C Without C

Rob Chapman

1 Abstract

My life began as a programmer the day I started taking heads off of my toys. It was then that I had taken an interest in the parts of the whole of things. A milestone of that young eagerness for disassembly, was my first reassembly (with a few extra parts and some lost). The next milestone was the assembly of something new, a whole of parts from which the parts were not the whole. This youthful exuberance continues on in my current life, as I read, learn, create, dissect using a toolkit which is both the whole as well as the parts.

Last year, at this conference, I did a show-and-tell about a Translator engine which I had been working on.(see Translator Frameworks and Stack Verification published in the 1991 Rochester proceedings). Since then I have done little work on the engine and focused more on creating simple Translator frameworks by solving simple translation problems.

In this paper, I will focus on a framework for translating Forth into C. This includes rules for: interpreting, compiling, commenting, naming restrictions, code-data space separation and the VFM (virtual Forth machine). This paper is a snapshot of my work so far in producing a set of rules for translating Forth to C. Enjoy.

2 Feed-ins

There are several reasons for undertaking this project:

1. Do it once.
   - Once Forth can be translated freely to C, it can take advantage of its benefits, like portability (my last metacompile? (I doubt it.)), or optimization.
2. Have a portable toolkit.
   - My personal set of software tools are not available in all the environments that I use.
   - in the 4 years of using a particular set of software tools, I have had to spend a certain amount of time to porting them to different machines (68K, RTX, 8051, 6811, Unix).
3. Traditionally, a VFM outputs machine code when compiling.
   - It seemed to be an interesting challenge to abstract out the VFM at the source code level in another language.
4. Extending other computer languages.
   - The VFM is infinitely extendable. Can this feature be captured in other languages which have more restrictions.
5. The world wants C programmers.
   - So, I'll build one.

3 Concept

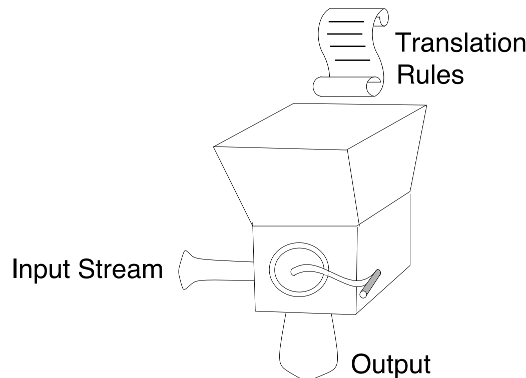

FIGURE 1. Translation engine for translating input to output governed by a set of rules. This may be thought of as a basic language gate with two inputs (Input Stream and Translation Rules) and one output.

By defining some rules, Forth may be translated to C:

```
( botForth Kernel : Mar 10, 1991  Rob Chapman )
( Assume virtual Forth machine exists )
HEX
( ==== Cell ==== )
: CELL  ( -- n' )  cell ;
: CELLS  ( n -- n' )  CELL * ;

( ==== Stacks ==== )
( ==== Data stack primitives ==== )
: SWAP  ( m \ n -- n \ m )  swap ;
: DUP  ( m -- m \ m )  dup ;
: DROP  ( m -- )  drop ;

: ?DUP  ( n -- [n] \ n )  DUP  IF  DUP  ENDIF ;
```

```
/* botForth Kernel : Mar 10, 1991  Rob Chapman */

#include "kernel.h"

/* Virtual Forth Machinery */
unsigned int
*sp,data_stack[64],*rp,return_stack[64]; /* Stacks */
unsigned int index;        /* loops */
void ***ip,**wp;    /* Threader */

/* ==== Cell ==== */

CELL()  /* -- n' */
{
        DUP();
        *sp=sizeof(unsigned int);
}

CELLS()  /* n -- n' */
{
        *sp*=sizeof(unsigned int);     /* CELL* */
}

/* ==== Stacks ==== */

/* ==== Data stack primitives ==== */

SWAP()  /* m \ n -- n \ m */
{
        *--rp=*sp;
        *sp=*(sp-1);
        *(sp-1)=*rp++;
}

DUP()  /* m -- m \ m */
{
        sp--,*sp=*(sp-1);
}

DROP()  /* m -- */
{
        sp++;
}
QUESTION_DUP()  /* n -- [n] \ n */
{
        DUP();
        if(*sp++)   /* IF */
        {
                DUP();
        }
}
```

FIGURE 2. An example of some Forth code which is translated to C code. The code on the right was produced by the translator, not by a human. Note the translation of the comment about a VFM into actual C code.

4 Overview

My software toolkit, which contains the botForth kernel, is the target code to translate. The translation is managed with a make file on Unix which does the necessary compiles when the rules or the kernel change. The input file is kernel.f and the rules files are c, headers and cnames. Two C files are produced: kernel.h and kernel.c. All the headers are kept in kernel.h. The VFM and all the source code for the bodies is in kernel.c. The kernel.c file may be compiled and run with an off the shelf C compiler.

