Formal Forth

Peter Knaggs & Bill Stoddart
School of Computing and Mathematics,
Teesside Polytechnic,
Middlesbrough,
Cleveland.
England.

Abstract

Formal notations (such as Z (Zed), and VDM) provide ways of specifying problems, in a clear and precise notation. As these notations are mathematically based we can mathematically prove properties of a given specification.

Here we present a system that will aid in ascertaining weather a program meets it's specification. By providing a formal bases for the Forth language we can formally discover the coherencies between the specification and it's implementation, thus we have the ability to prove that the program meets it's specification.

Introduction

Formal notations (such as Z (Zed) [Spi89], and VDM [Jon86]) provide a way of specifying a problem in a precise mathematical notation. The specification is an abstract mathematical model of the problem, describing what the system has to do, rather than how it is to be done. The notations use set theory, and first order predicate logic to build such models.

There are several reasons for using Formal notations.

Understanding

A formal specification can be passed from one person to another without the possibility of misunderstanding. Due to the ambiguity of natural language we can never be certain if another person fully understands our meaning from a statement. When using Formal notations we can be sure that our exact meaning is presented, as we are using a precise mathematical language [McM89] [Spi89].

Manageability

It has been found that when specifying large systems the formal specification is a great deal smaller than the natural language specification. Additionally the writing of the formal specification can bring out some of the more complex problems that would have remained hidden in the natural language specification [Nas89] [Phi89] [Hay87].

Reasoning

As the specification has been written using a formal notation that is firmly grounded in mathematics, we can mathematically reason about the specification. Indeed it is necessary to prove that the specification is complete, and is not contradictory. We are thus able to show that a specification meets its requirements [Dil90] [Jon86].

Requirement

There are several companies that now insist on the use of formal methods on safety critical systems. The British Ministry of Defence require formal methods to be used on high level safety critical systems. The Lloyds insurance house now insist on a formal proof of all new safety critical systems before they will issue an insurance.

Why Forth

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 [Equation: #s[dictionary of words]] . Each word has a [Equation: #s[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 performs all operation 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 makes it difficult to apply classical syntax theory to Forth.

The theories of programming language semantics, are fully applicable to Forth and there is reason to suppose that the semantic description of Forth will be simpler and more powerful than that of conventional languages.

Formal Notations

In order to present our system concisely we must use the language of Formal notations. In this section we give a brief introduction to this meta language.

Sets

A set is a collection of elements. We can specify the elements of a set by writing them between braces. For example:

X = {1, 2, 3}

The order in which elements of a set are listed does not matter and nor do duplicated entries. Thus:

{1, 2, 3} = {2, 3, 1} = {1, 2, 1, 3}

We use the notation *#X* to denote the number of elements in the set *X*. Thus for the above examples *#X* = 3.

Given a set *X* which contains an element *x* , we use *x:X* to denote that *x* belongs to the set *X*. This can be though of as saying that the variable *x* is of the type *X*. We use the notation *x* ❺ *X* to signify that the *x* is a member of the set *X*. Hence if we define the set *X* as before we can say that 1 ❺ *X*, 2 ❺ *X* and 3 ❺ *X*.

We use two simple set operations, union (⑨), and intersection (⑧). The union of two sets is the set that contains all elements from those sets:

{1, 2, 4}⑨{1, 3} = {1, 2, 3, 4}

The intersection of two sets is the set of elements that are in both sets. Thus:

{1, 2, 4}⑧{1, 3} = {1}

There is a special set that contains no elements. This is referred to as the empty set and is written:

{}

Ordered Pairs

In addition to sets we use ordered pairs. Given sets *A* and *B* , let p = (a, b) be an ordered pair such that *a:A*] and *b:B*. We use the notation *p*(*i*) to select the ith element of the pair *p*. Thus:

p(1) = *a* and *p*(2) = *b*

We use *A* ⑨ *B* to denote the set of all possible ordered pairs from the sets *A* and *B*. Thus the pair *p* would be a member of *A* ⑨ *B* and we can write: *p* : *A* ⑨ *B*

Relations

Let *P* be a set of ordered pairs (*p* : *A* ⑨ *B*). We say P] is a relation between *A* and *B*. We call *A* the source of the relation and *B* the target.

