Chapter 7

A FORTH Type Checker

In this chapter we look at the possibility of implementing a FORTH type checker based on the rules given in the previous chapter. We will be calling this type checking program "FLINT" by analogy with the `lint` program used for checking C programs.

FLINT is to provide a consistency check of FORTH source code. It will examine a file, checking high level FORTH definitions for stack depth and stack types. If an error is detected, the offending word, line, file and an appropriate error (or warning) message will be written to an error log. Checking will continue after the erroneous word.

In this chapter we give an initial specification for the FLINT program.

7.1 Invocation

The FLINT program should be used from the command line. The user will give the command "`flint foo`" to instruct the FLINT system to check the FORTH source code given in the file "`foo.fth`". The system should also check all of the "include" files used.

The user should have the ability to select warnings or errors only. The default being that both warnings and errors are written to the error log (in this instance "`foo.log`"). The user should be able to provide a command line switch to indicate the production the form of report they wish, possible switches are:

`-E` Report errors only to the error log.

`-W` Report warnings only to the error log.

`-C` Check the file, reporting number of errors and warnings to the video. Does not produce an error log but simply counts the errors.

`-V` Produce verbose error/warning reports.

`-S` Produce additional statistical information at the end of the error log.

`-O` Set maximum stack size (for overflow checking).

For such a system to be of any real practical use it must have the ability of being extended by the application code. As such, the system is to provide a basic programming ability that can be "hidden" from the FORTH compiler.

[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.**

7.2 Stack Notation

For this system to operate a formal stack notation system must be provided. A system based on the following rules should be provided:

```
<stack-definition>  ::=  [<stack-items>] --- [<stack-items>]
     <stack-items>  ::=  <stack-item> [<stack-item>]
      <stack-item>  ::=  [<reference>] <type>
       <reference>  ::=  *[<reference>]
```

Thus the user is allowed to place as many type indicators before the `---` part (verbalised as "gives" or "giving") indicating what is currently on the stack. He may also place as many type indicators after the "gives" indicating what is left on the stack after the operation has been performed. The type indicator may consist of any number of references to a given type, where <*type*> is a known type (of a known type class) defined by the system (or by the user) or a wildcard. A reference to a type is the address of the type. Hence if we have a type "`u`" indicating an "unsigned integer" (of class "single-cell") then we indicate the "address of an unsigned integer" with the type indicator "`*u`". The type "`**u`" is the "address of an address of an unsigned integer" (this is the same as the $*^2u$ notation used in section 6.7).

In this discussion we have referred to "user-defined types" and "classes of types". This relates back to the need to allow an application to expand on the types available in the checking mechanism.

7.3 Commands

The commands of the type checking mechanism should be totally "hidden" from the normal FORTH compiler. Hence all control operators must be given in comments:

```
<command>  ::=  \ <function> [ ; <comment>]
             |  ( <function> [ ; <comment>] )
```

Thus provided that the command <*function*> is the first word in a comment the function will be invoked. If FLINT does not recognise the text at the start of a comment to be a command, the comment is ignored and processing continues from the end of the comment. A FORTH system will simply ignore the comments, thus "hiding" the type commands from the FORTH compiler.

The following table lists the possible commands. The requirement for the command and its function is given in more detail in the following sections.

```
<function>  ::=  <type-command>
              |  <stack-command>
              |  <assume-command>
              |  <assert-command>
              |  <syntax-command>
```

7.3.1 Classes

There are to be a number of type classes. The user will not be able to provide any additional type classes[1] this must be coded into the FLINT system. The system will have the following classes:

Single-Cell: The base type for all items that occupy a single cell. There is provision for up to 255 pre- and user-defined types of this class.

[1]This is a limitation of the proposed implementation method. It is hoped that a method of allowing the user to add new type classes may be addressed in the future.

Double-Cell: The class of types that require two cells on the stack. If any attempt is made to access part of the item then a type violation is reported. There is provision for up to 255 pre- and user-defined types of this class.

Two-Cell: A class of types that may be considered a double number at times, but is really made up of two single-cell types. Such types use double number words (such as `2!` and `2@`) when accessing two single-cell items. There is provision for up to 255 pre- and user-defined types of this class.

