.c, functions/bal.c, and functions/trim.c. It may also be necessary to modify the definitions of CSETSIZE, setb, and tstb.

Search for the second PORT section. F_NQUAL, F_VAR, F_TVAR, F_PTR, F_NUM, F_INT, and F_AGGR should be given the same values they have in link/datatype.h.

Once rt.h has been completed, an analogous file, h/defs.s must tailored. defs.s is a subset of rt.h that is included in assembly language files. The PORT section of defs.s lists a number of constants that must be defined. Use the appropriate values from rt.h for each constant. If all assemblers used a default radix of 10 for constants, it would be possible to tailor defs.s mechanically, but since this is not the case, defs.s must be modified by hand.

3.6 Complete System Compilation

In order to determine if there are serious C compiler problems with the interpreter source, the entire system should be made at this point. Do a

```
make Icon
```

in the root directory of the Icon distribution. The entire system should compile without any problems. The resulting interpreter will be completely disfunctional, but if it is built without any problems, it provides further evidence that the C compiler is up to the task.

3.7 Porting the Assembly Language Routines

The porting of the assembly language routines is the most difficult part of porting Icon. This document has a section for each assembly language routine and each routine is described in three ways:

overview
generic operation
the routine on the VAX

The overview section briefly describes the action of the routine and how the routine may be encountered during the course of execution. The generic operation section tells what steps the routine takes to perform its given task. Each major step that the routine takes is described. These steps should be very similar from machine to machine. The section about the routine on the VAX details the operation of the routine on the VAX. This section complements the comments contained in the source code for the routine and should be read with the source code at hand. This section is very machine specific. (Ideally there would be one such section for each existing Icon implementation.)

Each routine must be formulated for the target machine. For the most part, the best approach is to take the same steps that are taken on the VAX. It is important to select the right level for modeling the VAX routines. Try to recognize the steps that are made rather than following the operations on a per-instruction basis. The most important thing is to have a good understanding of what actions are performed and how these can be done on the target machine.

The first goal is to get a very simple Icon program working. This first program is **test/hello.icn**. It is quite short:

```
procedure main()
   write("hello world")
end
```

The basis of routines mentioned above (**start.s**, **setbound.s**, **invoke.s**, **interp.s**, **efail.s**, and **pfail.s**) must be implemented for even a very simple Icon program to work. However, all these routines do not need to be written to make **hello** *begin* to work.

Translate and link **hello** by running the translator and the linker:

```
tran/itran hello.icn
link/ilink hello.u1
```

This creates an interpretable file named **hello**. Just to get the feel of things, run the interpreter on the file:

```
iconx/iconx hello
```

A message of some type and a core dump should be produced.

The files **tran/itran**, **link/ilink**, and **iconx/iconx**, are copied into the **bin** directory as the last action of

```
make Icon
```

in the root directory. The porter may find it convenient to link these files to the **bin** directory and then place the full pathname in his search path. It is necessary to remove **itran**, **ilink**, and **iconx** in the **bin** directory before linking them. Also, if the files are linked, the last step of *make* in the root directory will fail. This failure is inconsequential.

As **start.s** et al. are written, try stepping through them to be sure the correct actions are being performed. Most of the assembly language source files are straight line code with a branch or two and it is possible to do a large amount of verification of the assembly code by single stepping through it with a debugger.

When a routine has been completed, it may be added to the interpreter by doing a *make* in the directory containing the routine and then doing another *make* in the directory iconx.

3.7.1 iconx/start.s

Overview

The routine mstart in start.s is used to get Icon started. When the Icon interpreter is executed, the C routine main passes control to mstart, and merely serves as a front-end for mstart.

Generic Operation

(1) Call the routine init with the name of the file to interpret as its argument.

(2) Make an Icon list out of the command line arguments using the llist function.

(3) Invoke the main procedure of the Icon program.

start on the VAX

There is a short main program in iconx/main.c that calls mstart with two arguments:

```
main(argc, argv)
int argc;
char **argv;
{
mstart(argc, argv);
}
```

The number of command line arguments is in argc, and argv is a pointer to an array of pointers to strings representing the arguments. argv[0] is the command used to invoke the interpreter and argv[1] is the name of the file being interpreted. Additional command line arguments are passed along to the main procedure of the Icon program. When mstart gets control, 4(ap) is the argc value and 8(ap) is the argv value.

The first action taken by mstart is to call init to initialize the Icon run-time system. init loads the header and code portions of the interpretable file into memory, so init needs the name of the interpretable file. The word at 8(ap) is loaded into r9, pointing it at argv[0]. Then the name of the file to interpret (argv[1]), residing at 4(r9), is pushed on the stack as the argument for init, which is then called.

In order to provide conformity with the usual execution environment of Icon procedures, an expression frame is created for the execution of the main procedure. Both the old expression frame pointer and the old generator frame pointer are set to be 0 in the expression frame for main. The failure label must point to an interpreter opcode that will terminate execution of the program. The opcode 0 is used for this purpose. A word of storage, flab, is declared and initialized to 0. The failure label points to flab. Thus, if main fails, the interpreter executes the quit opcode.

The next task is to push the descriptor for the procedure main on the stack for later use by invoke. The variable _globals contains the address of the list of global variable descriptors. The first global variable descriptor is always the one for the procedure main; if no main procedure was found when the program was linked, the descriptor will be &null. The value of _globals is loaded into r0 and the word then referenced by r0 is checked to see if it is equal to D_PROC. (The first word of a descriptor for a procedure is always equal to D_PROC.) If the word is not equal to D_PROC, a branch is made to nomain which generates the appropriate run-time error. Otherwise, the descriptor for main is pushed onto the stack. (The effect of the instruction movq (r0),-(sp) is to move 8 bytes (the size of a quadword) starting at the address referenced by r0 to the 8 bytes referenced by the sp after subtracting 8 from the sp.)

The main procedure is to be invoked with a list consisting of the command line arguments (if any). The Icon run-time routine llist is used to make the list that is passed to the main procedure. llist stores the descriptor for the list that it creates in the descriptor above its first argument descriptor, so to accommodate the result, a null descriptor is pushed on the stack using the clrq instruction. Note that because llist calls setbound and clrbound, it is not possible to execute all of start until rt/setbound.s has been completed.

At the beginning of this routine, r9 was set to point at the first word of the argument list. Neither the name of the Icon interpreter nor the name of the interpretable file is desired in the argument list passed to main, so r9 is twice incremented by 4 (the size in bytes of a word) to point it at the first actual program argument.

The next step is to construct the argument list for llist. For each command line argument, the address of the string and then its length are pushed on the stack. The length and address pairs form descriptors that llist makes an Icon list from. r8 is used to count the arguments. After the addresses and lengths of each argument have been pushed, the number of arguments is pushed on the stack. At this point, the stack looks like this:

```
                descriptor for main procedure (2 4-byte words)
                a null descriptor (2 words containing 0)
                address of first argument to Icon program
                length of first argument
                ...
                address of last argument to Icon program
                length of last argument
sp ⟶            number of arguments
```

All addresses and lengths are one word in size. The calls instruction needs to be told how many words are in its argument list. (This is necessary because when a return is made from the subroutine, the specified number of words are removed from the stack.) There are two words for each argument and an additional word for the number of arguments. (Do not confuse the argument count for the llist subroutine and the argument list size.) r8*2+1 is calculated in r8. This value is used as the argument list length for the calls instruction. When llist returns, the arguments are stripped from the stack and the stack looks like this:

```
                descriptor for main procedure
sp ⟶            descriptor for list of command line arguments
```

Note that the null descriptor pushed earlier received the result of the llist function.

At this point, the main procedure is ready to be invoked. The descriptor for the main procedure is arg_0 and the descriptor for the list of command line arguments is arg_1. Before invoking the main procedure, the procedure frame pointer and the generator frame pointer are cleared. The main procedure is being invoked with one argument, so a constant 1 is pushed on the stack. The calls instruction is given an argument of 3 because a word is used for the number of arguments and two additional words are used for the descriptor for the list of command line arguments.

If the main procedure fails, the interpreter will encounter the 0 opcode discussed earlier. If the main procedure returns (this usually isn't done), the return will manifest itself as **invoke** returning. If this happens, _c_exit is called with an argument of 0. Incidentally, this also happens when the interpreter hits the 0 opcode.

There is a block of code labeled nomain that is executed when no main procedure is found. This calls the routine runerr to produce an error message. The actual call made is runerr(117,0). The number 117 is looked up in a table of run-time errors. If the second argument to runerr is non-zero, it is interpreted as being the address of a descriptor and the descriptor is examined to produce an "offending value" to accompany the run-time error.

The last portion of executable code in **start.s** is the subroutine _c_exit. If the variable _monres is non-zero, it indicates that profiling is on, and it must be turned off. This is accomplished by calling _monitor(0). The routine __cleanup is then called to shut down the i/o system. Finally, _exit is called with the argument of _c_exit to terminate execution of the Icon interpreter.

There are several data declarations in start.s. The first data declaration is a .space 60. This is an accommodation for the garbage collector. It insures that enough of the start of the data section is used up to force the addresses of other data objects to be greater than the defined constant MAXTYPE in **h/rt.h**.

Some assorted declarations are next. flab is referenced by the interpreter if the main procedure fails. It must be at least a byte long and contain a 0. _boundary must be a word long and contain a 0. _environ must be a word long; its contents are unimportant as it is written into at the beginning of start.s.

The _tended array is also used in conjunction with garbage collection. It must declare space for five descriptors (two words per descriptor) that are initialized to 0. The label _etended is used to mark the end of

the _tended array.

3.7.2 rt/setbound.s

Overview

setbound.s contains code for setbound and clrbound. setbound sets _boundary under appropriate conditions and clrbound clears _boundary under appropriate conditions.

Generic Operation

When a call is made from Icon into C, _boundary must be set to point at the procedure frame on the top of the stack. C routines that can be called from Icon have a call to setbound as their first operation. If setbound is called and _boundary is not set, that is, _boundary has a value of zero, boundary is set to value of the fp of the calling procedure.

When a C routine returns to Icon, _boundary must be cleared. If _boundary has the same value as fp, the routine was called from Icon and thus when it returns, it returns to Icon. At appropriate return points in routines, a call to clrbound is made. If the return takes control back to Icon, _boundary is cleared.

setbound on the VAX

The value of _boundary is tested. If boundary is not set, that is, if it has a value of zero, it is set to the value of the fp in the calling procedure. Because the call to setbound creates a procedure frame, the fp value saved in the frame is used. If _boundary is already set, it is not changed.

clrbound on the VAX

The value of _boundary is compared to the fp in the calling procedure. If the values are the same, the calling procedure was called from Icon and when it returns it returns to Icon. If this is the case, _boundary is cleared.

