A Portable Forth Engine

M. Anton Ertl

Institut für Computersprachen
Technische Universität Wien
Argentinierstraße 8, A-1040 Wien, Austria
anton@mips.complang.tuwien.ac.at
Tel.: (+43-1) 58801 4459
Fax.: (+43-1) 505 78 38

Abstract. The Forth engine discussed in this paper is written in GNU C, which provides several extensions that are important for Forth implementation. The indirect threaded Forth engine is completely machine-independent, direct threading requires a few machine-specific lines for each machine. Using a portable language like GNU C encourages producing an engine with many primitives. In order to make the development of primitives easier and less error-prone, an automatic tool generates most of the code for a Forth primitive from the stack effect notation, even if the TOS is kept in a register. The engine is combined with the parts of the system written in Forth by loading a machine-independent image file that contains the executable Forth code in relocatable form.

1 Introduction

When it became clear what the ANS Forth standard document would look like, a number of people on ForthNet expressed concerns that the standard would be too weak. They wanted a model-based standard, not an abstract programming interface. I.e., something like fig-Forth [Tin81] or F83, not Forth-79 or Forth-83.

In the summer of 1992 a team (then consisting of Bernd Paysan and me) set out to build a standard model for ANS Forth. So, the main design goals were that it is a model and that it will become a de-facto standard. To achieve the second goal, the system[1] should conform to the ANS for Forth, it should be efficient, powerful, freely available on a wide range of personal machines, and similar to the fig-Forth and F83 models.

This paper discusses the engine of this system, i.e. the part that is not written in (high-level) Forth.

2 Portability

One of the main goals of the effort is availability across a wide range of personal machines. fig-Forth, and to a lesser extent F83, achieved this goal by manually coding the engine in assembly language for several then-popular processors. This approach is very labor-intensive and the results are short-lived due to progress in computer architecture.

Others have avoided this problem by coding in C, e.g., [Pat90]. This approach is particularly popular for UNIX-based Forths due to the large variety of architectures of UNIX machines. Unfortunately an implementation in C does not mix well with the goals of efficiency and with using traditional techniques: Indirect or direct threading cannot be expressed in C, and switch threading, the fastest technique available in C, is significantly slower. Another problem with C is that it's very cumbersome to express double integer arithmetic.

Fortunately, there is a portable language that does not have these limitations: GNU C, the version of C processed by the GNU C compiler (`gcc`). Its *labels as values* feature makes direct and indirect threading possible, its *long long* type corresponds to Forths double numbers. GNU C is available for free on all important (and many unimportant) UNIX machines, VMS, 386s running MS-DOS, the Amiga, and the Atari ST[2], so a Forth written in GNU C can run on all these machines.

Writing in a portable language has the reputation of producing code that is slower than assembly. For our Forth engine we repeatedly looked at the code produced by the compiler and eliminated most compiler-induced inefficiencies. However, register allocation cannot be easily influenced by the programmer, leading to some inefficiencies on register-starved machines.

3 Threading

3.1 Basic Threading

GNU C's *labels as values* extension[3] makes it possible to take the address of *label* by writing `&&`*label*. This address can then be used in a statement like

[1] It does not have an official name yet. It is also described in [Pay93].

[2] Due to Apple's look-and-feel lawsuit it is not available on the Mac.

[3] available since `gcc-2.0`.

Paper and BibTeX entry are available at http://www.complang.tuwien.ac.at/papers/. This paper was published in: EuroFORTH '93 conference proceedings

goto **address*. I.e., goto *&&x is the same as goto x.

With this feature an indirect threaded NEXT looks like:

```
cfa = *ip++;
ca = *cfa;
goto *ca;
```

For those unfamiliar with the names: `ip` is the Forth instruction pointer; the `cfa` (code-field address) corresponds to ANS Forths execution token and points to the code field of the next word to be executed; The `ca` (code address) fetched from there points to some executable code, e.g., the code of the colon definition handler `docol`.

Direct threading is even simpler:

```
ca = *ip++;
goto *ca;
```

Of course we have packaged the whole thing neatly in macros called `NEXT1` (the part of NEXT after fetching the cfa) and `NEXT`. These code fragments are translated into perfect code on all machines we looked at. E.g., indirect threading on the MIPS R3000 RISC processor:

```
lw      $9,0($21)    ;cfa = *ip
#nop                 ;load delay slot
lw      $8,0($9)     ;ca= *cfa
addu    $21,$21,4    ;ip++
j       $8           ;goto *ca
#nop                 ;branch delay slot
```

For those who do not know the R3000 assembly language: `$9` means register 9, the result of operation is usually stored in the leftmost register and `lw` means *load word*. The `#nop`s indicate delay slots where independent instructions could be executed (see below).