Reference: A special class for address types. All locations must have a type associated with it. It is possible to make reference to all possible types[2] (no matter their class). Thus the reference type must be able to cater for all possible classes including a reference to a reference to a structure type. So, the reference class is really a super-class encompassing references to all types within it. FLINT should be able to cater for up to 255 levels of referencing (ie $*^{255}k$).

Wildcard: A special class of single-cell wildcards. An item of type wildcard will match with any other single-cell type. This is used in the definition of words such as `SWAP` where the definition is "`w1 w2 --- w2 w1`" (verbalised as "wild-one, wild-two, gives wild-two, wild-one"). `w1` and `w2` are considered to be two separate types of class wildcard. Thus when `w1` is matched to a known type, `w2` may be matched with another. There are 255 possible wildcard types under this class.

Double-Wildcard: A special class of double-cell wildcards. This operates in the same way as single-cell wildcards except that a double-cell wildcard will match with any value of the double-cell class, two-cell class or two values of the single-cell class. There are 255 possible wildcard types under this class.

7.3.2 Type Command

The type command allows the application programmer to add a new type to the list of known types. The type is defined to be of a given type-class. The format of the type command is:

```
<type-command>  ::=  type:  single-cell <type>
                  |  type:  double-cell <type>
                  |  type:  two-cell <type> =
                             <cell-item><cell-item>
                  |  type:  structure <type> = <stack-items>
```

The `type:` command is followed by a one of four type-class identifiers. The name of the new type is then given with the remainder of the command dependent on the class identifier:

`single-cell` The new type will be defined to be of one cell in length. All of the base types are defined to be of the single-cell class.

`double-cell` The new type is defined to be two cells long. Any attempt to gain access to a part of the type will be considered a type violation. For such access an "assume" command will be required to convert the double cell into two single cells.

`two-cell` The new type is defined to be two cells long. The type name is followed by two <*cell-item*>s (`single-cell` type names) that comprise this double cell. Ie, to define a co-ordinate pair that may be broken down into its two (single cell) signed integer parts. Hence when the `two-cell` type is used the system considers the stack image to be of the two `single-cell` types. Thus allowing a convenient short hand notation for such "doubles".

[2] Pass the address of a type, no matter what the type.

`structure` The new type is a compound structure consisting of other stack items[3]. All the types must be defined before the structure is declared. A structure may only be passed by reference. Any attempt to access the structure will result in a type violation. When an item is to be extracted from the structure an "assume" command should be used[4].

7.3.3 Stack Command

The stack command is used to give the expected stack operation of a word. Its format is:

<stack-command> ::= `stack` : *<stack-definition>*

The system will scan the definition of the next word to check that its definition meets the given stack. Flint will assume the stack will be as given on entry and check that the stack is as given on exit. If the expected exit condition is not the same as the "found" condition, an error is reported.

When further words use the word, the "expected" condition will be used and the word will be flagged with a warning.

The given stack image will be placed into a buffer and will be associated with all new words until a new `stack:` command is given, a `;` is found or an "include" is found. Thus the following code fragments are all valid.

```
\ Stack: --- *n | \ Stack: w --- w w | : xxx ( stack: w --- w w )
VARIABLE A      | : xxx              |   ...
VARIABLE B      | ... ;              | ;
```

It may be possible to do without the "`stack:`" part of this command and assume that all commands are "stack" unless their name is found. Thus, the format of the command would now read:

<stack-command> ::= [`stack` :] *<stack-definition>*

However, this would make it more difficult to write good comments.

7.3.4 Assume Command

The assume command will force the system to make an assumption about the current stack type. Its format is:

<assume-command> ::= `assume` : *<stack-items>* `---` *<stack-items>*

Flint will first check that its current stack image is the same as the stack image given in the command, before the "gives". Only the top most elements need be given. It will then replace those elements with the ones given in the stack image after the "gives". An example usage would be to convert an integer into an execution token:

```
\ assume: int --- token
```

This allows the programmer to "cast" form one type to another without supplying code to perform the transformation. The user is allowed to transform any class into any class. The number of *<stack-items>* on either side of the "gives" may vary. Hence:

[3] Including possible other structures and references.

[4] Given that we know the size and position of each element in the structure, it should be possible to validate an access into a structure. This extension has not, as yet, been investigated.

```
\ assume: w ---
```

is a valid assumption that removes a single-cell item from the stack. While:

```
\ assume: --- n
```

is also a valid assumption that introduces a signed integer to the stack.

