Chapter 4

Formal FORTH

In this chapter, we present a system that will aid in ascertaining whether a program meets its specification. By providing a formal base for the FORTH language, we can formally discover the coherence between the specification and its implementation, thus we have the ability to prove that the program meets its specification.

4.1 Introduction

The conventional computer science approach to programming languages starts by separating syntax from semantics.

The syntax deals with allowable statements or sentence formation and has been investigated using techniques that apply equally well to simplified forms of natural language. These techniques result in a classification of languages into categories such as phrase structured, context sensitive and context free. A powerful body of theory (and application) has built up around the syntax of a language.

The semantics of a language deals with the meaning of program text; the interpretation that is placed on a syntactically correct phrase in a given language.

Most language definitions have a formal description of the grammar that describes syntactically correct statements for the given language, however, the syntax of a FORTH system is semantically defined. You could say that FORTH is not a computer language, rather a *dictionary of words* where each word has a *definition* which describes the operation it performs in terms of existing definitions or in terms of the native code of the machine on which the system is implemented.

The set of words thus defined perform all operations executed by the system including the scanning of FORTH text to be compiled or interpreted. A word may be defined to ignore or amend following words in the input stream. It is these abilities that make it difficult to apply classical syntax theory to FORTH.

Many compiler developers use a virtual machine similar to the FORTH abstract machine as a universal intermediate code (Cook and Lee 1980; Miller 1987). Thus, one could say that the FORTH abstract machine is the ideal computer model (Kavipurapu and Cragon 1980).

4.2 The FORTH Toolbox

In order to talk formally about a program, we must have a formally described programming language/environment. FORTH provides us with a simple language with a programming environment and

[0] **This is a chapter taken from the Ph.D. thesis "Practical and Theoretical Aspects of Forth Software Development". A full copy of the thesis is available from the British library. Both the this chapter and the thesis are Copyright © Peter Knaggs.**

debugger. Due to the simple nature of FORTH, this can be formalised much more readily than most other languages (Stoddart 1988). The formal description of the FORTH programming environment will provide us with an additional toolbox to use when formally describing an application program.

4.3 The Basic Model

Let us assume that we have a set of all of the known memory locations in a system and that we have a set of all the possible (allowable) names for a FORTH system:

$$[ADDRS,\ NAMES]$$

It is possible to say that the FORTH dictionary is a relation between names and addresses. However, defining a simple relation does not capture the ordered (historical) nature of the dictionary so we make this a sequential relationship:

$$dict : \mathrm{seq}\,(NAMES \times ADDRS)$$

An example entry of this type would be $(6, (\text{“}\texttt{@}\text{”}, 204))$ where the FORTH word “`@`” is the 6^{th} entry in the dictionary and has an address of 204, quite how this is implemented is unimportant. In a token based system, the 6 could be thought of as being the token for the word while a threaded code system may not store the 6 at all but uses the associated address. For our purposes we will use the notion of a token:

$$token : \mathbb{N}$$

4.4 Word Definitions

We now have the sequence *dict* that tells us what words are in the dictionary and where they can be found. We have yet to record the definition of a given word, to do this we introduce a function that relates known words to their definitions:

$$body : token \rightarrow \mathrm{seq}\ NAME$$

where *token* is the index number of the word in the dictionary and seq *NAME* is the sequence of words that make up the definition. Hence, a word such as `NIP` may have a dictionary entry of[1]:

$$\{33, (\text{“}\texttt{NIP}\text{”}, 378)\}$$

and a definition of:

$$\{33 \mapsto \langle \texttt{SWAP}, \texttt{DROP} \rangle\}$$

4.5 Immediate Words

We must be able to discover if a word is immediate or not. Hence, we introduce a function taking the *token* of a word and returning a true if the word is immediate or a false if not:

$$immediate : token \rightarrow \{\mathrm{true}, \mathrm{false}\}$$

[1] Note that the addresses and token values given in this chapter are for example only and do not relate to any given system.

4.6 Storage Units

FORTH does not use types in the conventional manner. Instead of types it uses classes of storage unit. There are three classes of storage unit: *Character*, *Cell* and *Double Cell*.

Each class of storage unit is able to store any number of types that the application program requires. The only limitations being hardware restrictions. The application programmer may add to the list of possible types that a given storage class can hold, indeed he can even add new storage classes.

Words are defined with reference to the unit class rather than the exact type required. If we were to enforce the use of types in our model we would not be modelling the full behaviour of a FORTH system. Hence this system uses the notion of classes of storage unit.

We must introduce the classes of storage unit as given sets of types:

$$[Char,\ Cell,\ DoubleCell]$$

4.7 Stacks

We must provide a mechanism for the parameter and return stacks. This we do by defining two global variables consisting of a sequence of stack cells:

$$pstack, rstack : \mathrm{seq}\ Cell$$

Thus we could define the FORTH word `DEPTH` as:

$$\texttt{DEPTH} \mathrel{\widehat=} [\, pstack' = \langle \# pstack \rangle \frown pstack \,]$$

Ie, we push onto the stack (add to the start of the sequence) the size of the stack (sequence) as it was at the start of the statement. It should be noted that $pstack'$ is the standard way of indicating the state of the parameter stack after the operation while $pstack$ refers to the state of the parameter stack prior to the operation.

