Combining Forth and the Rest of the Computer World: Dynamic Loading of Subroutines Written in Other Languages and Their Use as Forth Words.

William L. Sebok

Princeton University, Dept. of Astrophysics
Peyton Hall, Rm. 129, Princeton, NJ 08544

ABSTRACT

Dynamic subroutine loading has been added to the local Vax [1] implementation of Forth-79 that runs under the Berkeley dialect of Unix [2] (so called "Princeton Forth"). This paper is about the semantics of how this might be specified and a sketch of how it was implemented here. Also mentioned are some of the problems that must be faced.

Introduction

In an operating system there may exist many other computer languages. One can often construct a program from pieces in different languages, compiling each piece separately and using a linker to combine them into a larger program. However when one writes part of a program in Forth the rest of the program must be written in Forth. One has to play the Forth game all the way.

There are several reasons one might want to include a subroutine written in some other language inside a Forth program:

1) A subroutine might already exist written in some other language.
2) One may want to access operating system features in an operating system version independent way. For example, in the Unix operating system files intended to be loaded into C programs are provided that give symbolic names to various magic numbers and which define the layout of various data structures. The names of these files and the symbolic names contained within them are generally guaranteed to remain the same between versions of Unix, but the values of the symbolic names and the layout of the data structures may change between versions of Unix. It is difficult to make use of these files outside of the C language.
3) For heavy numerical calculations Forth may not be one's language of choice.

In Unix it is quite practical to have Forth words which run other programs as separate processes. These programs can be written in any language. However, this is not enough. In general, Forth shines when used to maintain a list of programs accessed from the keyboard (*i.e.* as set of Forth words). That these programs are resident in memory can be a big advantage over a more conventional array of disk resident programs, as access time is much shorter.

It is also possible to relink the Forth kernel to add new subroutines. However, doing so requires exiting Forth. It also may require detailed knowledge of the operation of the

[1] VAX is a trademark of Digital Equipment.
[2] UNIX is a Trademark of Bell Laboratories.

kernel. A method of dynamically loading new subroutines from within Forth is more desirable.

Such a method of dynamically loading subroutines has been created for Princeton Forth, an implementation of Forth-79 which runs on a Vax computer under the Unix operating system. [3], [4] This implementation of Forth was distributed over Usenet, the Unix network, during July, 1984. In this implementation floating point numbers and character strings have their own stacks and integers are 32 bits wide.

Syntax

Dynamic loading is invoked by the defining word *SUBROUTINE*. Executing the word defined by *SUBROUTINE* executes the externally loaded subroutine.

The syntax of *SUBROUTINE* is:

incastn ... incast2 incast1 nargs outcast
filen_s ... file2_s file1_s nfiles entrypoint_s SUBROUTINE name

In the notation used here arguments ending with the characters "*_s*" are character strings on the string stack. All other items are integers on the ordinary parameter stack.

entrypoint_s is the name of the subroutine to be invoked.

nfiles is the number of strings to be passed to the loader.

file1_s, *file2_s*, ... *file2_s* are the set of strings to be passed to the loader. These strings contain things like file names.

outcast is a number which indicates the type of item returned by the function. One possible value is *VOID_TY*, *i.e.* return no item.

nargs is the number of arguments that this subroutine takes.

incast1, *incast2*,... *incastn* are integers that indicate the type of each argument. Arguments are gathered from the appropriate stack and passed to the subroutine, *incast1* corresponding to the first argument, *incast2* corresponding to the second argument, etc.

The values of which the casts can take are defined as *CONSTANT*s.

Definition of Argument Type Casts

Type	value	stack location	description
void_ty	0	---	null type
addr_ty	1	parameter stack	address (a pointer to something)
int_ty	2	parameter stack	"standard" integer, *i.e.* 1 stack-cell size
dbleint_ty	3	parameter stack	two stack-cells wide
float_ty	4	floating point stack	floating point
dfloat_ty	5	floating point stack	double precision float
string_ty	6	string stack	character strings
char_ty	7	parameter stack	1 byte character
uchar_ty	8	parameter stack	1 byte character unsigned
short_ty	9	parameter stack	2 byte word signed
ushort_ty	10	parameter stack	2 byte word unsigned
long_ty	11	parameter stack	4 byte word signed
ulong_ty	12	parameter stack	4 byte word unsigned

The *SUBROUTINE* implementation keeps track of any global symbols that have been previously loaded. The names of these symbols are passed to any further invocation of the loader. Thus if two *SUBROUTINE*'s are loaded and both reference a third subroutine, only

one copy of the third subroutine will be loaded. Also, if *SUBROUTINE* is invoked requesting an entry point already present in previously loaded code no further code is loaded. Instead a reference is compiled to the previously loaded entry point.

