Chapter 6

The Cell Type

It is generally considered that the lack of typing in FORTH is useful. This can be seen by the definition of the stack to hold values of type "cell". The definition of the type *cell* is sufficiently vague to allow any data type. However, this can also be misleading and confusing. Here we present a theory that allows us not only to type the arguments of a function, but additionally to check that the arguments are correct for any given function.

6.1 Introduction

The ANS ASC X3/X3J14 Technical Committee defines a cell as:

> The primary unit of information in the architecture of a FORTH system. Data stack elements, return stack elements, addresses, and single-cell numbers are one cell wide. Cell size is implementation-defined, specified in integer address units and the corresponding number of bits. The size of a cell is an integral multiple of the size of a character.

Let us look at the following FORTH code:

```
X @ EXECUTE
```

where the variable `X` is holding an integer. The word `@` will fetch a value of storage class *cell* and place it on the stack. The word `EXECUTE` will then take the *cell* storage 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. This is obviously a type clash. Due to the definition of a cell, we have no choice but to let this error stand. This is not a new problem, it has existed from the first implementations of FORTH.

6.2 Stack Types

One way to solve this problem is to implement some form of typing mechanism. Implementing a run time type checking mechanism would be too cumbersome to be of use. It would also restrict the programmer from performing certain "*tricks*" that require a change of type part way though a definition.

In this chapter, we propose a system that can be used to check the type requirements of a sequence of words at compile time. This has the advantage of not being operational at run time. It also

[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.**

has the advantage of not restricting the programmer from changing the type of a stack argument mid word.

This system can be used to check that any given program meets its stack requirements. This is not the same as saying that the program is complete or correct in operation. That is to say that a program does not invalidate the stack, but may be logically incorrect. This is the same as a Pascal program compiling, but not executing correctly. Such a program is known as having a "logic error" as opposed to a "syntax error" or a "type mismatch".

6.3 Notation

In order to discuss these ideas in a clear manner, we use the notations of set theory and new notations that we have developed for this system.

We give each word a "*type signature*" in the way that we currently give each word a signature (in comments). In order to make things look similar to the current practise, we use the notation $(\ s_1\ \texttt{---}\ s_2\ )$ to indicate a words type signature. In this example, the word is expecting a type sequence s_1 on entry and will leave the type sequence s_2 on exit from the word.

It would be possible to define a word with a type signature of $(\ a, b, c\ \texttt{---}\ a, a\ )$ to indicate that the word will take three arguments of type a, b and c returning two values of type a on the stack. Using this system, it is be possible to prove that the sequence of words that makes up a new word will actually perform the required type transformation.

Let us take another example, this time we will use the word `SWAP`. This has a type signature of $(\ w_1, w_2\ \texttt{---}\ w_2, w_1\ )$. Notice that here we are using the type w_1 to indicate a wildcard type, while the type w_2 indicates another wildcard type. Wildcards are items of unknown type.

We show a sequence of signatures by writing them next to each other. Thus a two word (signature) sequence would be written $(\ s_1\ \texttt{---}\ s_2\ )(\ t_1\ \texttt{---}\ t_2\ )$ where s_1 is the stack image on entry to the sequence and t_2 is the stack image on exit from the sequence.

6.4 Rules

In order to discover if a sequence of type signatures perform the type transformation we require, we use a number of rules for manipulating the signatures. The rules are broken into three logical groups: composition; reduction and wildcard.

6.4.1 Composition Rules

The *Composition Rules* are used to rewrite two signatures into one new signature. We will use the notation $(\ s_1\ \texttt{---}\ s_2\ )(\ t_1\ \texttt{---}\ t_2\ )$ to indicate two adjacent type signatures, where s_1, s_2, t_1 and t_2 are type sequences.

Rule 1: If s_2 is null (there are no types indicated) then we can add the requirements of the second word to that of the first, generating one signature.

$$\frac{(\ s_1\ \texttt{---}\ s_2\ )(\ t_1\ \texttt{---}\ t_2\ ), \#s_2 = 0}{(\ t_1, s_1\ \texttt{---}\ t_2\ )}$$

For example: $(\ a, b\ \texttt{---}\ \ )(\ c\ \texttt{---}\ d\ ) = (\ c, a, b\ \texttt{---}\ d\ )$

Here the first word takes arguments of type a and b off the stack and returns no arguments. The second word takes an argument of type c off the stack and returns a value of type d. Hence the argument c must be on the stack before this sequence is executed.