7.3.5 Assert Command

The assert command instructs the system to check the current stack image against that given in the command. Its format is:

```
<assert-command>  ::=  assert : <stack-items>
                   |   check : <stack-items>
```

If the current stack image does not match that given in the command, an error is reported giving the current stack image. If the stack holds more items than is given in the assert command, only those items given will be checked.

It may be better to insist that the given stack must match all of the current stack items, thus forcing the programmer to identify the complete stack and not only the top-most elements they are interested in. This has the advantage that the programmer will realise how many items are on the stack. This could also be extended to include a special stack item of "...", as the first element to indicate an unknown number of elements before the stack description, thus providing the "partial" check we currently have.

7.3.6 Syntax Command

The syntax command provides the ability for the programmer to define additional syntax structures. Its format is:

```
<syntax-command>  ::=  syntax : <word><syntax-items>
   <syntax-items>  ::=  <syntax-item> [<syntax-item>]
    <syntax-item>  ::=  <<text>> <delimiter>
```

The syntax command is only required when defining a word that defines its own syntax. The following shows some FORTH words and their syntax definitions. Note the space delimiter (␣) on the `CREATE` definition.

```
CREATE   \ syntax: CREATE <word>␣
ABORT"   \ syntax: ABORT" <error message>"
```

This facility is provided so that, when the new <*word*> is used, the system can take account of the fact that the word will scan ahead in the input stream (as far as the indicated <*delimiter*>). The given syntax is associated with the next or currently defined word. Thus the following two code fragments will give the same results:

`\ stack: ---`	`: mess ( stack: --- )`
`\ syntax: mess <id>  <message>#`	`( syntax: mess <id> <message># )`
`: mess ... ;`	`... ;`

It should be possible for FLINT to check that any word defined as having a syntax does have the given syntax. Hence any word that includes a syntax word (ie, uses a word with a syntax) but does not declare a syntax, could generate a warning. If there is a declared syntax and the "found" syntax does not match then an error may be reported.

7.4 Variable Stack Items

7.4.1 Or — `|`

There are words that would benefit from having more than one possible stack image. There are occasions where you would not want to place a wildcard, however, the value may be of two (or more) types. Thus we introduce the "`|`" notation to indicate a possible other valid type for a given stack entry. Hence, to indicate that a word takes a single item from the stack, being an unsigned integer (`u`) or a signed integer (`n`), the definition would be:

```
u | n ---
```

Note, that the `|` is considered to be left associative. Hence:

```
n u | n | f --- n | u f
```

indicates that the word takes two value from the stack, the first being a signed integer (of type `n`) and the second being an unsigned integer (`u`) or a signed integer (`n`) or a boolean flag (`f`). The result of the function is either a singed integer (`n`) or an unsigned integer (`u`), and a flag (`f`).

FLINT will treat such an entry as a special (restricted) form of wildcard. When the wildcard is matched with a known type, it becomes that type for the rest of the definition.

We must redefine <*stack-image*> thus:

<*stack-item*> ::= [<*reference-part*>] <*type*> [`|` <*stack-item*>]

7.4.2 Alternative descriptions — `+`

There are words that have a stack image that do not differ in type but in the number of arguments. One such word is `?DUP`, for this we introduce the "`+`" notation. A stack definition for the word `?DUP` could be:

```
u --- u + u --- u u
```

Indicating that the word takes an unsigned value (`u`) and returns an unsigned value, or it takes an unsigned value and returns two unsigned values.

On words such as these, FLINT will match the relative stack description dependent on the matching types. When such a word is used in a definition, the system will pass though that definition twice, using different versions of the stack definition.

This form of programming is not recommended, thus the use of such words will produce a warning and all words that are defined using it will be flagged with a warning.

Thus we must redefine <*stack-definition*>:

<*stack-definition*> ::= [<*stack-items*>] `---` [<*stack-items*>]
[`+` <*stack-definition*>]

It should be noted that the "`|`" notation can be thought of as a special form of the "`+`" notation. Ie, that stack definition:

```
n --- n | f
```

can be thought of as being the same as the stack definition:

```
n --- n + n --- f
```

7.5 Flow Control

FLINT is able to handle the basic flow control words. There is no mechanism for extending this system to cater for application defined control mechanisms. However this could be added by taking note of where the flow control words are used. This will be made simpler by the adoption of the ANSI standard for the writing of new control words.