3.2 Scheduling

There is a little complication: Pipelined and superscalar processors, i.e., RISC and modern CISC machines can process independent instructions while waiting for the results of an instruction. The compiler usually reorders (schedules) the instructions in a way that achieves good usage of these delay slots. However, on our first tries the compiler did not do well on scheduling primitives. E.g., for `+`

```
n=sp[0]+sp[1];
sp++;
sp[0]=n;
NEXT;
```

the NEXT came strictly after the other code, i.e., there was nearly no scheduling. After a little thought the problem became clear: The compiler cannot know that sp and ip point to different addresses, so it could not move the load of the cfa above the store to the TOS. Indeed the pointers could be the same, if code on or very near the top of stack were executed. In the interest of speed we chose to forbid this probably unused "feature" and helped the compiler in scheduling: NEXT is divided into the loading part (`NEXT_P1`) and the goto part (`NEXT1_P2`). `+` now looks like:

```
n=sp[0]+sp[1];
sp++;
NEXT_P1;
sp[0]=n;
NEXT1_P2;
```

This can be scheduled optimally by the compiler (see Section 4.2).

3.3 Direct or Indirect Threaded?

Both! After packaging the nasty details in macro definitions we realized that we could switch between direct and indirect threading by simply setting a compilation flag (`-DDIRECT_THREADED`) and defining a few machine-specific macros for the direct-threading case. On the Forth level we also offer access words that hide the differences between the threading methods.

Indirect threading is implemented completely machine-independently. Direct threading needs routines for creating jumps to the executable code (e.g. to docol or dodoes). These routines are inherently machine-dependent, but they do not amount to many source lines (13 for the R3000). I.e., even porting direct threading to a new machine is a small effort.

3.4 DOES

One of the most complex parts of a Forth engine is `dodoes`, i.e., the chunk of code executed by every word defined by a `CREATE`...`DOES>` pair. The main problem here is: How to find the Forth code to be executed, i.e. the code after the `DOES>` (the DOES-code)? There are two solutions (see Fig. 1):

In fig-Forth the code field points directly to the dodoes and the DOES-code address is stored in the cell after the code address (i.e. at cfa cell+). It may seem that this solution is illegal in the Forth-79 and all later standards, because in fig-Forth this address lies in the body (which is illegal in these standards). However, by making the code field larger for all words this solution becomes legal again. We use this approach for the indirect threaded version. Leaving a cell unused in most words is a bit wasteful, but on the machines we are targetting this is hardly a problem. The other reason for having a code field size of two cells is to avoid having different image

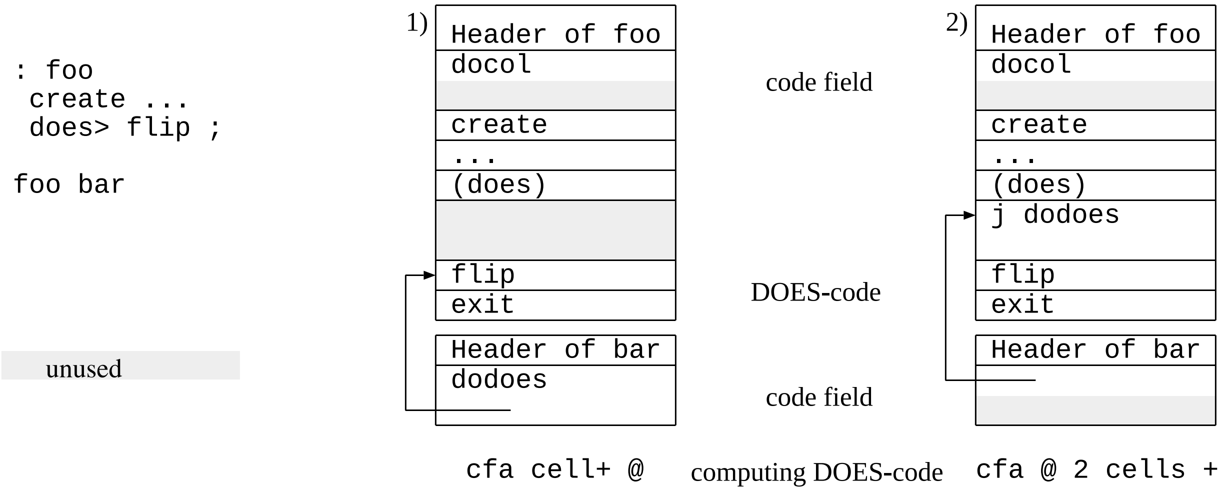

Fig. 1. The two approaches for `DOES>` on an indirect threaded engine.

files for direct and indirect threaded systems (see Section 5).