Rule 2: If t_1 is null (the second word takes no arguments) then we can append the results of the second word to those of the first word.

$$\frac{(\ s_1\ \texttt{---}\ s_2\ )(\ t_1\ \texttt{---}\ t_2\ ), \#t_1 = 0}{(\ s_1\ \texttt{---}\ s_2, t_2\ )}$$

For example: $(\ a\ \texttt{---}\ b\ )(\ \ \texttt{---}\ c\ ) = (\ a\ \texttt{---}\ b, c\ )$

The second word takes no arguments and so the combination of the two sequences can be given by simply adding t_2 onto the end of s_2.

Rule 3: If the last element of s_2 does not match the last element of t_1 then we have a type clash.

$$\frac{(\ s_1\ \texttt{---}\ s_2\ )(\ t_1\ \texttt{---}\ t_2\ ), \mathit{last}\ s_2 \neq \mathit{last}\ t_1}{0}$$

Eg: $(\ a\ \texttt{---}\ a, b\ )(\ a, c\ \texttt{---}\ d\ ) = 0$

Here we have the first word leaving an element of type b on the stack while the second word requires an element of type c. This is a type clash and is written as 0.

6.4.2 Reduction Rules

The following *Reduction Rule* is used to reduce the type signatures until a composition rule can be used on the sequence.

Rule 4: If the last element of s_2 is the same as the last element of t_1 then the types do not clash and the argument passing is internal to the sequence of operations. Hence we can rewrite the sequence removing this element.

$$\frac{(\ s_1\ \texttt{---}\ s_2\ )(\ t_1\ \texttt{---}\ t_2\ ), \mathit{last}\ s_2 = \mathit{last}\ t_1}{(\ s_1\ \texttt{---}\ \mathit{front}\ s_2\ )(\ \mathit{front}\ t_1\ \texttt{---}\ t_2\ )}$$

Eg: $(\ a\ \texttt{---}\ b\ )(\ a, b\ \texttt{---}\ c\ ) = (\ a\ \texttt{---}\ \ )(\ a\ \texttt{---}\ c\ )$

The first word passes an argument of type b to the second word. This is internal to the sequence of operation and so does not need to be shown.

6.4.3 Wildcard Rules

The remaining rules are intended to provide for wildcards, where a wildcard argument is able to match with an argument of any known type. We refer to these as *wildcard rules* even though they are reducing the type signature and thus can be considered as reduction rules. We indicate a known type as being a member of the set $\mathbb{K}$ and a wildcard type as being a member of the set $\mathbb{W}$.

Rule 5: If the last element of s_2 is of a known type and the last element of t_1 is a wildcard we remove the matching items, rename any additional occurrences of the wildcard in the second signature with the known type from the first signature.

$$\frac{(\ s_1\ \texttt{---}\ s_2\ )(\ t_1\ \texttt{---}\ t_2\ ), \mathit{last}\ s_2 \in \mathbb{K}, \mathit{last}\ t_1 \in \mathbb{W}}{(\ s_1\ \texttt{---}\ \mathit{front}\ s_2\ )((\ \mathit{front}\ t_1\ \texttt{---}\ t_2\ )[\mathit{last}\ s_2 / \mathit{last}\ t_1])}$$

$$\begin{array}{rl} \text{Example:} & (\ a\ \texttt{---}\ b, c\ )(\ w_1, w_2\ \texttt{---}\ w_1, w_2, w_1\ ) \\ \Rightarrow & (\ a\ \texttt{---}\ b\ )((\ w_1\ \texttt{---}\ w_1, w_2, w_1\ )[c/w_2]) \\ \Rightarrow & (\ a\ \texttt{---}\ b\ )(\ w_1\ \texttt{---}\ w_1, c, w_1\ ) \end{array}$$

The first word passes the second word an argument of type c which is matched with the wildcard w_2 expected by the second word. Thus we can determine the type of w_2 for the second signature.

Rule 6: If the last element of s_2 is a wildcard and the last element of t_1 is of a known type, we can remove the matching types and replace any occurrences of the wildcards in the first signature by the known type.

$$\frac{(\ s_1\ \text{---}\ s_2\ )(\ t_1\ \text{---}\ t_2\ ),\mathit{last}\, s_2 \in \mathbb{W}, \mathit{last}\, t_1 \in \mathbb{K}}{((\ s_1\ \text{---}\ \mathit{front}\, s_2\ )[\mathit{last}\, t_1 / \mathit{last}\, s_2])(\ \mathit{front}\, t_1\ \text{---}\ t_2\ )}$$

For Example: $(\ w_1, w_2\ \text{---}\ w_2, w_1\ )(\ a, b\ \text{---}\ c\ )$
$\Rightarrow$ $((\ w_1, w_2\ \text{---}\ w_2\ )[b/w_1])(\ a\ \text{---}\ c\ )$
$\Rightarrow$ $(\ b, w_2\ \text{---}\ w_2\ )(\ a\ \text{---}\ c\ )$

We can determine the type of w_1 because it must match the type b given in the second word.