An Alternative Approach

If the C system on the target machine uses calls at the beginning and end of C routines to save and restore registers, it is possible to modify these routines to set and clear the boundary. This approach is used on the PDP-11.

The file rt/csv.s contains replacement routines for csv and cret. When csv is called to save registers, if _boundary is 0, sp is saved in _boundary. This insures that the first call from Icon into C leaves _boundary at the top of the stack at that point. When cret is called to restore registers, if the procedure frame pointer is equal to _boundary, the return is taking control back to Icon code and _boundary is cleared.

Note the resemblance between calling setbound at the start of a C routine and having csv set _boundary whenever a C routine is entered. Similarly, there is a resemblance between calling clrbound at appropriate points and having cret clear _boundary when necessary.

The initial implementation of Icon was on the PDP-11 and the modified csv and cret approach was used. When the system was ported to the VAX, the subsumption of csv and cret by the hardware required that a software approach be taken. The trade-offs are marginal if the target machine uses save and restore routines. Having a distinct setbound and clrbound may be easier to implement, but using modified save and restore routines is definitely faster.

If this approach is used, then SetBound and ClearBound in rt.h should be defined, but given no value.

3.7.3 lib/invoke.s

Overview

invoke.s handles four specific tasks. These are

call a built-in function
call an Icon procedure
create a record
perform mutual evaluation

Note that all of these operations rise from a source code expression of the form

$arg_0(arg_1,\ldots,arg_n)$

where each arg_i is an expression of some type. Icon has strict left-to-right evaluation and thus, for the preceding expression, arg_0 is evaluated first, then arg_1, and so forth through arg_n. As each expression is evaluated, its result is pushed on the stack. After the expressions have been evaluated (and assuming none failed), the stack looks something like

```
                value from arg0
                value from arg1
                ...
                value from argi
                ...
sp ⟶            value from argn
```

Then, arg_0 is *invoked.*

Recall that stacks are represented as growing downward. Thus, arg_n is on the "top" of the stack. Also note that each "value" on the stack is actually a two-word descriptor.

Generic Operation

invoke is an interpreter opcode that takes a single operand specifying how many arg_i are present (arg_0 is not counted). invoke is also called from start.s to invoke the main procedure.

(1) invoke must determine whether it is to call a built-in function, call an Icon procedure, create a record, or perform mutual evaluation. If arg_0 is not something that can be invoked, invoke notes this as a run-time error.

(2) If mutual evaluation is to be done, invoke selects the value of arg_i that corresponds to the value of arg_0. That is, if arg_0 is 2, then arg_2 is selected and returned. If arg_0 references a value that is out of range, invoke fails.

(3) If an Icon procedure or built-in procedure is being called, the argument list is adjusted to conform to the number of arguments that the procedure or function is expecting. This may mean supplying &null for missing values, or discarding the last few arg_i. Some built-in functions take a variable number of arguments, but all Icon procedures take a fixed number of arguments. If the desired operation is creation of a record, invoke treats this just like invocation of a built-in procedure.

Built-in functions and Icon procedures are handled in very different ways. If a built-in function is being invoked, it is simply called. (The calling sequence is somewhat convoluted on the VAX and is described later.) Calling an Icon procedure is more involved; the following actions are taken.

(1) Each arg_i in the (adjusted) argument list is dereferenced.

(2) If &trace has a non-zero value, the function ctrace is called with appropriate arguments. ctrace produces output that includes the name of the procedure being called and the arguments that are being passed to it.

(3) The remainder of the procedure frame (partially constructed by the call itself) is built. This includes pushing values for _file and _line on the stack. _file is a pointer to a string that names the source file from which the code currently being executed came. _line is the number of the source line that is currently being executed. A descriptor for &null is pushed on the stack for each dynamic local of the procedure.

(4) The generator frame pointer is cleared (because a new expression context is being entered). ipc is loaded with the entry point of the procedure being called. Control is then passed back to the interpreter using a jump instruction.

invoke on the VAX

On the VAX, immediately after invoke has been entered, the stack is

		value from arg_0
		value from arg_1
		...
	8	value from arg_n
	4	number of expressions - 1 (nargs)
ap →	0	number of words in argument list (nwords)
	-4	saved r11
	-8	saved r10
		...
		saved r1
		saved pc
	12	saved fp
	8	saved ap
	4	program status word and register mask
sp, fp →	0	0 (condition handler address)

The first action of invoke is to set _boundary to the value of fp. This is done because invoke may be invoking a built-in procedure.

The number of arguments with which arg_0 is to be invoked is contained in the *nargs* word, which resides at 4(ap). This value is frequently used and is put into r8.

invoke makes frequent use of arg_0, but its address is not a fixed distance from any known point. Rather, the address of arg_0 must be calculated using the address of arg_n and the number of arguments. The VAX movaq instruction makes this calculation easy. The desired calculation is

```
r11 = 8 + ap + (r8 * 8)
```

and is performed by

```
movaq   8(ap)[r8],r11
```

arg_0 may be a variable and if so, it needs to be dereferenced. r11, which contains the address of arg_0 is pushed on the stack and deref is called. The dereferencing is done "in place"; the previous value of arg_0 is replaced with the dereferenced value. The dereferenced value is a descriptor whose first word contains type information and whose second word (in some cases) contains the address of a data block which holds the actual value of the object. Note that r11 points to the first word of this descriptor.

Recall that the first task of invoke is to determine what arg_0 is and to act accordingly. The simplest case is when arg_0 is a procedure. That is checked for by comparing 0(r11) with D_PROC. If arg_0 is a procedure, a forward jump is made to doinvk.

It is more interesting if arg_0 is not a procedure. The first alternative investigated is mutual evaluation. mutual evaluation is similar to a procedure call, but rather than arg_0 being a procedure, it is an integer that selects one of the arg_i. The selected arg_i is the outcome of the mutual evaluation. The routine cvint is used to try to convert arg_0 to an integer. If arg_0 cannot be converted to an integer, a forward branch is taken to trystr to explore another possibility. For mutual evaluation, a non-positive value of arg_0 is acceptable and is converted to a positive value using the cvpos routine. (Expressions in the argument list are indexed the same way that characters in a string are indexed.) If the position is greater than the number of expressions in the list, that is, if the reference is out of range, the mutual evaluation fails by calling the routine fail. If the position is in range, the selected arg_i must be returned as the result of the invocation (and the result of the mutual evaluation). The arg_i to return is selected by multiplying the position by 8 (each arg_i is a descriptor) and subtracting

that from r11, which points at arg_0. The descriptor thus referenced is copied into the location of arg_0. (Recall that arg_0 is used to receive the result of an operation.) _boundary is then cleared and invoke returns.

If arg_0 is not convertible to an integer, conversion to a procedure is attempted. (Note that this is an experimental extension to Icon.) arg_0 is first converted to a string using cvstr. If the conversion is successful, the routine strprc is called to see if the string "names" a procedure. The conversion performed by strprc is "in place", i.e., arg_0 becomes a descriptor for a procedure. If either the conversion in cvstr or strprc fails, arg_0 is deemed to be uninvocable and this is noted by run-time Error 106.

At this point (the label doinvk), arg_0 is a descriptor to a procedure to be invoked and r11 points to arg_0.

The next operation is to make the number of arguments supplied conform to the number of arguments that the procedure is expecting. The number of arguments that a procedure expects is in the fifth word of its procedure block. This value for the procedure being invoked is obtained and placed in r10. If the value is negative, the number of arguments that the procedure expects is variable. Only built-in procedures can have a variable number of arguments, so if the desired argument count is negative, control passes to the label builtin.

r8 contains the number of arguments given and r10 contains the number of arguments desired. The value of r10 is subtracted from r8, leaving r8 with the difference. If the number of arguments supplied is the same as the number expected, no adjustment is needed and a forward jump is made to doderef. Otherwise, the stack must be adjusted.

First, *nwords* and *nargs* on the stack are adjusted. Recall that *nargs* is used by Icon and is the number of arguments for a procedure. *nwords* is used by the VAX hardware and is the number of words that the argument list for a subroutine occupies. *nargs* resides at 4(ap) and is updated by storing r10 in 4(ap). *nwords* is trickier because only the low-order byte of the *nwords* word is to be used for the word count. The low-order byte of r10 is stored in 0(ap), doubled, and then incremented by one. (The increment by one is to allow for the *nargs* word that is part of the argument list.)

The deletion of excess arguments or addition of &null for missing arguments is accomplished by moving the lower portion of the stack up or down to overwrite excess arguments or to leave space for missing arguments. Consider the following: A procedure that expects one argument has been invoked with three arguments. The stack is

	128	arg_0
	120	arg_1
	112	arg_2
	104	arg_3
	100	nargs (3 at call, 1 after adjustment)
ap →	96	nwords (7 at call, 3 after adjustment)
	92	r11
		...
	48	r1
	44	pc
	40	fp
	36	ap
	32	psw
sp, fp →	28	0

The situation desired is

	128	arg_0
	120	arg_1
	116	nargs (1)
ap →	112	nwords (3)
	108	r11
		...
	64	r1
	60	pc
	56	fp
	52	ap
	48	psw
sp, fp →	44	0

Note the address field (which has been arbitrarily assigned). Observe that arg_0 and arg_1 are in the same place, but that the lower portion of the stack has moved up. Consider what would be desired if the procedure being invoked requires five arguments and only three are supplied. The desired stack configuration is

	128	arg_0
	120	arg_1
	112	arg_2
	104	arg_3
	96	&null (arg_4)
	88	&null (arg_5)
	84	nargs (5)
ap →	80	nwords (11)
	76	r11
		...
	32	r1
	28	pc
	24	fp
	20	ap
	16	psw
sp, fp →	12	0

As before, the "good" arguments are in the same place, but the stack has moved down to make room for arg_4 and arg_5 and a value of &null is supplied for them.

The VAX hardware makes these stack manipulations easy. Recall that r8 contains

number of arguments expected - number of arguments supplied

Thus, in the first case r8 has a value of 2, while in the second case r8 is -2. The value of r8 is multiplied by 8 to turn the argument count into a byte count. This byte count is added to sp, effectively moving it to where it should be. Now, a block of memory starting at where fp points must be moved to where sp points. The size of the block needs to be considered. It starts at the fp and contains the condition handler, the old psw, ap, fp and pc — five words. Eleven saved registers are included; eleven more words. (A constant, INVREGS, is used to represent the number of saved registers.) Finally, two words for *nargs* and *nwords* — a total of 18 words. The VAX instruction that makes the move is

```
movc3   $(INVREGS+7)*4,(fp),(sp)
```

which moves 18*4 (=72) bytes from where fp is pointing to where sp pointing. The VAX allows the source and destination fields to overlap without producing "curious" results.

Some housekeeping must be performed after the move is made. fp is set to point to the same word sp points to, and the boundary is set to point to the same place. ap is adjusted to point to the *nwords* word.