In the examples that follow NAMES and ADDR will be used to represent the set of valid Forth names, and the set of address.

An example of a relation is the set of names and address pairs of words in a Forth dictionary. For a particular Forth system part of this relation might take the form:

R = {("@",204), ("!",226),...,("IF",6045),...,("IF",20464),...}

The source of this relation is NAMES (the set of names), and its target is ADDR (the set of addresses).

Given a relation we say that it's domain is the set of all the elements from the source set that form part of the relation. The range of a relation is the set of elements from the target set that are used in the relation.

For any relation *R* we denote its domain by dom *R*, and its range by ran *R*.

Functions

Partial Functions

A partial function is a relation which has at most one element of the range associated with each element of its domain. Note that not every member of the source need be in the domain. An example of a partial function would be the set of name address pairs which associates with each name the address that would be returned by a dictionary search on the name. If we call this function ***findit*** we use the following notation to indicate that ***findit*** is a partial function from NAMES to ADDR.

findit : NAMES +> ADDR

We call this the "signature" of the function. The notation

NAMES +> ADDR

is defined to mean the set of all possible partial functions from NAMES to ADDR.

Total Functions

A total function is a partial function for which every member of the source is associated with a member of the range. The function #f[findit]] is not total because there are some members of NAMES which are not associated with any address. An example of a total function would be a memory function, which associated each memory address with its contents. We write the signature of this function as:

$$mem : ADDR \rightarrow BYTE$$

Application of Functions to arguments

The function *f* applied to argument *a* is written as *f a*.

For *f a* to be meaningful, a must be a member of dom f. *f a* is the corresponding member of ran *f*. We sometimes enclose the argument a] within brackets, thus written *f*(*a*).

We must use brackets where the argument is an ordered pair. For example *f* (*a*, *b*) denotes the function *f* applied to (*a*, *b*).

If *f* and *g* are both functions, *g* (*f* (*a*)) can be written as *g* (*f* a), or as *g f a*. That is to say function are applied from right to left.

Sequences

A finite sequence of size n] is a function whose domain is the set:

$$\{1, 2, ..., n\}$$

For example, consider the set:

$$S = \{(1,a), (2,b), (3,c), (4,b)\}$$

for this set we have:

$$\text{dom } S = \{1, 2, 3, 4\}$$

$$\text{ran } S = \{a, b, c, d\}$$

We introduce a special notation for writing sequences in which the position of each element denotes the element of the domain being mapped. With this notation we write:

$$S = < a, b, c, b >$$

A sequence is also a function. If S] is a sequence of elements from the set *A* we could write its signature as:

$$S : \{1, 2, ..., \#S\} \rightarrow A$$

We introduce the notation *S* : seq*A* as a more readable way of writing this signature.

It is possible to concatenate one sequence onto the end of another. Let us take the sequence *A* = <1, 2, 3> and *B* = <7, 8, 9> we can concatenate the two sequences together to form a third. The new sequence will have all the elements of the first sequence followed by all the elements of the second.

$$C = A\hat{\ }B = <1, 2, 3, 7, 8, 9>$$

Sequence concatenation will only work on two sequences with the same signature. The resulting signature for *C* is the same as those for *A* and *B*.

There are four functions that allow us to manipulate the content of a finite sequence. Let us assume the sequence *S* = <1, 2, 3, 4>.

$$\text{head } S = 1$$

$$\text{tale } S = <2, 3, 4>$$

$$\text{front } S = <1, 2, 3>$$

$$\text{last } S = 4$$

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 come programming environment come debugger. Due to the simple nature of Forth this can be formalised much more readily than most other languages [Sto88]. The Formal description of the Forth programming environment will provide us with an additional Toolbox to use when formally describing an application program.

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:

$$dict : \text{seq } NAMES \times ADDRS$$

An example entry of this type wold be (6,("@",204)) where the word @ is the 6th entry in the dictionary, and has an address of 204. Quite how this is implementated is up to the individual. In a token based system, the 6 could be though of as being the token for the word @, while a threaded code (or threaded subroutine) system may not store the 6 at all, but use the associated address.