A possible definition for `DROP` would be:

$$\texttt{DROP} \mathrel{\widehat=} [\, pstack' = tail\ pstack \,]$$

Ie, the stack (sequence) now holds all that was previously on the stack (in the sequence) except for the top most (first) element.

Our system will also have to cater for words with variable stack effects such as the `?DUP` word. We can represent this by placing a side condition on the stack description. Ie, `?DUP` is defined as:

$$\texttt{?DUP} \mathrel{\widehat=} \left[\begin{array}{c} \Big((pstack' = pstack) \land (pstack(1) = 0) \Big) \\ \lor \\ \Big((pstack' = \langle pstack(1) \rangle \frown pstack) \land (pstack(1) \neq 0) \Big) \end{array} \right]$$

So far we have only discussed words that effect the parameter stack ($pstack$). However, the system is sufficiently flexible, we can define words such as `>R` which not only effect the parameter stack but also the return stack ($rstack$). The definition of `>R` would be:

$$\texttt{>R} \mathrel{\widehat=} \left[\begin{array}{c} pstack' = tail\ pstack \land \\ rstack' = \langle pstack(1) \rangle \frown rstack \end{array} \right]$$

while the definition for `R>` would be:

$$\texttt{R>} \mathrel{\widehat=} \left[\begin{array}{c} pstack' = \langle rstack(1) \rangle \frown pstack \land \\ rstack' = tail\ rstack \end{array} \right]$$

4.8 Code Definitions

There are many words that are coded in the native machine language of the host computer, `SWAP` and `DROP` are two such words. In order to cater for such words, we introduce a set of code level words. As these words are defined in the native machine language, we can not give their definitions, however, we can give a formal description of the function that they perform.

Assuming that the words `SWAP` and `DROP` have the following dictionary entries:

$$(3, (\text{“}\texttt{SWAP}\text{”}, 30)) \text{ and } (4, (\text{“}\texttt{DROP}\text{”}, 36))$$

then we could represent there actions as:

$$\left\{3 \mapsto [\, pstack' = \langle pstack(2), pstack(1)\rangle \frown tail\, tail\, pstack \,]\right\}$$

and

$$\left\{4 \mapsto [\, pstack' = tail\, pstack \,]\right\}$$

So we now have a function that relates known “code-level” words to their required action:

$$code : token \rightarrow axiom$$

Thus giving us an additional set of *axioms* to work with when reasoning about the implementation. Thus the dictionary is split into “high-level” (*body*) or “code” (*code*) words.

$$\mathrm{dom}\, body \cap \mathrm{dom}\, code = \varnothing$$

It should be noted that we have not provided an instruction pointer. To do so would be to restrict the number of possible implementation techniques. Although the FORTH abstract machine calls for an instruction pointer, we leave it up to the implementor to introduce and define their own.

A consequence of this is that operations such as `NEXT`, `EXIT` and the run-time action of `:` and `;` are currently not specifiable. They must be provided by the implementor, thus allowing them to model their particular method of implementation.

4.9 Wordlists

We define a set of wordlists so that the dictionary is composed of several wordlists, where the wordlists include all the entries in the dictionary. Yet no single entry occurs in more than one wordlist, ie, the dictionary is partitioned into wordlists:

$$wordlist \,\mathrm{partition}\, dict$$

At any point in time, the dictionary has a search order associated with it. The search order is simply a sequence of wordlists that are to be searched:

$$search\_order : \mathrm{seq}\, wordlist$$

There is also the compilation wordlist:

$$compilation\_wl : wordlist$$

4.10 Defining words

When a new word is created, the system state is updated in three ways:

1. Its name is appended to the current compilation wordlist and thereby to the dictionary.
2. Its definition is appended to the *body* or *code* relations dependent on the type of word being defined.
3. The *immediate* relation is updated to indicate if the word is immediate or not.

Let us look at a few examples to see how this works.

4.10.1 High-Level words

We could define the word `+!` as:

```
: +! ( n addr -- ) DUP @ ROT + SWAP ! ;
```

This would add the name `+!` to the currently defined compilation wordlist:

$$compilation\_wl' = compilation\_wl \cup \{(244, (\text{“}\texttt{+!}\text{”}, 8270))\}$$

We now add the word's definition to the system. As it is a "high-level" definition we do this by adding an entry to the *body* relation:

$$body' = body \cup \{224 \mapsto \langle \texttt{DUP}, \texttt{@}, \texttt{ROT}, \texttt{+}, \texttt{SWAP}, \texttt{!} \rangle\}$$

Finally, we extend the *immediate* function so as to return a false value for this word:

$$immediate' = immediate \cup \{244 \mapsto \text{false}\}$$

4.10.2 Immediate words

The definition for the word `IF` could be:

```
: IF COMPILE ?BRANCH >MARK ; IMMEDIATE
```