If arguments were deleted, the adjustment process is done. If arguments were added, &null must be supplied as the value for each argument that was added. Because the descriptor for &null consists of two words containing 0, the task amounts to filling in null bytes in the "hole" that was made. r8 contains the number of

bytes in the "hole". r8 is negative and it is negated to obtain a positive byte count. The instruction

```
movc5   $0,(r0),$0,r8,(INVREGS+7)*4(sp)
```

does all the work. Specifically, this instruction moves bytes of zeroes starting at (r0) to a field that starts at 72(sp) and extends for r8 bytes. The third operand of the instruction is a "fill character" that is used in the event that the destination field is longer than the source field. In this case, the source field is 0 bytes long, and a null-byte fill character is moved into each byte of the destination field. This is the usual way to zero out a block of memory on the VAX.

At this point, the correct number of arguments for the procedure are on the stack.

For an Icon procedure, all arguments must be dereferenced before the procedure is called. Procedure blocks have a field that tells how many dynamic local variables the procedure has. For built-in procedures this field has a value of -1. If a built-in procedure is to be invoked, control jumps to the label builtin. If an Icon procedure is to be invoked, but it has no arguments (and thus they do not need to be dereferenced), a forward jump is made to cktrace.

r11 points to the descriptor for arg_0. The address of arg_1 is used later in invoke, and its address is calculated using r11 because it's handy. r10 contains the number of arguments and this value is stored in r5 for subsequent use of r5 as a counter. The arguments must be dereferenced, this is done by calling deref with the address of each argument. The instruction

```
pushaq  -(r11)
```

is used to decrement r11 by 8 and then push the value of r11 on the stack. Because r11 initially references arg_0, the first time through r11 is decremented to point at arg_1 and deref is called with the address of arg_1. r11 is backed down through the expression list, with each arg_i being dereferenced in turn. Note that the sobgeq instruction is used as a loop controller, decrementing r5 and jumping back to nxtarg as long as r5 is not 0.

At this point, an Icon procedure is being invoked; it has the correct number of arguments and they have been dereferenced.

If tracing is on, (indicated by a non-zero value for _k_trace), a trace message must be produced at this point. The routine ctrace does all the work. It needs to be called with the appropriate arguments. ctrace requires three arguments: procedure block address, number of arguments, and the address of the first argument. These are pushed on the stack and ctrace is called.

The portion of the stack from arg_0 on down constitutes a partial procedure frame and it must be completed. _line and _file are pushed on the stack. To complete the frame, the local variables must be pushed on the stack. Local variables have an initial value of &null, and the movc5 idiom used previously is used again to zero out the space required for the local variables.

Because an Icon procedure is being invoked, the boundary is cleared and _k_level (the &level keyword) is incremented. The entry point for a procedure is stored in the third word of the procedure block. This value is loaded into ipc. A new expression context is being entered, and both gfp and efp are cleared using a clrq instruction.

Control is passed back to the main loop of the interpreter by jumping to interp. The Icon procedure is now being executed. The procedure eventually terminates by use of pret or pfail.

The section of code following builtin: handles the case where a built-in procedure is to be invoked.

If nothing is changed, when a built-in function returns, it would return to the instruction after the call to invoke. This is unsatisfactory, since the boundary needs to be cleared because of the transition from the C environment to the Icon environment. Rather than having a call to clrbound at the end of each built-in procedure, the boundary is cleared at a common return point.

The pc value that is stored at 16(fp) is "hidden" at 20(fp) where the value of r1 should be saved. 16(fp) is replaced with the address of the forward label bprtn. Thus, when the built-in procedure returns, it goes right to bprtn. Because a C environment is being entered, the boundary is set to the current value of fp. The third word of the procedure block contains the entry point of the built-in procedure and it is jumped to. It is important to understand that the procedure is not called because the call frame has already been constructed. Also, the entry point address stored in the procedure block *must* be past any instructions that are used to establish

the stack environment for the routine because this environment should already be on the stack.

When the built-in procedure returns, it comes to bprtn. The boundary is cleared because execution is back in an Icon environment. The procedure return restores r1, which contains the pc value that invoke should return to. Because the return has already swept off the old frame, 0(r1) is jumped to and invoke is finished. Note that the built-in procedure has left its result in arg_0, which is left on the stack. Thus, the direct result of invoke is an additional value on the stack.

Some Comments on invoke

It is important to understand the purpose of invoke. Unless mutual evaluation is being performed, the task of invoke is to create a procedure frame for an Icon or built-in procedure and then transfer control to the procedure.

On the VAX and the PDP-11, the call to invoke partially creates the procedure frame. invoke then completes the frame. For Icon procedures, control eventually passes out of invoke and goes back to the interpreter loop. However, for built-in procedures, after the frame is built, invoke *branches into* the C code for the procedure. Thus, it appears that the built-in procedure was called directly. This scheme is facilitated by the fact that entry points of C routines on the VAX and PDP-11 are a constant distance from the start of the routine. (The EntryPoint macro in rt.h specifies the distance.) On some machines this may not be practical and other schemes may need to be developed.

Another point that needs to be addressed is that of argument removal. When a built-in procedure, Icon procedure, or built-in operation is performed, the net result almost always is simply the addition of a descriptor to the stack. More precisely, the new descriptor is actually a replacement for arg_0, which is the descriptor for the procedure being called, or in the case of a built-in operation is &null. Recall that arguments are below arg_0 on the stack. The arg_0 word must be on top of the stack when a procedure or operation is complete. The VAX does this via hardware, which manages the stack and removes arguments when a procedure call is complete. The PDP-11 does not have such hardware facilities and thus the arguments must be removed manually. The problem is compounded by the fact that for built-in operations, invoke is never called; rather, the interpreter loop calls the appropriate subroutine directly.

The method employed on the PDP-11 uses the cret routine to remove arguments. cret is called at the end of each C routine to restore registers. Icon has a replacement for cret that restores registers but also removes arguments when appropriate. rt/csv.s contains the replacement routine. When cret is called, if r5 (the pfp on the PDP-11) is equal to _boundary, then cret is returning to Icon code and any arguments on the stack are removed, leaving arg_0 on the top of the stack. A similar technique may be needed on the target machine.

rt/csv.s also contains a replacement for the csv routine which saves registers upon entry to C functions on the PDP-11. Both csv and cret also contain code that is used to set and clear the boundary at appropriate times. See the description of rt/setbound.s for more details.

3.7.4 iconx/interp.s

Overview

interp.s is the main loop for the interpreter. The execution of an Icon program is stack based. As the interpreter executes an Icon program, it fetches instructions and accompanying operands out of the instruction stream of the interpretable file. Operands for interpreter instructions are pushed on the stack and results accumulate on the stack as operands for other instructions. In addition to simple incremental and decremental stack changes, the expression evaluation mechanism may cause portions of the stack to be duplicated and may also cause the top portion of the stack to be removed.

Generic Operation

An Icon program is executed by interpreting the interpretable file produced by the linker. The interpretation process itself is fairly simple. ipc points at the next instruction to be executed. (Recall that the interpretable file is loaded into memory.) The opcode of the instruction is fetched and the corresponding word in the jump table is taken as the address of a sequence of instructions that perform the desired operations. A branch is taken to the referenced location and the operation is performed. The operation may require operands; if so,

they appear in the instruction stream following the opcode. The segment of code that performs a particular operation is responsible for fetching the appropriate operands out of the stream. When the operation is complete, a jump is taken to the top of the interpreter loop and the process continues.

Interpreter operations are of two types. Operations of the first type call a routine to perform a task. Operations of the second type are executed entirely by the interpreter; no subroutine call is necessary.

Operations that require a call to be made call routines in the operators or lib directories. The routine being called may require one or more arguments. If arguments are required, they appear on the stack. When the routine returns, it removes any arguments that it was called with from the stack and leaves its result on the top of the stack.

To facilitate the calling of routines, a table known as optab parallels the jump table. An opcode *n* references the *n*th word of the jump table. If the operation designated by the opcode requires a call, the *n*th word of optab contains the address of the routine that should be called.

The interpreter saves space in its instruction stream by encoding operand information in some opcodes. For example, the line instruction has one operand that is used to set the value of _line, the current source line number. The linex instruction is an alternate form of line which encodes the line number as the low order bits of the opcode. Specifically, the opcodes from 192 to 256 are linex opcodes. For example, opcode 195 is equivalent to a line opcode with an operand of 3.

Implementing the Interpreter Loop

interp.s stands alone among the assembly language files as one that is well suited to coding in a macro fashion. Most of the interpreter loop is written in terms of *cpp* macros and thus porting it is largely a matter of writing the macros for the target machine.

The following #defines must be made.

Op

The operand register. Any general purpose register will do. The value of the register need not preserved between instructions; its lifetime is only from the time that an operand is fetched until the next opcode is fetched or a routine is called.

GetOp

This must expand into code that fetches the next operand out of the instruction stream and places it in the register Op. Recall that operand size is determined by the #define for OPNDSIZE in the linker. On the VAX, GetOp is merely

```
movl    (ipc)+,Op
```

This is because operands are one word long and can begin on any byte boundary. If the VAX did not support word fetching from arbitrary boundaries, it would be necessary to get the bytes from the instruction stream one at a time and make a word out of them using boolean operations. If such were the case, a reasonable alternative would be to make opcodes one word in size and thus all instruction stream objects (opcodes, operands, and words), would be of the same size and lie on word boundaries.

PushOp

Push the Op register on the stack. The VAX uses

```
pushl   Op
```

PushNull

Push a descriptor for &null on the stack. That is, push two words of zeroes. The VAX clrq instruction does the trick:

```
clrq    -(sp)
```

Push_R(x), Push_S(x), Push_K(x)

Push the value of x on the stack. To accommodate machines with non-orthogonal instruction sets, Push_R is used to push a register value, and Push_S is used to push the contents of a storage location. Push_K is used to push a constant value. The VAX uses

```
pushl   x
```

for both Push_R(x) and Push_S(x), while

```
pushl   $x
```

is used for Push_K(x).

PushOpSum_R(x), PushOpSum_S(x)

PushOpSum_R(x) adds the value of the register x to Op and pushes the result on the stack. PushOpSum_S(x) is similar, adding the value in the memory location x to Op and pushing the result. On the VAX,

```
addl3   Op,x,-(sp)
```

is used for both.

NextInst

Branch to the top of the interpreter loop. The VAX uses

```
jmp     _interp
```

CallN(n)

Call the routine corresponding to the current opcode with n arguments. On the VAX, the opcode fetching segment loads r0 with a byte offset into the jump table. This same byte offset references the location in optab which contains the address of the routine which corresponds to the current opcode. CallN(n) expands to

```
pushl   $n
calls   $((n*2)+1),*optab(r0)
```

pushl $n pushes the number of arguments on the stack. This word becomes the *nargs* word of the procedure frame. The first of operand of the calls instruction is the length of words in the argument list, since each argument is a two word descriptor and the *nargs* word occupies another word, n*2+1 is used as the length of the argument list. The address contained in the optab word referenced by r0 is the routine to call.