Example

Suppose there exists a C library subroutine *cos* which takes a floating point number as an argument, takes the cosine of this number, and returns it as a floating point number. One desires to use this library routine to define a Forth word that does the equivalent operation, taking a floating argument and returning a floating point value. The definition that will do this is:

```
float_ty 1 float_ty \-lm 1 \_cos SUBROUTINE cos
```

Note that in this Forth implementation a backslash precedes a string constant which is placed on the string stack. The string -lm is a reference to the math library. Under Unix entry points in the C language have underscores appended to them. Thus the entry point name is "_cos".

Implementation

Loading takes place in 5 steps:

1) A dictionary header is constructed and an interface routine is compiled. This interface routine, 1) uses the supplied casts to gather the arguments from the various stacks and constructs a parameter list to pass to the subroutine, and 2) calls the subroutine. The return stack pointer (which in this implementation is the same as the hardware stack) is saved and the hardware stack is made the parameter stack. This is done because in this implementation the parameter stack is bigger than the return stack and one does not know how big a stack the called subroutine might need. Upon return from the subroutine the returned value is placed on the proper stack.
2) A temporary file which contains an image of Forth memory is constructed for the use of the loader. The file also contains a symbol table constructed from the symbol table kept in memory. This symbol table contains the symbols brought in by previously loaded subroutines and also contains the names of routines initially linked into the kernel. It is sufficient here to store zeroes (actually file system holes, which do not take up disk space) into the memory image as the loader only cares about the size of the image and the symbol table.
3) The loader ("ld") is invoked. Two special options exist in the loader which make this process easier. The first option is the -A option which allows one to specify a file whose symbol table is referenced but not actually loaded. The file constructed in the previous step is used with this option. The second option is the -T option that allows one to specify where the output of the loader is to start. The current high point of the Forth dictionary is used with this option.
4) The contents of the output of the loader are loaded into Forth memory.
5) The new symbols in the symbol table of the output of the loader are appended to the symbol table kept in Forth memory.

A Complication — *malloc* vs. *FORGET*

There is a complication that has had to be faced. This concerns this interaction of Forth with the standard Unix library subroutine *malloc*.

In this Unix implementation of Forth, growth of the Forth dictionary is handled transparently to the user. Under Unix, virtual memory is divided into two segments. One segment grows downward from high addresses. In this segment are placed the stacks, with the parameter stack occupying the leading edge. The other segment grows upward from low addresses. In this segment resides the Forth dictionary. The boundary of this segment is

extended or contracted by a Unix system call. This is done automatically by the Forth memory management routines.

Malloc allocates a block of memory. It takes a single argument which is the number of bytes of memory desired. It returns a pointer to the allocated block of memory. Also present is the routine *free* which frees a block of memory previously allocated by *malloc*. *Malloc* first tries to allocate memory from memory that has been previously freed. However, if this is not possible it will also issue the system call to move the leading edge of memory.

Thus there is a potential conflict between *malloc* and Forth memory allocation. It is quite possible to create a version of *malloc* which respects Forth memory allocations. However, a fundemental problem still remains.

This problem occurs when *malloc* is called and then a *FORGET* is issued. Suppose pointers to the *malloc*'ed memory are saved somewhere in the memory of the calling subroutine. These pointers are no longer valid after a *FORGET*. Attempts to use these pointers will cause an illegal memory access. The issue is made hard if one wants to use standard library routines which are unaware of Forth's use of memory. This problem has been solved in a way that imposes no new restrictions on either Forth or the non-Forth subroutine.

A call to *free* which frees memory adjacent to the dictionary pointer will cause the dictionary pointer to be moved back to the beginning of the area freed. A call to *malloc* which cannot be satisfied from existing free areas causes the dictionary pointer to be advanced. If a *FORGET* is issued and the memory to be forgotten contains one or more areas allocated by *malloc* and not *free*ed then in addition to its usual actions *FORGET* takes these actions:

1) The dictionary pointer is moved back to the end of the last area allocated by *malloc*.
2) The areas of memory that are above the *FORGET* point but are between the *malloc* allocated memory areas are marked "free" as if they had been freed by the library *free* routine. Thus this memory will be the first memory reused by further calls to *malloc*.

The net effect of all of this is that *malloc*, *free* and normal dictionary growth and *FORGET*ting can co-exist in the same memory, without the necessity of pre-allocation of fixed size areas of memory to either *malloc* or to dictionary growth.

Conclusion

A package like the one described here can allow Forth to live comfortably under Unix and make the best use of whatever tool is most suited to the task at hand. It is likely that similar methods are applicable to Forth under other operating systems.

Acknowledgements

I would like to thank Mitch Bradley and Jim Gunn for helpful discussions. As often is the case most of the time here was spent working on the syntax and the form of the interaction with the operating system rather than the actual coding.

References

[3] Sebok, W. L., 1984, *Rochester Forth Conference*, "A Vax Implementation of the Forth-79 Standards" (abstract)

[4] Sebok, W. L., 1984, *Journal of Forth Applications and Research*, "A Vax Implementation of the Forth-79 Standards", submitted.