The other approach is that the code field points or jumps to the cell after `DOES>`. In this variant there is a jump to `dodoes` at this address. `dodoes` can then get the DOES-code address by computing the code address, i.e., the address of the jump to dodoes, and add the length of that jump field. A variant of this is to have a call to `dodoes` after the `DOES>`; then the return address (which can be found in the return register on RISCs) is the DOES-code address. Since the two cells available in the code field are usually used up by the jump to the code address in direct threading, we use this approach for direct threading. We did not want to add another cell to the code field.

4 The Primitives

4.1 Automatic Generation

Since the primitives are implemented in a portable language, there is no longer any need to minimize the number of primitives. On the contrary, having many primitives is an advantage: speed. In order to reduce the number of errors in primitives and to make programming them easier, we provide a tool, the primitive generator, that automatically generates most (and sometimes all) of the C code for a primitive from the stack effect notation. The source for a primitive has the following form:

Forth-name stack-effect category [pronounc.]
[""glossary entry""]
C code
[:
Forth code]

The items in brackets are optional. The category and glossary fields are there for generating the documentation, the Forth code is there for manual implementations on machines without GNU C. E.g., the source for the primitive `+` is:

```
+    n1 n2 -- n    core    plus
n = n1+n2;
```

This looks like a specification, but in fact `n = n1+n2` is executable C code. Our primitive generation tool extracts a lot of information from the stack effect notations[4]: The number of items popped from and pushed on the stack, their type, and by what name they are referred to in the C code. It then generates a C code prelude and postlude for each primitive. The final C code for `+` looks like this:

```
I_plus:              /* label */
{
Cell n1;      /* definitions */
Cell n2;
Cell n;
n1 = (Cell) sp[1];  /* input */
n2 = (Cell) TOS;
sp += 1; /* stack adjustment */
fp += 0;
{   /* taken from the source */
n = n1+n2;
}
NEXT_P1;      /* NEXT part 1 */
TOS = (Cell)n ;    /* output */
}
NEXT1_P2;     /* NEXT part 2 */
```

[4] We use a one-stack notation, even though we have separate data and floating-point stacks; The separate notation can be generated easily from the unified notation.

This looks long and inefficient, but the GNU C compiler optimizes quite well and produces optimal code for `+` on the R3000 (see Section 4.2) and the HP RISC machines: Statements like `fp+=0` produce no code; Defining the `n`s also does not produce any code, and using them as intermediate storage also adds no cost.

There are also other optimizations, that are not illustrated by this example: Assignments between simple variables are usually for free (copy propagation). If one of the stack items is not used by the primitive (e.g. in `drop`), the compiler eliminates the load from the stack (dead code elimination). On the other hand, there are some things that the compiler does not do, therefore they are performed by the generator: The compiler does not optimize code away that stores a stack item to the place where it just came from (e.g., `over`).

While programming a primitive is usually easy, there are a few cases where the programmer has to take the actions of the generator into account, most notably `?dup`, but also words that do not (always) fall through to NEXT.

4.2 TOS Optimization

An important optimization for stack machine emulators, e.g., Forth engines, is keeping (caching) one or more of the top stack items in registers. If a word has the stack effect $\mathtt{w}_1 ... \mathtt{w}_x$ `--` $\mathtt{w}_{x+1} ... \mathtt{w}_{x+y}$, keeping the top n items in registers

- is better than keeping $n-1$ items, if $x \geq n$ and $y \geq n$, due to fewer loads from and stores to the stack.
- is slower than keeping $n-1$ items, if $x \neq y$ and $x < n$ and $y < n$, due to additional moves between registers.

in particular, keeping one item in a register is never a disadvantage, if there are enough registers. Keeping two items in registers is a disadvantage for frequent words like `?branch`, constants, variables, literals and `i`. Therefore our generator only produces code that keeps zero or one items in registers. The generated C code covers both cases; the selection between these alternatives is made at C-compile time. `TOS` in the C code for `+` is just a simple variable name in the one-item case, otherwise it is a macro that expands into `sp[0]`. Note that the GNU C compiler tries to keep simple variables like `TOS` in registers, and it usually succeeds, if there are enough registers[5]. With the TOS optimization turned on, the compiler translates the code for `+` (indirect threaded) into:

```
lw   $2,4($22)    ;n1 = sp[1]
lw   $9,0($21)    ;cfa = *ip
addu $22,$22,4    ;sp += 1;
lw   $8,0($9)     ;ca= *cfa
addu $21,$21,4    ;ip++
j    $8           ;goto *ca
addu $23,$2,$23   ;TOS += n1
```

The primitive generator performs the TOS optimization for the floating-point stack, too. For floating-point operations the benefit of this optimization is even larger: floating-point operations take quite long on most processors, but can be performed in parallel with other operations as long as their results are not used. If the FP-TOS is kept in a register, this works. If it is kept on the stack, i.e., in memory, the store into memory has to wait for the result of the floating-point operation, lengthening the execution time of the primitive considerably.