CallNameN(n,f)

Call the routine named f with n arguments. This is very similar to CallN, the only difference being that the routine to call is explicitly named rather than being implicitly determined from the opcode value. The VAX uses

```
pushl   $n
calls   $((n*2)+1),f
```

BitClear(m)

The constant value m designates bits in the Op register to leave on. All other bits in Op should be turned off. That is, the complement of m is ANDed with the contents of Op and the result is placed in Op. This is used to decode opcodes with encoded operands. The VAX uses

```
bicl2   $0!m,Op
```

Jump(lab)

Branch to the label lab. The destination label is close to the jump, so a short jump of some type may be used. The VAX uses

```
jbr     lab
```

LongJump(lab)

LongJump is like Jump with the exception that lab may be quite distant. The VAX uses

```
jmp     lab
```

Label(lab)

Generate a label declaration for lab. The VAX uses

```
lab:
```

VAX Specific Sections of interp

Several sections of interp are machine specific and must be coded on a per-machine basis. The sections in question are explained on an individual basis:

_interp

The next opcode is fetched and loaded into r0. movzbl moves a byte and zero extends it to a word value. Because a byte was fetched, ipc is incremented by 1. The opcode is saved in Op in case it contains an encoded operand. r0 is multiplied by 4 to turn it into a byte offset. A jump is made to the address indexed by r0 in jumptab to perform the desired operation. Eventually, a jump returns control to the label _interp to fetch and execute the next instruction.

op_bscan

A descriptor for _k_subject is pushed on the stack. Then the value of _k_pos is pushed, followed by the constant D_INTEGER. The routine corresponding to op_bscan, _bscan, is called with 0 arguments. (This causes the descriptors for _k_subject and the value of _k_pos to be left on the stack.) When _bscan returns, a branch is made to _interp.

op_ccase

A null descriptor is pushed on the stack. The word immediately above the current expression frame is then pushed on the stack.

op_chfail

The operand of chfail is fetched into Op. Op and ipc are added together and the result replaces the failure address in the current expression frame.

op_dup

A null descriptor is pushed on the stack. The value that was on top of the stack is now at 8(sp), and it is copied to the top of the stack using a movq.

op_eret

eret gets the value on top of the stack, removes the current expression frame and puts the previous top of stack value back on the top of the stack. First of all,

```
movq    (sp)+,r0
```

moves the descriptor on the top of the stack into the r0-r1 register pair and increments the stack pointer by 8. The gfp is loaded with the gfp value stored in the expression frame marker. sp is loaded from efp, bringing the expression marker to the top of the stack. The old efp value from the marker is loaded into efp. Finally, the value stored in the r0-r1 pair is pushed on the stack.

op_file

The operand of file is loaded into Op. Op and the value of _ident are added and the result in placed in _file.

op_goto

The operand is loaded into Op and then added to ipc.

op_incres

The eighth word of the co-expression heap block for the current expression is incremented by one.

op_init

This one is tricky. The init instruction arises from the initial statement in Icon and is used to effect one-time execution of a segment of code. The operand of init is the address of the first instruction after the segment that is to be executed once. The instruction

```
movb    $59,-(ipc)
```

decrements ipc by 1 and then stores the constant 59 in the byte that ipc references, which is the init

opcode. The magic number 59 is the opcode for goto, so in effect, the init had been made into a goto that skips a section of code. By adding 5 to ipc, it leaves ipc pointing at the first instruction of the initial code. The constant 5 is derived from the width of the opcode and associated operand, i.e., OPSIZE+OPNDSIZE.

op_invoke

This section has two entry points: op_invoke gets control if invoke has an operand, and op_invkx gets control if the operand is encoded in the opcode. If an operand is specified, it is fetched into Op. If the operand is encoded, BitClear(7) is used to isolate the operand in Op. Control converges at doinvoke. The operand is the number of arguments to invoke the procedure with and it is pushed on the stack as the *nargs* word. (The arguments are already on the stack.) The calls instruction needs the length of the argument list, so Op is multiplied by 2 and then incremented by 1. invoke is called.

op_int

As in op_invoke, op_int has a secondary entry point, op_intx, for operands encoded in the opcode. At the op_int entry point, a WORDSIZE value is fetched out of the instruction stream and loaded into Op. At the op_intx entry point, the Op value is decoded from the operand. The Op value is pushed on the stack and is followed by a D_INTEGER word, forming an integer descriptor.

op_line

Like op_invoke and op_int, op_line has a secondary entry point. The operand value is obtained and then moved into _line.

op_llist

llist is similar to invoke in that it has a number of arguments already on the stack and that the operand specifies the number. The operand is fetched into Op and pushed on the stack to become the *nargs* argument of llist. Op is then multiplied by 2 and incremented by 1 to serve as an argument list size for calls.

op_mark

The operand is fetched into Op and ipc is added to it. efp is pushed on the stack and the new sp value is put in efp. gfp is pushed on the stack and cleared. Op is pushed on the stack.

op_mark0

Like op_mark, with an implicit operand value of zero.

op_pop

The two tstl instructions serve to add 8 to sp which removes the top value from the stack.

op_sdup

The descriptor on the top of the stack is pushed on the stack, duplicating it.

op_unmark

The operand, the number of expression frames to remove from the stack, is fetched into Op. efp is restored from the current expression frame. The instruction

```
sobgtr  Op,unmkjmp
```

decrements Op and then branches to dounmark if Op is not zero. This chains through the number of expression frames specified by the operand. gfp is restored from the current expression marker. efp is loaded into sp to move the expression marker to the top of the stack. Finally, efp is restored from the marker and sp is incremented to remove the last word of the marker.

op_unmk1-7

Similar to unmark, but uses successive restorations of efp rather than a loop.

op_global

Dual entry points are used to deal with possible operand encoding. The operand, which is a number of a variable in the global region, is multiplied by 8 to provide a byte offset from the start of the global region. The sum of Op and the value of globals is pushed on the stack to provide a descriptor address. The constant D_VAR is pushed on the stack to complete the descriptor for the global variable.

op_static
Identical to op_global except that statics is used instead of globals.

op_local
The operand value is the number of a local variable for which a variable descriptor is to be pushed on the stack. Recall that the local variables lie below the procedure frame and, on the VAX, the descriptor for the first one is at -16(fp). Op is negated. The instruction

```
pushaq  -16(fp)[Op]
```

performs the calculation

```
-16 + fp + (Op * 8)
```

which computes the address of the descriptor of the desired variable and pushes it on the stack. The variable descriptor is completed by pushing D_VAR on the stack.

op_arg
Like op_local, but it uses 8(ap) as the base for the address calculation and the operand value is not negated.

quit
Push a 0 on the stack and call the routine _c_exit to terminate execution of the Icon program.

err
err should never be encountered during normal execution. Reaching it indicates that an invalid opcode was encountered. It need not do anything more than abort execution. On the VAX, it calls sprintf to create a string containing the invalid opcode and the ipc where it was encountered and then calls syserr with the string as an argument.

3.7.5 lib/efail.s

Overview

efail handles the failure of an expression. When Icon evaluates an expression, it tries to produce a result from it. If at some point in the evaluation of an expression the expression fails, Icon resumes inactive generators in the expression in an attempt to make the expression succeed. efail is at the heart of this activity.

efail has three distinct outcomes:

(1) Resumption of the newest inactive generator in the current expression frame.

(2) Failure of the current expression with execution continuing at the failure address contained in the expression marker.

(3) Failure of the current expression with propagation of failure to the enclosing expression frame. This is similar to (2), but occurs when the failure address is 0. After the current expression fails, efail loops back to its entry point to fail again.

efail is branched to rather than being called. This is because it serves as a "back-end" for several failure actions that may occur during the course of execution:

(1) When a built-in procedure fails, it calls the routine fail, which in turn branches to efail.

(2) When an Icon procedure fails via the pfail routine, pfail terminates by branching to efail.

(3) When the efail opcode is executed by the interpreter, efail is branched to.

(4) The generator frames built by esusp and lsusp use efail as a return address. This is explained in detail later.

Generic Operation

efail is essentially a simple routine. There are two separate paths of execution that efail may take. The first is to resume an inactive generator. The second is to cause failure of the expression in lieu of an inactive generator.

If there is an inactive generator in the current expression frame, it must be resumed. If the generator is an Icon procedure and tracing is in on, **atrace** is called with appropriate arguments. **_k_level**, **_line**, and **_file** are restored from the generator frame. A return is performed and the net result is that the stack is restored to the state that it was in before the suspension that created the generator.

If there are no inactive generators that can be resumed, the expression being evaluated must fail. This is done by popping the stack back through the current expression frame and resuming execution at the point indicated by the failure address in the expression marker. This is a two-step process. The first is to pop the frame and the second is to resume execution. When the frame is popped, the expression has failed. The failure address in the expression marker is saved before the frame is popped. If this address is not zero, execution is continued by branching to the address. If the address is zero, the failure is propagated to the enclosing expression by branching to efail.

Zero failure addresses are generated by the ucode instruction

```
mark   L0
```

Such instructions are used to avoid a special case during code generation and the only purpose they serve during program execution is to create an expression marker for the corresponding **unmark** instruction to remove. (**mark** and **unmark** instructions are paired.) Thus, whenever **efail** pops an expression whose marker has a zero failure address, **efail** causes failure in the enclosing expression.

efail on the VAX

The first action is to determine if there is an inactive generator that can be reactivated. If the generator frame pointer is non-zero, it points to the newest inactive generator. Note that whenever a new expression frame is created, the generator frame pointer is zeroed. Thus, if **gfp** is non-zero, the generator frame that it points to belongs to a generator in the current expression frame.

If an inactive generator is available, it must be reactivated. First, **_boundary** is restored from the generator frame. The stack is popped back to the generator frame by loading **fp** from **gfp**. But, before **fp** is loaded, its value is saved in **r0**. **fp** now points at word 0 of the generator frame, but that is a word below the actual stack frame that it should be pointing at, so **fp** is incremented by 4 using a **tstl**.

There are three types of generators that may be encountered by **efail**.

(1) An Icon procedure that did a **suspend**. In such cases, the routine **psusp** handled the suspension.

(2) A built-in procedure that called the C function **suspend()**.

(3) A generator created by an **esusp** or **lsusp** instruction. Such generators arise from source code constructs like $expr_1$ | $expr_2$, |*expr*, and $expr_1$ \ $expr_2$, which are referred to as *control regimes*.

The generators may be treated the same way as far as resumption goes. However, if an Icon procedure is being resumed, a tracing message must be generated if **_k_trace** is not 0.