5 From Forth to C

Currently, there are some 500 rules in 7 rule sets. Each rule set takes care of a different part of the translation. For instance, most of the time, source code is compiled, so there is a rule set dedicated to translating compilable Forth source code to compilable C source code. Some of the rule sets are discussed below.

5.1 Compiling

All words become procedure calls. Most word-calls are directly translated to C procedure calls. The control structures are translated into C control structures. For example:

```
: CMOVE  ( src \ dest \ count -- )
   FOR  >R  C@+ SWAP  R> C!+  NEXT  2DROP ;
```

becomes:

```
CMOVE()  /* src \ dest \ count -- */
{
        for(*--rp=index,index=*sp++;index;index--)      /* FOR */
        {
                TO_R();
                C_FETCH_PLUS();
                SWAP();
                R_FROM();
                C_STORE_PLUS();
        }         /* NEXT */
        index=*rp++;
        TWO_DROP();
}
```

All code is laid out as 1 operation to a line. This translates a horizontal style of code into a vertical style of coding. On the toolkit translation, 6 pages of Forth produce about 50 pages of C.

5.2 Comments

Stack comments are kept and appear right after a procedure is declared. Any other comments are included in the same line as the last word which was compiled. If a comment appears by itself on a line, it is assumed to be a heading for the next section of code and it is preceded by a blank line.

5.3 Naming Restrictions

In C, you can only use alphanumerics and the underbar for names and the first character must not be a number. Forth places no naming restriction on name composition except that it is hard to have a blank in a name. In translating Forth names to legal C names, the nonalphanumeric characters are replaced with their pronunciation. Sometimes, this is context dependant. For example:

```
>R, R> and U>
```

become:

```
TO_R, R_FROM and U_GREATER_THAN
```

5.4 Interpreting

C doesn't allow code to be executed while it is compiling so this mode of Forth must be translated into something acceptable to a C compiler. In the case of building up data structures, this is translated into a C struct{}. For example:

```
CREATE prompt 20 ALLOT  ( contains count prefixed string for prompt )
```

becomes:

```
struct
{
        char name12[32];
        /* contains count prefixed string for prompt */
}name13;

prompt()
{
        *--sp=(unsigned int)&name13;
}
```

Data structures are name-numbered and then pushed onto the stack by a procedure which is given the Forth name of the data structure. When building up data structures, all the pieces are accumulated in memory and then flushed out by the next Forth definition. This allows the translator to be one pass but still pick up all the pieces from `ALLOT`s, `,`(comma) and `C,`.

5.5 Code-Data Space Separation

In Forth, there is no such thing as code-space and data-space. Headers and bodies are usually contiguous in memory. In C, this is not allowed. This doesn't really create much of a problem since all the bodies of the Forth words will be translated into code and all the headers will be translated into data structures. By assuming an indirect threaded model, the inner interpreter pointer (code-fields in figForth) simply points to the body:

```
struct{void *link; unsigned char name[5]; void (*tick)();}\
_SWAP={&_CELLS,0x80|4,'S','W','A','P',SWAP};
```

The name of the procedure without the () leaves the address of that procedure in the data structure.

When words are ticked, the address of the inner interpreter field is pushed onto the stack. For instance:

```
: LITERAL  ( n -- [n] )  compile @
   IF  ' LIT COMPILE  ,  ENDIF ; IMMEDIATE
```

becomes:

```
LITERAL()  /* n -- [n] */
{
        compile();
        FETCH();
        if(*sp++)   /* IF */
        {
                *--sp=(unsigned int)LIT;     /* ' */
                COMPILE();
                COMMA();
        }
}
```

5.6 The VFM

By assuming an indirect threaded model, we can freely mix the C code with code produced by the VFM compiler. The kernel has the ability to compile code (simply ,) so that it may be extended. ITC gets compiled into the data space and is interpreted by one of the four inner interpreters: `INNER-VARIABLE, INNER-CONSTANT, INNER-: or INNER-DOES`.

```
: CREATE  ( -- )  HEADER  ' INNER-VARIABLE , ;
: VARIABLE  ( n -- )  CREATE , ;
```

6 Status

The output files kernel.c and kernel.h pass the compile test. This means that I am producing compilable code and there are no name translation problems. Small portions of the code have been actually run to test out parts of the VFM but the C version of the kernel has not yet been run.

7 Future

Possibilities for the future include:

- intermixed Forth and C code. Where we used to allow in-line assembler, we should now allow in-line C code:

```
: FOO  ( -- )  ( Forth code ) {  /* C code */  } ( Forth code ) ;
```

8 Summary

The goal is to write C code without having to write or think in C. I've had enough experience at writing C code to claim to be a novice expert but I find it much easier to think and solve problems in Forth. By creating a set of rules to translate Forth to C, I am liberated to program in Forth yet be able to produce C. My boss will be happy because I am producing C code (pretty, at that!) and I'm happy because I'm programming in Forth. The virtual machine assembler is now C.