The following is a list of flow control words and the action take by FLINT when the word is encountered:

`RECURSE` will compare the current stack image to the entry stack assumption. If there is a type mismatch an error is reported. Note: if additional items are left on the stack, this is **not** considered an error.

`?DUP` is defined as "`u --- false + u --- u true`" thus the code will be checked twice, once for the "`true`" condition, once for the "`false`" condition.

`IF` ... is defined as "`flag ---` " (where "`flag`" is defined as "`true | false`"). It will consume the flag and save the current stack image in a buffer.

...`ELSE` ... will swap the contents of the saved buffer with the current stack image.

...`THEN` compares the current stack image with the one in the saved buffer. Thus, checking that the stack image has not been changed by the `IF` condition. This also checks that an `IF`...`ELSE`...`THEN` statement leaves the stack in the same condition, no matter which execution path is taken. An error is produced if these stack images do not match.

`BEGIN` ... will save the current stack image in a buffer for later processing.

...`AGAIN` compares the current stack image to the buffered one. If they do not match exactly an error is produced. Thus a `BEGIN`...`AGAIN` sequence must have a balanced stack.

...`UNTIL` consumes a flag and then compares the current stack image with the buffered one. If they do not match an error is given.

...`WHILE` ... will consume a flag, then compares the current stack image against the buffered image. An error exists if they do not match.

...`REPEAT` compares the current stack image against the buffered one, reporting an error if they differ. Hence the stack image must be the same at `BEGIN`, `WHILE` (except for the flag) and `REPEAT`. Thus, no matter how many times the loop is executed we know the condition of the stack on exit.

`DO` ... will consume two (signed) numbers, then save the current stack image.

...`LEAVE` ..., definition take from (ANSI 1991), compares the current stack image against the buffered image that will take effect if the leave were to be executed. An error exists if there is a mismatch.

...`LOOP` checks that the current stack image matches with the buffered one. An error exists if not. Note that this enforces "balanced" stacks when using the `DO`...`LOOP` structure.

...`+LOOP` is the same as `LOOP` except it first consumes a signed integer.

Note: If at any comparison too many items are found in the current image a "possible overflow" error is produced. If too few items are found a "possible underflow" error is given. As we are unable to discover how many times a loop is executed these must be errors and not warnings.

FLINT may be able to "syntax check" the high level code, stopping the mismatching of control flow words.

7.6 Defining words

Defining words can be broken into two parts. The "Pre-defined" words and the building blocks to allow "user-defined" defining words.

7.6.1 Pre-defined

`CODE` words cannot be checked. Thus the given stack comment is taken to be correct. If no stack comment is given then an error is recorded. The system could be made to give a warning when the `CODE` word was used indicating that the word could not be fully type-checked.

When a method of type checking assembler code is developed, (possibly similar to the type checking system given in chapter 6) it may be possible to provide type checking of `CODE` level words. However, this currently does not exist (to our knowledge), thus we are unable to type check assembler definitions.

: will define an entry in the data-base, associating the (checked) stack comment with the word. Later (when the word is used) its stack comment is assumed to be correct. If an error occurred when checking the definition, a warning is given for any word that uses it indicating the uncertain status of the check.

`VARIABLE` defines a word with the stack description of " `--- *w`" (giving a reference to a wildcard). Thus allowing any single-cell type to be stored in the variable. If a stack comment is given then it will be used (provided that it matches with the default).

`2VARIABLE` acts in the same way as `VARIABLE` except that the default stack description is " `--- *dw`" (giving a reference to a double wildcard).

7.6.2 User-defined

`CREATE` will mark the current word as a defining word. Words defined using this word will inherit the run time stack signature associated with the word. The system may also check the syntax definition of the defining word.

`DOES>` will check that the current stack signature matches the stack command given for the word. It will then take the following stack command as the run-time stack signature. The system will check that the typing of the run-time part of the word is correct. Note that the address reference placed on the stack by the `DOES>` word *must* be given in the stack comment.

`;CODE` will check that the current stack signature matches with the signature given for the word. An error is reported if the two signatures do not match. As code entries cannot be checked the run-time stack signature is assumed to be correct. When the defined word is used the system can be made to give a warning indicating the use of an unchecked word.