If the value of **_boundary** is not the same as **fp**, the generator is a built-in procedure and tracing is not done. If the **fp** saved in the current frame is the same as the **fp** was upon entry to **efail** (the value was saved in **r0**), the generator was made by an **esusp** or an **lsusp** and tracing is not done.

Otherwise, the generator is an Icon procedure, and **atrace** must be called. **atrace** takes one argument, the address of the procedure block for the procedure being resumed. Recall that arg_0 on the stack is a descriptor for the procedure block. The address of arg_0 is calculated using

$$\&arg_0 = \text{ap} + 8 + (8 * \text{nargs})$$

The resulting address is used as the single argument for **atrace**.

The generator is now ready to be resumed. **_k_level**, **_line**, and **_file** are restored by popping them from the generator frame. If the generator is a built-in procedure, **_boundary** is cleared. A return is performed to activate the generator.

The return has different effects depending on the type of generator being resumed.

If the generator is a built-in procedure, the return restores the stack to the state it was in before **suspend** was called, and execution proceeds at the point just after **suspend()**. In this case the **pc** value being returned

to references an instruction in the built-in procedure.

If the generator is an Icon procedure, the stack is restored to the state it was in before the psusp ucode instruction was executed. The pc value being returned to references an instruction in the interpreter loop. Execution of the program continues with the interpreter instruction following the psusp.

If the generator is a control regime, the stack is restored to the state it was in before the esusp or lsusp that created the generator was performed. The return pc points to efail itself. Thus, when the return is done, the stack is cleared, and an efail is performed. This has the effect of transferring control to the failure label in the expression marker of the bounding expression frame.

If there is no generator to reactivate, the expression must fail. This is handled at the label nogen. efp points to the expression frame marker. ipc is loaded from -8(efp) which contains the address to go to in the event that the current expression fails. (As it has.) gfp is restored from the expression marker. efp is restored from the marker and the marker is popped off the stack.

If the failure address in ipc is non-zero, control is passed back to the interpreter via a branch and execution of the ucode resumes at the failure address. If ipc is zero, the expression failure is transmitted to the surrounding expression frame by a branch to efail. (Recall that a zero failure address comes from a mark L0 instruction and that a failure that reaches a mark L0 marker must be propagated to the next expression marker.)

3.7.6 lib/pfail.s

Overview

pfail handles the failure of an Icon procedure. An Icon procedure can fail by

executing a fail expression
executing return *expr* when *expr* fails
allowing the flow of control to reach the end of a procedure

pfail is entered via a branch when the interpreter encounters the pfail instruction.

Generic Operation

The task of pfail is to signal failure in the expression that contains the procedure call being evaluated. This is done by removing the Icon procedure frame from the stack, restoring appropriate registers and values, and calling efail. The key is that all pfail needs to do is to remove the procedure frame from the stack and from then on things can be handled just like expression failure. Thus, efail does most of the work.

pfail calls ftrace to produce a trace message if tracing is on. pfail also decrements _k_level because a procedure is being exited.

Note that the procedure frame on the stack is a frame that was created by invoke.

pfail on the VAX

After _k_level is decremented, _k_trace is checked to see if a trace message should be produced. If tracing is on, ftrace must be called. ftrace takes one argument, the address of the procedure block for the failing procedure. arg_0 is the descriptor for the procedure block, and the address of arg_0 is calculated using

$$\&arg_0 = (nargs * 8) + 8 + ap$$

The resulting address is pushed on the stack and ftrace is called.

Execution continues at dofail to remove the procedure frame from the stack. The frame cannot be merely popped because it contains state information that must be restored. _line and _file are extracted from the frame. efp, gfp, and ipc are restored from the frame using addresses relative to ap. Note that this works because invoke saves these registers and their location is known.

ap and fp are restored from the frame. When fp is restored, it serves to remove the procedure frame (made by invoke) from the stack. At this point, the stack is the same state it was in before the interpreter performed the invoke instruction. A branch is made to efail to cause failure in the enclosing expression.

3.8 Testing the Basis

At this point, enough of the system has been written to run some very simple Icon programs. test/hello.icn should be functional and more complete testing is in order. Refer to *Testing the Basis* in [5].

3.8.1 rt/arith.s

Overview

arith.s contains code for routines that add, subtract, and multiply integers and check for overflow. If overflow occurs, run-time error 203 is produced.

The arguments to ckadd, cksub, and ckmul are two C long integers on which to operate. For example, if ckadd were written in C, it would be declared

```
long ckadd(a,b)
long a,b;
{
...
}
```

The routines return the result of the operation using standard C return conventions.

arith on the VAX

The two arguments appear on the stack; a is at 4(ap) and b is at 8(ap). The appropriate 3-operand VAX instruction is used to perform the operation and the result is placed in r0 in accordance with C return conventions. If overflow occurs during the operation, the overflow bit in the program status word is set.

After the operation is performed, the overflow bit is checked. If it is on, indicating that an overflow occurred, a branch is taken to oflow, where runerr(203,0) is called. If overflow did not occur, the routine returns and the value in r0 is the value returned to the calling expression.

arith.s is trivial on the VAX because the hardware supports operations on C long integers. This may not be the case on the target machine. If so, arith.s will be considerably more complicated. However, it usually is not difficult to locate routines that perform these functions. As a last resort, look at the code the C compiler generates for the various arithmetic operations on long integers.

3.8.2 rt/fail.s

Overview

fail handles the failure of built-in procedures and operations. Built-in procedures and operations are C routines and they signal failure by calling fail(). When a failure of this type occurs, the failure must be transmitted to the Icon expression whose evaluation is in progress and that requires the services of a C routine.

Generic Operation

fail itself does very little, the real work is done by efail. fail pops the stack back to where it was before the C routine was called and then branches to efail to make the enclosing expression fail.

fail is akin to pfail in that it pops the stack back to a state that it was in when an expression was being evaluated and then causes failure of the expression. The primary difference is that an Icon procedure frame is being removed in pfail and it contains extra information that must be restored.

fail on the VAX

_boundary points to the procedure frame for the first C routine that was called from Icon. fp is loaded from _boundary and this puts the stack back to the state that it was in when the C routine was entered. The C routine is for either a built-in procedure such as read or for an operator such as +. For a built-in procedure, the procedure frame now on the top of the stack (after loading fp from _boundary) is the frame constructed in invoke. For an operator, the frame on the stack is the one constructed when the interpreter loop called the

C routine for the operator.

The task at hand is to remove the procedure frame and restore the registers that were saved in the frame. The mask/psw word of the frame is manipulated so that the mask portion of the word resides in bits 0:11 of r0 and the remaining bits of r0 are 0. The VAX popr instruction takes a register mask as an operand and pops words from the stack into the registers indicated by the mask. For example,

```
popr    $0x0005
```

moves the top word of the stack into r0 and the second stack word into r2. sp is then incremented by 8.

The saved registers start at 20(fp) and sp is loaded with this address. Then popr r0 restores the registers that are saved in the frame. Note that the manipulations of the mask/psw are necessary because it is not known *a priori* which registers were saved. In particular, popr $0x0fff would be disastrous.

After the registers have been restored, ap and fp are restored from the saved ap and fp values in the frame.

At this point, the stack is as it was before the frame for the built-in procedure or operator was created. All that remains is to signal failure in the expression being evaluated and this is done by branching to efail.

3.8.3 lib/pret.s

Overview

pret handles the return of a value from an Icon procedure. pret is called from the interpreter loop with a single argument (which is on the stack) that is the value being returned. The value is dereferenced if necessary. If tracing is on, a trace message is produced. The return value is copied over arg_0 in the frame of the procedure that is returning a value. The procedure frame is removed, leaving the result on the stack, and pret returns.

Generic Operation

(1) &level is decremented because a procedure is being exited.

(2) The stack address where the return value is to be placed is calculated. Recall that when a procedure is invoked, the return value (if any) ultimately replaces arg_0, the descriptor for the procedure returning the value.

(3) The value being returned must be dereferenced if it is a local variable or argument. This is because local variables and arguments are on the stack and the portion of the stack associated with a procedure "goes away" when a procedure returns. If the return value is a variable (is of type T_VAR) and its address is between the base of the current expression stack*, and the stack pointer, it is dereferenced. If it is a substring trapped variable (is of type T_TVAR and points to a block of type T_TVSUBS), and the address of the variable containing the substring is between the base of the current expression stack and the stack pointer, it is dereferenced.

(4) If &trace is non-zero, rtrace is called with the address of the block for the returning procedure and the address of the value being returned.

(5) fp, _line, and _file are restored from the returning procedure. Because the impending return restores control to the Icon environment, _boundary is cleared.

(6) pret returns from the Icon procedure by executing a return instruction. Because the current fp points to the procedure frame for the Icon procedure, and the frame was built by invoke, the return is effectively a return from invoke and the net result is the return value left on the stack.

*For purposes of uniformity, the system stack is treated as if it were a co-expression stack. The global variable current is a pointer to the descriptor for the co-expression stack block for the current co-expression. Co-expressions need not be implemented; it is only important that current and the descriptor that it points to be initialized correctly. This is done in iconx/init.c.

pret on the VAX

pret does not save any registers because the frame built upon entry to pret is discarded.

_boundary is set because deref may be called and deref can cause a garbage collection.

_k_level is decremented because a procedure is being exited.

pret needs to know the address of the descriptor for the Icon procedure that is returning. Before pret was called, fp pointed at the procedure frame for the current Icon procedure (which is now returning), and ap pointed at its argument list. The call to pret created a procedure frame which contains the old ap. This value is extracted and used in conjunction with *nargs* to calculate the address of the descriptor for the procedure. This value is stored in r11 for future use.

As described, the value being returned needs to be dereferenced in certain cases. The return value is a descriptor and is the argument to pret. The first word of this descriptor lies at 8(ap) and contains type and flag information. This word is placed in r1 for further examination.

The VAX bitl instruction tests a set of bits. The two operand values are ANDed together and the condition codes are set according to the value of the result. The result itself is discared. The instruction

```
bitl    $F_NQUAL,r1
```

ANDs the type and flags word with the F_NQUAL mask. The F_NQUAL bit is set if a descriptor is *not* a string qualifier. If the F_NQUAL bit is not on, the result of the AND is a 0. The test is followed by

```
beql    chktrace
```

Thus, if the return value *is* a string qualifier, a branch is taken to chktrace, and no dereferencing is performed.

If the return value does have the F_NQUAL attribute, it is checked to see if it is a variable. The F_VAR bit is tested. If it is not on, the return value is not a variable and does not have to be dereferenced. A branch is made to chktrace if this is the case.

If a variable is in hand, the F_TVAR bit is checked to see if it is a trapped variable. If it is not a trapped variable, the address field of the return value's descriptor is moved into r1 for further testing and a branch is taken to chkloc.