The TOS optimization makes the automatic generation of primitives a bit more complicated. Just replacing all occurrences of `sp[0]` by `TOS` is not sufficient. There are some special cases to consider:

- In the case of `dup ( w -- w w )` the generator must not eliminate the store to the original location of the item on the stack (see Section 4.1), if the TOS optimization is turned on.
- Primitives with stack effects of the form `--` $\mathtt{w}_1 ... \mathtt{w}_y$ must store the TOS to the stack at the start. Likewise, primitives with the stack effect $\mathtt{w}_1 ... \mathtt{w}_x$ `--` must load the TOS from the stack at the end. But for the null stack effect `--` no stores or loads should be generated.

5 System Architecture

Our Forth system consists not only of primitives, but also of definitions written in Forth. Since the Forth compiler itself belongs to those definitions, it is not possible to start the system with the primitives and the Forth source alone. Therefore we provide the Forth code as an image file in nearly executable form. At the start of the system a C routine loads the image file into memory, sets up the memory (stacks etc.) according to information in the image file, and starts executing Forth code.

The image file format is a compromise between the goals of making it easy to generate image files and making them portable. The easiest way to generate an image file is to just generate a memory dump. However, this kind of image file cannot be used on a different machine, or on the next version of the engine on the same machine, it even might not work with the same engine compiled by a different version of the C compiler. We would like to

[5] Unfortunately the compiler thinks that there are not enough registers available on the R3000; if the TOS optimization is used, it keeps `TOS` in a register, but spills the return stack pointer, which cancels out much of the benefit of the optimization. Hopefully the register allocation of the compiler will improve.

have as few versions of the image file as possible, because we do not want to distribute many versions of the same image file, and to make it easy for the users to use their image files on many machines. We currently need to create a different image file for machines with different cell sizes and different byte order (little- or big-endian)[6].

Forth code that is going to end up in a portable image file has to comply to some restrictions: addresses have to be stored in memory with special words (`A!`, `A,`, etc.) in order to make the code relocatable. Cells, floats, etc., have to be stored at the natural alignment boundaries[7], in order to avoid alignment faults on machines with stricter alignment. The image file is produced by a metacompiler [Rod89].

So, unlike the image file of Mitch Bradleys version of F83 [Bra87], our image file is not directly executable, but has to undergo some manipulations during loading. Address relocation is performed at image load-time, not at run-time. The loader also has to replace tokens standing for primitive calls with the appropriate code-field addresses (or code addresses in the case of direct threading).

6 Conclusion

The system is up and running on at least three machines and two operating systems, but not yet ready for release. It has more than 200 primitives. The object file of the engine needs less than 16Kb on a DecStation, the executable with all linked-in libraries needs about 69Kb, the image file needs 22Kb; 64 iterations of the Sieve on the indirect threaded version with TOS optimization take 7.9s on the DecStation 125 (25MHz R3000) and 3.9s on the HP 720 (50MHz HP-PA). A later version of this paper will be part of the system's documentation.

Acknowledgements

Bernd Paysan did part of the work that is described in this paper; in particular, he is the author of the image file format and the loader. Lennert Benschop contributed, too. Andi Krall commented on an earlier version of this paper. Finally, I would like to thank those who built the Internet and its tools; this international effort would not be possible without it.

[6] We consider adding information to the image file that enables the loader to change the byte order.

[7] E.g., store floats (8 bytes) at an address dividable by 8. This happens automatically in our system when you use the ANSI alignment words.

References

[Bra87] Mitch Bradley. *68000 Unix Forth-83*, 1987. Available via ftp, newer versions commercial.

[Pat90] Mikael Patel. *TILE Release 2.1*, 1990. Available via ftp from any GNU archive site.

[Pay93] Bernd Paysan. ANS fig/GNU/??? Forth. In *Forth-Tagung*, 1993.

[Rod89] Brad Rodriguez. Moving Forth: Principles of metacompilation. Technical report, T-Recursive Technology, 55 McCaul St. #14, Toronto, Ontario M5T 2W7 Canada, 1989.

[Tin81] C. H. Ting. *Systems Guide to fig-Forth*. Offete Enterprises, Inc., San Mateo, CA 94402, 1981.