This would add the name `IF` to the current compilation wordlist:

$$compilation\_wl' = compilation\_wl \cup \{(300, (\text{“}\texttt{IF}\text{”}, 10030))\}$$

The definition of the word is also added to the *body* relation:

$$body' = body \cup \{300 \mapsto \langle \texttt{COMPILE}, \texttt{?BRANCH}, \texttt{>MARK} \rangle\}$$

While the `IMMEDIATE` places a true mapping into the *immediate* function:

$$immediate' = immediate \cup \{300 \mapsto \text{true}\}$$

4.10.3 Code words

When a "code" level word, such as `ROT`, is defined, we add its name to the current compilation wordlist:

$$compilation\_wl' = compilation\_wl \cup \{(5, (\text{“}\texttt{ROT}\text{”}, 40))\}$$

We must now assume that its definition is correct and simply add the description of its function to the set of axiomatic definitions:

$$code' = code \cup \left\{5 \mapsto [\langle pstack(3), pstack(1), pstack(2) \rangle \frown tail\, tail\, tail\, pstack\,]\right\}$$

Finally, the *immediate* function is updated in the same manner as for "high-level" definitions. It is assumed that the word is not immediate unless the `IMMEDIATE` word is placed after its definition.

$$immediate' = immediate \cup \{5 \mapsto \text{false}\}$$

4.11 Dictionary Searching

In order to model the dictionary search, we define a boolean function that returns a true if a given word is in a given wordlist, otherwise it returns a false:

$$inwordlist_1(n, wl) = n \in \mathrm{dom}(\mathrm{ran}(wl))$$

A true result is obtained if n belongs to the set $\mathrm{dom}(\mathrm{ran}(wl))$. Let us clarify this by means of an example:

$$\begin{array}{rl} \text{Assume} & wl = \{(3, (\text{“}\texttt{SWAP}\text{”}, 30)), (4, (\text{“}\texttt{DROP}\text{”}, 36)), (5, (\text{“}\texttt{ROT}\text{”}, 40))\} \\ \text{then} & \mathrm{ran}\, wl = \{(\text{“}\texttt{SWAP}\text{”}, 30), (\text{“}\texttt{DROP}\text{”}, 36), (\text{“}\texttt{ROT}\text{”}, 40)\} \\ \therefore & \mathrm{dom}\,\mathrm{ran}\, wl = \{\text{“}\texttt{SWAP}\text{”}, \text{“}\texttt{DROP}\text{”}, \text{“}\texttt{ROT}\text{”}\} \end{array}$$

We can now define a function to find a given name within a given wordlist:

$$\begin{array}{rrl} find_1(n, wl) = & \mathsf{if} & n = first(second(wl)) \\ & \mathsf{then} & second(second(wl)) \\ & \mathsf{else} & find_1(n, front\, wl) \end{array}$$

This recursive definition says that if the name being searched for (n) in wordlist (wl) is the last name in the wordlist ($first(second(wl))$) then return its associated address ($second(second(wl))$). Otherwise, it repeats the operation on a new wordlist being the *front* of the current wordlist. Note that the definition for $find_1$ does not indicate what will happen if the name is not in the wordlist.

We now introduce a boolean variable to indicate if the word has been found or not:

$$wordfound : \{\mathrm{true},\ \mathrm{false}\}$$

We can now complete our model of the dictionary search operation by defining a function that takes a name and a search order as arguments, returning an address:

$$\begin{array}{rrl} find(n, so) = & \mathsf{if} & so \neq \varnothing \\ & \mathsf{then} & \quad \mathsf{if} \quad inwordlist_1(n, head\, so) \\ & & \mathsf{then} \quad find_1(n, head\, so) \\ & & \qquad\quad wordfound' = \mathrm{true} \\ & & \mathsf{else} \quad find(n, tail\, so) \\ & \mathsf{else} & wordfound' = \mathrm{false} \\ & & 0 \end{array}$$

This function works by checking that the required word (n) can be found in the first wordlist of the search order (so). If it can, we use the function $find_1$ to find it and set the variable *wordfound* to true otherwise we start again using the first wordlist from the remaining wordlists in the search order. Note that if the search order becomes empty, then we have searched through all of the given wordlists without finding the word. Hence, we simply set the variable *wordfound* to false. Thus, we can use the value of *wordfound* to indicate that the word has been found in one of the given wordlists.

4.12 References

Cook, R. and I. Lee (1980, 26–28 May). An extensible stack-oriented architecture for a high-level language machine. In *Proc. of the International Workshop of High-Level Language Computer Architecture*, Fort Lauderdale, FL, pp. 231–237.

Kavipurapu, K. and H. Cragon (1980, 26–28 May). Quest for an 'ideal' machine language. In *Proc. of the International Workshop of High-Level Language Computer Architecture*, Fort Lauderdale, FL, pp. 33–39.

Miller, D. (1987, April). Stack machines and compiler design. *Byte 12*(4), 177–185.

Stoddart, W. J. (1988). Specification & optimisation. In *Proc. EuroFORML Conf.*, Southampton. MicroProcessor Engineering Ltd.