If the return value is a substring trapped variable, it may reference a local variable or an argument. The type bits of the descriptor are isolated by ORing it with TYPEMASK. If the type is not T_TVSUBS, no dereferencing is needed and a branch is taken to chktrace. If it is a substring trapped variable, the address of the variable containing the substring is obtained from the trapped variable's data block and is loaded into r1.

At this point (chkloc), r1 points to a descriptor that is directly or indirectly referenced by the return value. If the descriptor is in the current expression stack, the return value must be dereferenced. r1 is first compared to sp. If it is less than sp, the descriptor is below the stack and a branch is made to chktrace. Otherwise, r1 is compared to the base address of the current expression stack. If r1 is greater than the base of stack, it is above the stack and a branch is made to chktrace.

It is now certain that the return value must be dereferenced, lest it "disappear" when the portion of the stack it is in is re-used. The address of the return value is pushed on the stack and deref is called. Note that deref completely handles dereferencing of substring trapped variables and thus no special provisions need to be made.

At chktrace, the return value has been dereferenced if necessary and it is time to produce a tracing message if &trace is non-zero. rtrace does the work and it requires two arguments: the address of the block for the returning procedure, and the address of the return value. Earlier, the address of descriptor for the procedure block was calculated and left in r11. The second word of this descriptor is pushed on the stack along with the address of the return value.

pret "returns" the designated value by overwriting the procedure's descriptor with the descriptor of the return value. r11 points at the descriptor for the procedure and 8(ap) still points at the descriptor for the return value, so

```
movq    8(ap),(r11)
```

does the trick.

Everything is set, the actual return must be performed. The fp saved in the current frame is the procedure frame pointer of the Icon procedure and the saved value is loaded into fp to bring the Icon procedure frame to the top of the stack. _line and _file are restored from the Icon procedure frame. _boundary is cleared because the return will take execution back into an Icon realm.

A ret is executed. The return goes through the procedure frame built by invoke. Thus, control is returned to the point just after the call to invoke and it appears as if invoke itself had just returned.

3.8.4 lib/esusp.s

Overview

esusp suspends a value from an expression. esusp is called from the interpreter loop and the value to suspend appears as an argument. A generator frame hiding the current expression is created. The surrounding expression frame is duplicated. esusp leaves the value being suspended on the top of the stack.

The esusp operation arises from the alternation ($expr_1$ | $expr_2$) control structure. For example

```
p(5 | 10)
```

indicates that the call p(5) should be made and if it fails, then p(10) should be called.

The function of esusp is best explained using an example. The following ucode is generated for p(5 | 10)

```
        mark    L1
        var     0       (the variable p)
        mark    L2
        int     0       (constant 5)
        esusp
        goto    L3
lab L2
        int     1       (constant 10)
lab L3
        invoke  1
        unmark  1
lab L1
```

When execution reaches esusp, the stack looks like

```
                expression marker with L1 as failure address
                descriptor for variable p
efp  ⟶          expression marker with L2 as failure address
sp   ⟶          descriptor for constant 5
```

gfp is zero at this point. After the esusp is performed, the stack is

```
efp  ⟶          expression marker with L1 as failure address
                descriptor for variable p
                expression marker with L2 as failure address
                descriptor for constant 5
gfp  ⟶          generator frame built by esusp
                descriptor for variable p          } duplicated region
sp   ⟶          descriptor for constant 5
```

A branch is taken to L3, where invoke 1 is performed. This invokes p with one argument, the constant 5 on the stack. If p(5) succeeds, the unmark 1 is performed and the stack is popped back through the L1 expression frame, the current location of efp.

Suppose that instead of succeeding, p(5) fails. p fails by calling pfail, which removes the procedure frame from the stack and then calls efail. The previous stack picture shows what the stack looks like after the procedure frame has been removed. efail finds that gfp is not null and restores certain values that are saved in the generator frame. The frame, which was created by esusp, contains a return address that points to efail. Thus, when efail removes the frame by returning through it, control goes back to the start of efail and the stack is

```
                expression marker with L1 as failure address
                descriptor for variable p
efp  ⟶          expression marker with L2 as failure address
sp   ⟶          descriptor for constant 5
```

This time around, gfp is zero, so efail must remove the current expression frame and branch to the failure address in the frame's marker. When the expression frame is removed, the stack looks like

```
                expression marker with L1 as failure address
sp   ⟶          descriptor for variable p
```

The failure address in the expression frame was L2, so control is transferred to label L2 in the ucode. (Note how much went on as the result of the invoke being executed.) The instruction int 1 is executed and a descriptor for the constant 10 is pushed on the stack giving:

```
                expression marker with L1 as failure address
                descriptor for variable p
sp   ⟶          descriptor for constant 10
```

invoke 1 is performed again, which does p(10).

If p(10) succeeds, the unmark 1 is executed, which removes the L1 marker and transfers control to L1. If p(10) fails, the same thing happens, but efail does the work rather than unmark.

Generic Operation

(1) The procedure frame created by the call to esusp partially forms the generator frame. The frame is completed by pushing _boundary, _k_level, _line, and _file. The generator frame pointer is set to point at the word of the frame which contains the boundary.

(2) The bounds of the expression frame to be duplicated are determined. The lower bound is the stack word above the current expression frame marker. The upper bound is dependent on efp and gfp values saved in the current expression marker. If the saved gfp is non-zero, the upper bound is the first word below the generator frame marker. If the saved gfp is zero, the upper bound is the first word below the expression frame marker referenced by the saved efp. In the example, this region only contains the descriptor for the variable p. The region is copied to the top of the stack. The stack pointer is adjusted to point to the new top of stack.

(3) The value being suspended is pushed on the stack.

(4) The return address in the new generator frame is replaced by the address of efail so that when efail removes the frame by returning through it, efail regains control. The old return address is momentarily retained. The procedure frame pointer and argument pointer are restored. _boundary is cleared because control is returning to Icon code.

(5) efp in the current expression marker replaces the expression frame pointer. Thus, if an unmark is performed, the entire expression frame is removed. In the example, this happens if p(5) or p(10) succeeds.

(6) The return pc value which was saved is jumped to. This is in effect a return from esusp, but the stack is untouched.

esusp on the VAX

When esusp is entered, the generator frame is partially constructed. ipc, gfp, and efp are saved in the frame. _boundary is set to the current fp value and is pushed on the stack. The generator frame pointer, is pointed at the word containing the boundary. The frame is completed by pushing _k_level, _line, and _file on the stack.

The lower bound of the region to copy is the first word above the current expression frame marker. Recall that an expression frame looks like

efp →	0	old expression frame pointer
	-4	old generator frame pointer
	-8	failure address

Thus,

```
addl3   $4,efp,r0
```

points r0 at the lower end of the region to copy.

The upper bound of the region to copy is the low word of the marker for the enclosing generator or expression frame. If gfp is non-zero the generator frame marker is used. Otherwise, the expression frame marker is used. Recall that a generator frame looks like

		saved registers
	20	reactivation address (saved pc)
	16	saved fp
	12	saved ap
	8	psw and register mask
	4	0
gfp →	0	boundary
	-4	saved _k_level
	-8	saved _line
	-12	saved _file

So, if the saved gfp is non-zero, the upper bound of the region to copy is

saved gfp - 12

Otherwise, it is

saved efp - 8

The appropriate calculation is performed and r2 pointed at the bounding word.

At this point, the stack looks something like

r2 →		low word of expression or generator frame marker	
		last word of region to copy	
		...	
r0 →	4	first word of region to copy	
efp →	0	saved expression frame pointer	} expression marker
	-4	saved generator frame pointer	
	-8	failure label	
	8	descriptor for value to suspend	
	4	nargs (1)	
ap →	0	nwords (3)	
	-4	saved r11 (efp)	} generator marker
		saved r10 (gfp)	
	24	saved r9 (ipc)	
	20	reactivation address (saved pc)	
	16	saved fp	
	12	saved ap	
	8	psw and register mask	
	4	0	
	0	boundary (fp at entry to esusp)	
	-4	_k_level	
	-8	_line	
sp →	-12	_file	

The region starting at r0 and extending to r2 is to be copied to the top of the stack. The length of the region in bytes is calculated in r2. The value of r2 is subtracted from sp, moving sp down to accommodate the region. The region is then copied using

```
movc3   r2,(r0),(sp)
```

which moves r2 bytes starting at 0(r0) to 0(sp).

The descriptor for the value to suspend is at 8(ap) and it is pushed on the stack using

```
movq    8(ap),(sp)
```

The stack now looks like

r2 →	low word of expression or generator frame marker
	last word of region to copy
	...
r0 →	first word of region to copy
efp →	expression frame marker
8(ap) →	descriptor for value to suspend
	...
gfp →	generator frame marker
	last word of copied region
	...
	first word of copied region
sp →	descriptor for value to suspend

The reactivation address that is saved in the generator frame is moved into r1 for later use. It is then replaced by the address of efail so that when the frame is returned through, control will go to efail.

fp and ap are restored from the generator frame. _boundary is cleared because control is being returned to Icon code.

efp is pointed at the previous expression frame. That is, efp is moved back one step in the expression frame chain.

Control is returned to the interpreter loop by branching to 0(r1), the reactivation address originally saved in the generator frame.

3.8.5 lib/lsusp.s

Overview

lsusp suspends a value from a limited expression. A limited expression arises from a source code expression of the form

$$expr_1 \setminus expr_2$$

This limits $expr_1$ to at most $expr_2$ results. ($expr_2$ must be a non-negative integer.)

lsusp is just like esusp except that it has provisions for checking and decrementing the limit counter and taking the appropriate action when the counter reaches zero. As a simple example, consider

```
p(x \ 2)
```

which generates the ucode

```
mark    L1
var     0       (variable p)
int     0       (constant 2)
limit
mark    L0
var     1       (variable x)
lsusp
invoke  1
unmark  1
...
```

When control reaches the lsusp, the stack looks like

```
        expression marker with L1 as failure label
        descriptor for variable p
        descriptor for integer with value of 2
efp ->  expression marker with L0 as failure label
sp  ->  descriptor for variable x
```

The limit instruction insures that the value on the top of the stack (its argument) is a non-negative integer. After lsusp, the stack is

```
efp ->  expression marker with L1 as failure label
        descriptor for variable p
        descriptor for integer with value of 2
        expression marker with L0 as failure label
        descriptor for variable x
gfp ->  generator frame built by lsusp
        descriptor for variable p               } duplicated region
sp  ->  descriptor for variable x
```

This is the same thing that esusp would do, with the exception that the limit counter, the integer descriptor, is not part of the duplicated region.

Generic Operation

(1) The procedure frame created by the call to esusp partially forms the generator frame.