However, for our purposes we will use the notion of a token:

$$token : N$$

Word Definitions

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

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

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:

$$\{33,(\text{"}NIP\text{"}, 378)\}$$

and a definition of:

$$\{33 \rightarrow <SWAP, DROP>\}$$

This allows us to reason about words that are defined with other words.

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 it is not:

$$IMMEDIATE : token \rightarrow \{true, false\}$$

Storages 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 in the basic Forth system. Such as: *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 limitation being a hardware restriction. The application program may also add to the list of possible types that a given storage class can hold.

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 modeling a valid Forth system. Hence our system uses the notion of classes of storage unit.

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

$$[CHAR, CELL, DOUBLE]$$

The Parameter stack

We must provide a mechanism for the parameter (and return) stack. This we do by defining a global variable of a sequence of stack cells:

$$pstack, rstack : \text{seq } cell$$

Thus we could define the Forth word **DEPTH** as pstack' = *<#pstack>^pstack*. Ie., we push onto the stack (add to the start of the sequence) the size of the stack (sequence) as it was on entry to the statement.

A possible definition for **DROP** would be *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.

Code Definitions

There are many words that are coded in the native machine language of the host computer. The SWAP and DROP are two 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,("*SWAP*", 30)},{4,("*DROP*", 36)} }

then we could represent there actions as:

$$\{ \{3 \rightarrow [pstack' = \langle pstack\ 2\rangle \frown \langle pstack\ 1\rangle \frown \text{tail tail } pstack\ ]\}$$

$$\{4 \rightarrow [pstack' = \text{tail } pstack]\} \}$$

So we now have a function that relates known "code level" words to there required action:

$$code : token \rightarrow axiom$$

Thus giving us an additional set of axioms to work with when reasoning about the implementation.

No word may be in both the *body* and the *code* relations:

$$\text{dom } body \cap \text{dom } code = \{\}$$

Wordlists

We define a set of wordlists:

$$WORDLIST : \{wl_1, wl_2,..., wl_n\}$$

The dictionary is composed of several wordlists such that the wordlists include all the entires in the dictionary:

$$dict = wl_1 \cup wl_2 \cup ... \cup wl_n$$

Yet no single entry occurs in more than one wordlist, ie., for any two wordlists wl_i, and wl_j:

$$wl_i \cap wl_j = \{\}$$

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:

$$search\_order : \text{seq } WORDLIST$$

There is also the compilation wordlist:

$$compilation\_wl : WORDLIST$$

Defining words

When a word is created it is appended onto the end of the current compilation wordlist.

Let us take the example of the word +!, we could define this word as:

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

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

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

However, this has only added the name of the word to the system. We must now add the word's definition to the system. This we do by adding an entry to the body relation, thus:

$$\text{body}' = \text{body} \cup \{244 \rightarrow \langle \text{DUP}, @, \text{ROT}, +, \text{SWAP}, ! \rangle\}$$

We must now extend the *IMMEDIATE* function so as to return a *false* value for this word (*token*):

$$IMMEDIATE' = IMMEDIATE \cup \{244 \rightarrow false\ \}$$

The definition for the word IF could be:

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

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

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

The definition of the word is added to the #f[body]] relation:

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

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

$$IMMEDIATE' = IMMEDIATE \cup \{300 \rightarrow true\}$$

A CODE definition is added to the set of axiomatic definitions. Hence the word DROP would be entered as:

$$compilation\_wl' = compilation\_wl \cup \{(4, (\text{"}DROP\text{"}, 36))\}$$

$$code' = code \cup \{4 \rightarrow pstack' = \text{tail } pstack\}$$

$$IMMEDATE' = IMMEDATE \cup \{4 \rightarrow false\}$$

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*:

$$\text{inwordlist}_1\ (n, wl) = n \in \text{dom ran } wl$$

That is, a *true* result is obtained if *n* belongs to the set dom ran *wl*. Let us clarify this by means of an example:

Assume $wl = \{(343,(\text{"}MENU\text{"},40761)), (347,(\text{"}HELP\text{"},41633))\}$

then ran $wl = \{(\text{"}MENU\text{"},40761), (\text{"}HELP\text{"},41633)\}$

$\therefore$ dom ran $wl = \{\text{"}MENU\text{"}, \text{"}HELP\text{"}\}$

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

```
find$_1$ $(n, wl)$ = IF $n$ = ((last $wl$) (2)) (1)
                     THEN ((last $wl$) (2)) (2)
                     ELSE find$_1$ ($n$,front $wl$)
```

This recursive definition says that if the name being searched for (*n*) in wordlist (*wl*) is the last name in the wordlist (((last *wl*) (2)) (1)), then return is associated address (((last *wl*) (2)) (2)). Otherwise repeat 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.

Let us show how the two accessing functions work in this definition. Let us assume that the last entry of the wordlist is (200, ("*BLOCK*", 40562)). Then:

$((200, (\text{"}BLOCK\text{"},40562))(2))(1) = (\text{"}BLOCK\text{"},40562)(1)$

$= \text{"}BLOCK\text{"}]$

$((200, (\text{"}BLOCK\text{"},40562))(2))(2) = (\text{"}BLOCK\text{"},40562)(2)$

$= 40562$

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

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

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, and returns an address:

```
find($n$, $s$) = IF s $\neq$ { }
    THEN IF $inwordlist_1(n, \text{head } s)$
        ELSE $wordfound' = true$
            find$_1$($n$, head $s$)
        ELSE find($n$, tail $s$)
    ELSE $wordfound'$ = $flase$
```

This function works by checking that the required word (*n*) can be found in the first wordlist of the search order (*s*). If it can, then 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.

Type clashes

We have managed to formally describe the Forth environment, including the typeless stack. However the class of storage unit notion introduces an additional problem. Let us look the following sequence of Forth code:

```
X @ EXECUTE
```

Where the variable **X** is holding an integer. The word **@** will fetch a value of storage unit *cell* and place it on the stack. The word **EXECUTE** will then take the *cell* storage unit class and execute the related definition.

There are two types used in this example, *integer*, and *execution token*. Both types belong to the storage unit class cell. In this example we have the word **EXECUTE** expecting a value of type **execution token** when there is a value of type *integer* on the stack. As these are both of storage unit class cell the possible error goes undetected. The type clash would be uncovered when proving that the program meets the specification, provided that a rigourous enough proof is conducted.

It is possible to add a "*Stack Type Algebra*" to the system that would catch such errors [Pöi90] [Sto91].

Conclusion

We have seen how parts of the Forth environment can be specified using Formal Notations. Thus providing us with a programming environment, and an extendible toolset to develop code that can be formally reasoned about. It should be possible to prove that an application program meets with its (formal) specification, and hence with its requirements (this is not to say that the program is optimally coded). Thus we can say that the program is correct with respect to its specification.

Bibliography

[Dil90] Antoni Diller, "Z: An introduction to Formal Methods", John Wiley & Son, 1990.

[Hay87] I. Hayes (Ed.), "Specification Case Studies", Prentice Hall, 1987.

[Jon86] C.B. Jones, "Systematic Software Development Using VDM", Prentice Hall, 1986.

[McM89] M.A. McMorran, J.E. Nicholls, "Z User Manual", Version 1.0, IBM Technical Report TR12.274, July 1989.

[Nas89] Trevor Nash, "Using Z to describe Really large system", Proc. 1989 Z Users Meeting, Oxford University.

[Phi89] M. Phillips, "Results of using Z in CICS", Proc. 1989 Z Users Meeting, Oxford University.

[Pöi90] Jaanus Pöial, "The Algebraic Specification of Stack Effects for Forth Programs", Proc. 1990 EuroFORML Conference.

[Spi89] J.M. Spivy, "The Z Notation: A Reference Manual", Prentice Hall, 1989.

[Sto88] Bill Stoddart, "Specification & Optimisation", Proc. 1988 EuroFORML Conference.

[Sto91] Bill Stoddart, Peter Knaggs, "A Type signature algebra for stack based languages", submitted to "Formal Aspects of Computing", BSC & Springer International.