Thus a defining word may be defined:

```
: CCONST \ syntax: CCONST <char> <word>
         \ stack: ---
  BL WORD 1+ C@ CREATE C,
  DOES>  \ stack: *c --- c
         C@
;
```

7.7 Vocabularies

To increase the speed of the system, it will store the checked type signature as part of a word's header definition. As the type signature is being stored along with the word name, the vocabulary structure offered by the host system will automatically be adhered to. Thus, FLINT will need to follow the host systems vocabulary structure. Adoption of the standard vocabulary mechanism (such as the one proposed in the ANSI standard) would be an advantage to this system.

7.8 Error Log

The error log is the output file from the FLINT system. It will consist mainly of error and warning reports.

7.8.1 Error report

The format of the error/warning report will look something like:

```
<level>: <filename> ( <line-no> )
         <word>:-  <message>.
```

where:

<level> indicates the level of the error. That is to say that it is "`Error`" or "`Warning`", where "`Error`" is fatal and must be resolved, "`Warning`" is only informative and is probably caused by a previous error.

<filename> is the name of the file being processed when the error or warning was discovered. This is the name of the current file, thus if the main file includes sub-files, this will be the name of the sub-file.

< line-no > is the line number in the given file upon which the error or warning occurred. More importantly, it is the line at which the error/warning was discovered.

<word> is the word definition that was being checked at the time the error or warning was produced.

<message> is a single line text message given the error or warning condition.

7.8.2 Verbose reports

If the verbose flag (`-V`) is given on the command line the reports will have four parts:

```
<level>:  <filename> (<line-no>)
          <word> :-  <message>.
          <stack>
          <processed-line>
                         <unprocessed-line>
```

where *<level>*, *<filename>*, *<line-no>*, *<word>* and *<message>* are the same as for the normal error log, *<stack>*, *<processed-line>* and *<unprocessed-line>* are given as:

<stack> is the current stack image held by the system at the point of error.

<processed-line> shows the error line. The line up to the point at which the error was discovered is shown on this line.

<*unprocessed-line*> shows what is left on the line. If the report was simply a warning, the text on this line will be processed. However, if the report was an error then the rest of this line and indeed the definition is ignored.

It should also be noticed that each error or warning starts at the first character of the line. All subsequent lines are indented, leaving a space at the start of the line and a blank line between errors. The system is designed in this way so as to ease the automatic searching of the error log.

7.8.3 Statistics

At the end of the error log the system will output the following statistics that it has managed to discover during the type checking:

```
<n>  Error[s]
<n>  Warning[s]
```

Note that the optional "s" is only left off the word when the number <*n*> is 1. This is correct English and should be provided on all such programs.

7.8.4 Statistics flag

When the `-S` (statistics) command line option is given then the following lines of additional statistical information is also given:

```
<n>  line[s],
<n>  file[s],
<n>  assumption[s],
<n>  assert[s],
<n>  definition[s],
<n>  user defined type[s].
```

Although the information given in these statistics is only concerned with the type checking, there is no reason why the system could not be made to give statistical information, and software metrics (Hand 1988), should the user require it.

7.9 Problems

When this system is implemented there are several design decisions that must be made. They are:

1. How to check if a word's definition will cause a stack underflow to occur and how to report the condition.

2. How to check that a definition will not cause a stack overflow to occur when executed. If it will, how is the condition to be reported.

3. How is the system to handle the error reporting around the `PICK` and `ROLL` words. The system could enforce the use of the `assert:` command before the word and the `assume:` command after the word, reporting an error if they are not presence. Alternatively, it can simply mark the word as being unchecked due to the presents of these words.

4. Whether or not to implement the `syntax:` command. They will also need to add checking to compare the declared syntax is correct compared with the word's definition.

5. The capability to type check "in-to" a structure definition. This will require the definition of a standard structure definition operator within the FORTH system.

As FLINT is mainly concerned with the type checking of FORTH programs, these problems are incidental and could be seen as optional extras.

7.10 References

ANSI (1991). *ANS ACS X3/X3J14 Programming languages* — FORTH*: Draft Standard* (first ed.). American National Standards Institute.

Hand, T. (1988). Software metrics for FORTH. In *Proc. Rochester* FORTH *Conf. on Programming Environments*, Rochester, NY, pp. 67–68. Institute of Applied FORTH Research.