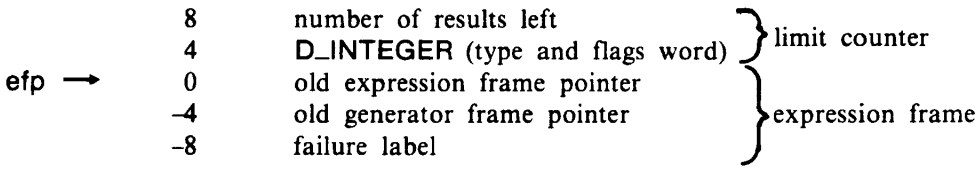
(2) The limit counter is decremented and if it is zero, no suspension is performed. Instead, the current expression frame is removed and the limit counter is replaced by the value that would have been suspended had the limitation not been in effect. lsusp returns, leaving the value on the top of the stack.

(3) If the limit counter is not zero, execution proceeds exactly as it does for esusp with the exception that the determination of the region to copy takes the limit counter into consideration and does not include it in the region that is copied.

lsusp on the VAX

When lsusp is entered, the generator frame is partially constructed. ipc, gfp, and efp are saved in the frame.

The expression frame and associated limit counter have the following layout:



The limit counter is decremented and if it is not zero, control passes to dosusp: and from then on execution proceeds exactly as it does in esusp. Specifically, the code beginning at dosusp: is an exact duplicate of that in esusp with the exception of the instruction that determines the lower bound of the region to be duplicated. esusp uses

```
addl3 $4,efp,r0
```

which points r0 at the word immediately above the expression frame. lsusp uses

```
addl3 $12,efp,r0
```

which points r0 at the word above the limit counter that is directly above the expression frame.

If the limit counter is zero, the counter is to be replaced with the value which was to be suspended. The value appears as an argument to lsusp. This is accomplished with

```
movq    8(ap),4(efp)
```

The value of gfp that is stored in the expression frame is restored.

The saved pc in lsusp's frame is moved into r0 for later use.

The expression frame is removed by moving efp into sp, which leaves the expression frame marker word that contains the old efp on the top of the stack. This word is popped of the stack and moved into efp, restoring efp and leaving the return (would-be suspended) value on the top of the stack.

ap and fp are restored from the procedure frame made upon entry to lsusp.

lsusp "returns" by jumping to 0(r0), the return point that was saved in the frame. The value that was to be suspended, but was not because of the limitation, is left on the top of the stack.

3.8.6 lib/psusp.s

Overview

psusp suspends a result from an Icon procedure. psusp is called from the interpreter loop and the value to suspend appears as an argument to psusp. A generator frame is created and the generator or expression frame immediately containing the frame for the suspending procedure is duplicated on top of the stack. psusp returns through the duplicated frame, leaving the suspending value on top of the stack. A return from psusp is manifested as a return from invoke.

The psusp operation arises from the suspend *expr* statement.

psusp is conceptually similar to esusp, the difference being that a procedure frame is part of the expression frame being duplicated and that requires some extra work. To get a feel for what psusp does, consider a simple example:

```
procedure main()
        f(p(3))
end

procedure p(a)
        suspend a
end
```

The generated ucode for main is

```
                ...
                mark        L1
                var         0           (the variable f)
                var         1           (the variable p)
                int         0           (the constant 3)
                invoke      1
                invoke      1
                ...
lab             L1
```

and the generated code for p is

```
                mark        p.L1
                mark        L0
                var         0           (the argument a)
                psusp
                ...
lab             p.L1
```

When control reaches the invoke instruction, the stack resembles

```
efp ⟶   expression marker with L1 as failure address
        descriptor for variable f
        descriptor for variable p
sp  ⟶   descriptor for constant 3
```

after p has been invoked, just before the psusp is executed the stack is

```
        expression marker with L1 as failure address
        descriptor for variable f
        descriptor for variable p
        descriptor for constant 3 (becomes argument a)
        procedure frame for p (created by invoke)
        expression marker with p.L1 as failure address
efp ⟶   expression marker with L0 as failure address
sp  ⟶   descriptor for argument a
```

Just before control returns from psusp, the stack is

```
                expression marker with L1 as failure address
                descriptor for variable f
                descriptor for variable p
                descriptor for constant 3
                procedure frame for p
                expression marker with p.L1 as failure address
                expression marker with L0 as failure address
                descriptor for argument a (being suspended)
gfp ⟶           generator frame built by psusp
                descriptor for variable f          ⎫
                descriptor for variable p          ⎬ duplicated region
                descriptor for constant 3          ⎭
sp  ⟶           procedure frame for p
```

After psusp returns, the situation is

```
efp ⟶           expression marker with L1 as failure address
                descriptor for variable f
                descriptor for variable p
                descriptor for constant 3
                procedure frame for p
                expression marker with p.L1 as failure address
                expression marker with L0 as failure address
                descriptor for argument a
gfp ⟶           generator frame built by psusp
                descriptor for variable f
sp  ⟶           descriptor for constant 3 (the suspended value)
```

The return from psusp goes to the second invoke, which calls f with one argument, the constant 3 that was suspended. If f(3) fails, the procedure frame for f is removed. efail takes control and returns through the generator frame built by psusp. This leaves the descriptor for a on top of the stack. Execution continues by p failing, and then main failing.

Generic Operation

(1) The procedure frame created by the call to psusp partially forms the generator frame. _boundary is set as the current location of fp and it is added to the generator frame.

(2) As in pret, the value being suspended must be dereferenced in certain cases. For example, if the value is a local variable or an argument, it is dereferenced. The same code that handles dereferencing in pret appears in psusp as well. Note that while suspension leaves the local variables and arguments of a procedure intact, if the enclosing expression frame should be removed by an unmark, the procedure frame would be destroyed, leaving undeferenced values pointing at meaningless data.

(3) The generator frame is completed by pointing gfp at the boundary value already in the frame and by adding _k_level, _line, and _file.

(4) The bounds of the expression frame to be duplicated are determined. The lower bound is the low word of the procedure frame for the suspending procedure and the upper bound is the marker for the expression frame or generator frame which is just prior to the procedure frame. As in esusp, if gfp is non-zero, the marker it points to is used. Otherwise, the marker referenced by efp is used. The gfp and efp values used are those found in the procedure frame of the suspending procedure. The region is copied to the top of the stack. In the example, the duplicated region contained the procedure frame marker for p, and the descriptors for the constant 3, and the variables p and f.

(5) If &trace is non-zero, strace is called to produce a trace message noting that the procedure is suspending a value. strace requires the address of the block for the suspending procedure and the address of the descriptor for the value being suspended.

(6) _line and _file are restored from the frame of the suspending procedure. This is done because when psusp is finished, it is as if the Icon procedure had returned. Thus, the line number and file name need to be what they were before the procedure was called.

(7) The generator frame pointer saved in the duplicated procedure frame on the top of the stack is replaced by the current value of gfp, which points to the newly created generator frame. When psusp returns through the frame on the top of the stack, gfp is restored from the value in the frame and gfp then references the new generator frame.

(8) The descriptor for arg_0 in the argument list of the Icon procedure that is suspending is a descriptor for the procedure itself. This descriptor is replaced with the descriptor for the value being suspended. When psusp is done, this descriptor is left on the top of the stack.

(9) _boundary is cleared because control is returning to Icon code.

(10) psusp returns. The return uses the duplicated procedure frame on the top of the stack. The result is that it appears as if the invoke that originally called the suspending procedure has returned.

psusp on the VAX

When psusp is entered, the generator frame is partially constructed as a result of the call. _boundary is set to the current value of fp and this value is pushed on the stack as part of the generator frame.

The value being suspended needs to be dereferenced if it is a local variable or an argument. This operation is the same as is done in pret; consult the section on it for details of the actions taken.

The generator frame is completed by pointing gfp at the frame word containing the boundary value and by adding _k_level, _line, and _file to the frame.

The region to be duplicated is determined. The low word to be copied is the low word of the procedure frame of the suspending procedure. (The word that contains the 0.) This is readily accessible as the fp saved the procedure frame of psusp and is placed in r7.

The high word to be copied is dependent upon the expression and generator environment of the suspending procedure. If the gfp in the suspender's environment is not zero, the word just below the generator frame marker is the highest word to be copied. If gfp is zero, the word just below the expression marker pointed at by efp in the suspender's environment is the highest word to be copied.

The fact that the saved efp and gfp appear on the stack just below the *nwords* word (referenced by 0(ap)) is used to retrieve and test them. As in esusp, if the saved gfp is non-zero,

saved gfp - 12

is used for the lower bound, otherwise

saved efp - 8

is the lower bound. r4 is pointed at the appropriate word on the upper end. As in esusp, sp is moved down to accommodate the region to be duplicated and the region is copied to the top of the stack using a movc3.

After _k_level is decremented, _k_trace is checked to see if a trace message should be produced. If so, strace is called with pointers to the descriptors for the suspending procedure and the value being suspended. The address of the value being suspended is 8(ap) and the address of the descriptor for the procedure is determined using the standard

$\&arg_0$ = (nargs * 8) + 8 + ap

calculation.

The values of _line and _file are restored from the suspender's frame.

The saved gfp in the duplicated procedure frame on the top of the stack must be replaced by the current value of gfp. Finding the location of the saved gfp is a little tricky. The distance between fp and ap in the duplicated frame is calculated by subtracting the value of fp from the ap value and putting the result in r0. The value in r0 represents the distance from the top of the stack to 0(ap). Adding the current sp (which points at the low word of the duplicated procedure frame) to r0 points r0 at the *nwords* word of the new

frame. ap normally points at the *nwords* word, so r0 serves as a pseudo-ap. It is known that the saved gfp is the second word below the *nwords* word; thus the new gfp is stored in -8(r0), replacing the old value. Thus, when psusp returns through the duplicated frame, the value just stored is the restored value of gfp.

The descriptor for the suspending procedure must be replaced by the descriptor for the return value. The previous calculation left r0 pointing at 0(ap) in the duplicated frame. The address of arg_0 is calculated using

$$\&arg_0 = (\text{nwords} * 4) + 4 + \text{r0}$$

Note that + 4 accounts for the four bytes that the *nwords* word itself occupies. The descriptor for the return value is put in place using a movq.

sp is moved into fp so that the pending return uses the new frame on the top of the stack.

_boundary is cleared because control is going back into Icon code.

A ret is executed to return from psusp. This return uses the duplicated procedure frame and thus the duplicated frame is removed. The final result is that it looks like the original call to invoke that started the Icon procedure has returned and the suspended value is left on the top of the stack.

3.8.7 rt/suspend.s

Overview

suspend suspends a value from a built-in procedure or operator. suspend is similar to psusp and amounts to little more than a simplified version of it. Recall that built-in procedures are C functions; thus, suspend is directly called from C.

A generator frame is created and the generator or expression frame immediately containing the frame for the suspending procedure is duplicated on the top of the stack. suspend returns through the duplicated frame, leaving the value being suspended on the top of the stack. When suspend returns, it appears as a return from the original call to the built-in procedure.

Note that suspend handles the suspension of values from both built-in procedures such as upto and from operators such as the element generation operator, !. For built-in procedures, the procedure frame is built by invoke, while for operators, the procedure frame is built directly by the call to the appropriate function from the interpreter loop. The value being suspended by the C function is represented by the arg_0 descriptor in the argument list. When suspend is called, the value to suspend is in place in arg0.

Generic Operation

suspend can be considered as a "subset" of psusp. The descriptions of operations in this section and the next are excerpts from psusp.

The actions of psusp that are *not* taken by suspend are:

(1) The return value does not need to be dereferenced because the suspending function has already taken care of that.

(2) No tracing message is produced because tracing is only done for Icon procedures.

(3) The return value does not need to be moved into the duplicated procedure frame because it is already in place in the original procedure frame and when the frame is duplicated, the return value is duplicated along with it.

(4) _k_level is not decremented because &level keeps track of Icon procedure calls and suspend is returning from a C routine. _line, and _file are not restored because they are not part of the procedure frame of the built-in procedure.

The operations that are performed by suspend are:

(1) The procedure frame created by the call to suspend partially forms the generator frame. _boundary is set as the current location of fp and it is added to the generator frame.

(2) The bounds of the expression frame to be duplicated are determined. The lower bound is the low word of the procedure frame for the suspending procedure and the upper bound is the marker for the expression frame or generator frame which is just prior to the procedure frame. As in esusp, if gfp is non-zero, the marker it points to is used. Otherwise, the marker referenced by efp is used. The gfp and efp values used are those found in the procedure frame of the suspending procedure. The region is copied to the top of the stack.

(3) The generator frame pointer saved in the duplicated procedure frame on the top of the stack is replaced by the current value of gfp, which points to the newly created generator frame. When suspend returns through the frame on the top of the stack, gfp is restored from the value in the frame and gfp then references the new generator frame.

(4) _boundary is cleared because control is returning to Icon code.

(5) suspend returns. The return uses the duplicated procedure frame on the top of the stack. The result is that it appears as if the original call to the suspending procedure has returned.

suspend on the VAX

When suspend is entered, the generator frame is partially constructed as a result of the call. _boundary is set to the current value of fp and this value is pushed on the stack as part of the generator frame. The generator frame is completed by pointing gfp at the frame word containing the boundary value and by adding _k_level, _line, and _file to the frame.

The region to be duplicated is determined. The low word to be copied is the low word of the procedure frame of the suspending function. (The word that contains the 0.) This is readily accessible as the fp saved in the procedure frame of suspend and r7 is pointed at the word containing the saved fp.

The high word to be copied is dependent upon the expression and generator environment of the suspending function. If the gfp in the suspender's environment is not zero, the word just below the generator frame marker is the highest word to be copied. If gfp is zero, the just word below the expression marker pointed at by efp in the suspender's environment is the highest word to be copied.

The fact that the saved efp and gfp appear on the stack just below the *nwords* word (referenced by 0(ap)) is used to retrieve and test them. As in esusp, if the saved gfp is non-zero,

saved gfp - 12

is used for the lower bound, otherwise

saved efp - 8

is the lower bound. r4 is pointed at the appropriate word on the upper end. As in esusp, sp is moved down to accommodate the region to be duplicated and the region is copied to the top of the stack using a movc3.

The saved gfp in the duplicated procedure frame on the top of the stack must be replaced by the current value of gfp. Finding the location of the saved gfp is a little tricky. The distance between fp and ap in the duplicated frame is calculated by subtracting the fp value from the ap value and putting the result in r0. The value in r0 represents the distance from the top of the stack to 0(ap). Adding the current sp (which points at the low word of the duplicated procedure frame) to r0 points r0 at the *nwords* word of the new frame. ap normally points at the *nwords* word, so r0 serves as a pseudo-ap. It is known that the saved gfp is the second word below the *nwords* word and thus the new gfp is stored in -8(r0), replacing the old value. Thus, when suspend returns through the duplicated frame, the value just stored is the restored value of gfp.

A ret is executed to return from suspend. This return uses the duplicated procedure frame and thus the duplicated frame is removed. The final result is that it looks like the original call to function has returned and

the suspended value is left on the top of the stack.

3.8.8 functions/display.c

Overview

display.c implements the Icon function display(). display traces back through Icon procedure frames printing various sorts of information. Therefore, some of the code in display is machine dependent.

Generic Operation

display makes one calculation that is machine dependent. The calculation is to take a frame whose address is contained in the variable fp and calculate the address of the procedure descriptor in the frame that is pointed at by the fp value saved in the frame that fp references.

display on the VAX

ap and fp are restored from the frame referenced by fp. The number of arguments to the procedure is contained in ap[1]. This is loaded into the variable n. The address of the procedure descriptor (arg_0) is calculated using:

```
dp = ap + 2 + 2*n
```

Note that this is the same computation that is made at several points in the assembly language routines. As in sweep, the calculations are being made using int * variables and thus the constants represent word counts instead of byte counts as they do in the assembly language routines.

3.8.9 rt/gcollect.s

Overview

gcollect is a simple routine that insures that garbage collections are done using the stack for the main co-expression. This done by saving certain values in the co-expression block of the current co-expression, restoring values from the co-expression block for &main, calling the garbage collector, and then restoring the original values. gcollect takes a single argument that is passed directly to collect.

If co-expressions are not implemented, gcollect need only consist of a call to collect, being sure to pass its argument on through.

gcollect on the VAX

r0 is pointed at the heap block for the current co-expression. sp, ap, and _boundary are saved in the appropriate words of the block.

r0 is pointed at the heap block for &main, the co-expression that is initially active. sp, ap, and _boundary are restored from values saved in the block.

The argument to gcollect is pushed on the stack, and collect is called with one argument.

r0 is pointed at the heap block for the current co-expression and the sp, ap, and _boundary values saved at the start of the routine are restored.

gcollect returns.

3.8.10 rt/sweep.c

Overview

sweep is used during garbage collection to sweep a stack, marking all the descriptors in the stack. sweep begins at the low word (the top) of a stack and moves up through the stack, looking for descriptors and marking them. A stack is composed of four kinds of objects: descriptors, and markers for procedure, generator, and expression frames. sweep uses knowledge of frame marker formats to skip over markers and to process

the intervening descriptors.

Although sweep is written in C, the knowledge of frame formats that it employs requires that it be written on a per-machine basis.

Generic Operation

There are three places that descriptors can appear on the stack: above an expression marker, in an argument list, and above an argument list. This can be considered as only two places because descriptors above the argument list can be considered as part of the argument list.

sweep is called with a single argument that is the address of the first word of a stack to mark. For purposes of discussion assume that sp references the stack word of current interest. sweep has a loop and each time through the loop, one of four actions is taken based on the word that sp is pointing at:

(1) If sp is pointing at the low word of a procedure frame marker, sp is moved to point at the low word of the argument list of the procedure. efp, gfp, and pfp are restored from the procedure frame. The number of arguments to the procedure is placed in *nargs*.

(2) If sp is pointing at the low word of a generator frame marker, fp is restored from the boundary word of the generator frame and sp is pointed at the low word of the frame referenced by fp.

(3) If sp is pointing at the low word of an expression frame marker, gfp and efp are restored from the marker and sp is pointed at the word above the marker.

(4) If none of the preceding conditions are true, the word that sp points at is assumed to be the low word of a descriptor and that descriptor is marked. sp is incremented by 2 to move past the descriptor. If *nargs* is not zero, it is decremented.

This process continues as long as fp and *nargs* are not both zero. *nargs* is used so that the arguments in the very last frame are processed. The fp at that point is 0.

sweep on the VAX

The routine getap is used by sweep. getap takes the address of a frame and returns the address of 0(ap) in that frame. That is, it returns the address of the start of the argument list for the frame.

Note that the C code uses int * variables for the various calculations that are performed. Thus, a calculation such as x + 2 is actually performing x + 8. Similarly, x[-1] would be the address x - 4.

sweep is called with a single parameter, fp. fp holds the address of the frame with which to start the marking process. This address is a _boundary value, and thus it points to the 0 (condition handler) word of a procedure frame.

sp is set to fp - FRAMELIMIT, so that the first time throughout the loop, the procedure frame on the top of the stack is processed. This gets the ball rolling, so to speak.

sweep loops while fp and *nargs* are not both zero. It should be noted that the variables used in sweep have no connection to actual registers other than having the same name.

If sp is equal to fp - FRAMELIMIT, it indicates that sp is pointing at a procedure frame marker. FRAMELIMIT is 2 on the VAX because there are two words, the saved values of _line and _file, that lie below the word in the frame that fp points at.

When a procedure frame marker is encountered, efp and gfp values are restored using negative displacements from ap. ap points at the *nwords* word of the frame, and sp is set to ap + 2 so that it points at the descriptor for the first argument. *nargs* is loaded from the argument list. ap and fp are restored from the frame

A generator frame is indicated by sp being equal to gfp - 3. This is because there are three words, _line, _file, and _k_level in the generator frame below the word that generator frame pointer points at. fp is restored from the frame. A new ap value is calculated from fp using getap. sp is set to fp - FRAMELIMIT to cause recognition of a procedure frame the next time around.

An expression frame marker is indicated by sp being equal to efp - 2. efp and gfp are restored from the marker. sp is incremented by 3 which leaves it pointing at the word above the marker, which may be a

descriptor.

If sp suits none of the preceding criteria, it is assumed to point at a descriptor. mark is called with the value of sp as its argument. sp is incremented by 2 to move past the descriptor just marked. If *nargs* is non-zero, it is decremented.

Acknowledgements

Ralph Griswold patiently suffered through a number of drafts of this document and made innumerable suggestions about grammar, form, and content. Steve Wampler graciously answered a number of questions about the internal workings of Icon and also made a number of comments on a late draft.

References

1. R. E. Griswold, R. K. McConeghy, and W. H. Mitchell, *A Tour Through the C Implementation of Icon; Version 5.9*, Technical Report 84-11, Department of Computer Science, The University of Arizona, August 1984.
2. *VAX Architecture Handbook*, Digital Equipment Corporation, Maynard, Massachusetts, 1982.
3. D. M. Ritchie, "A Tour Through the UNIX C Compiler", *UNIX Programmers Manual, Volume 2B*, Bell Telephone Laboratories, Inc., Murray Hill, New Jersey, 1979.
4. S. C. Johnson, "A Tour Through the Portable C Compiler", *UNIX Programmers Manual, Volume 2B*, Bell Telephone Laboratories, Inc., Murray Hill, New Jersey, 1979.
5. R. E. Griswold, *An Overview of the Porting Process for Version 5.9 of Icon*, Department of Computer Science, The University of Arizona, October 1984.