Rule 7: If there are wildcard types in the first signature and similarly named wildcard types in the second signature, we rename the wildcards in the second signature by decorating them with a prime.

$$\frac{(\ s_1\ \text{---}\ s_2\ )(\ t_1\ \text{---}\ t_2\ ),\operatorname{ran}(s_1 \cup s_2) \cap \operatorname{ran}(t_1 \cup t_2) \cap \mathbb{W} \neq \varnothing}{(\ s_1\ \text{---}\ s_2\ )((\ t_1\ \text{---}\ t_2\ )[w'/w])}$$

Eg: $(\ w_1, w_2\ \text{---}\ w_2, w_1\ )(\ w_1, w_2\ \text{---}\ w_2, w_1\ )$
$\Rightarrow$ $(\ w_1, w_2\ \text{---}\ w_2, w_1\ )((\ w_1, w_2\ \text{---}\ w_2, w_1\ )[w'/w])$
$\Rightarrow$ $(\ w_1, w_2\ \text{---}\ w_2, w_1\ )(\ w_1', w_2'\ \text{---}\ w_2', w_1'\ )$

We have renamed all of the wildcards in the second signature to be different to those in the first signature.

Rule 8: If the last element of s_2 is a wildcard and the last element of t_1 is a wildcard, we can remove the matching wildcards, renaming all remaining occurrences of the wildcard in the second signature with the wildcard from the first signature, provided that the wildcard does not already exist in the second signature (there is not a name clash).

$$\frac{(\ s_1\ \text{---}\ s_2\ )(\ t_1\ \text{---}\ t_2\ ),\mathit{last}\, s_2 \in \mathbb{W}, \mathit{last}\, t_1 \in \mathbb{W}, \mathit{last}\, s_2 \notin \operatorname{ran}(t_1 \cup t_2)}{(\ s_1\ \text{---}\ \mathit{front}\, s_2\ )((\ \mathit{front}\, t_1\ \text{---}\ t_2\ )[\mathit{last}\, s_2 / \mathit{last}\, t_1])}$$

Example: $(\ w_1, w_2\ \text{---}\ w_2, w_1\ )(\ w_1', w_2'\ \text{---}\ w_2', w_1'\ )$
$\Rightarrow$ $(\ w_1, w_2\ \text{---}\ w_2\ )((\ w_1'\ \text{---}\ w_2', w_1'\ )[w_1/w_2'])$
$\Rightarrow$ $(\ w_1, w_2\ \text{---}\ w_2\ )(\ w_1'\ \text{---}\ w_1, w_1'\ )$

The wildcard w_1 from the first signature has been matched with the wildcard w_2' from the second signature. The operation of this rule is exactly the same as rule 4 with the exception that it is for wildcard arguments and not for arguments of known types.

6.5 Simple Examples

Here are some examples of how you would compose two or more signatures together using these rules.

1.

$(\ a\ \text{---}\ b, c, d\ )(\ w_1, w_2\ \text{---}\ w_2, w_1\ )$	
$(\ a\ \text{---}\ b, c\ )(\ w_1\ \text{---}\ d, w_1\ )$	Resolve wildcard (5)
$(\ a\ \text{---}\ b\ )(\ \text{---}\ d, c\ )$	Resolve wildcard (5)
$(\ a\ \text{---}\ b, d, c\ )$	Combine (2)

2.

$(\; w_1, w_2, w_3 \; \text{---} \; w_2, w_3, w_1 \;)(\; a, b \; \text{---} \; c \;)$

$(\; b, w_2, w_3 \; \text{---} \; w_2, w_3 \;)(\; a \; \text{---} \; c \;)$ Resolve wildcard (6)

$(\; b, w_2, a \; \text{---} \; w_2 \;)(\; \text{---} \; c \;)$ Resolve wildcard (6)

$(\; b, w_2, a \; \text{---} \; w_2, c \;)$ Combine (2)

Since the naming of wildcards is arbitrary we could simply write the last line of this example as $(\; b, w, a \; \text{---} \; w, c \;)$.

3.

$(\; w_1, w_2 \; \text{---} \; w_1, w_2, w_1 \;)(\; w_1, w_2 \; \text{---} \; w_1, w_2, w_1 \;)$

$(\; w_1, w_2 \; \text{---} \; w_1, w_2, w_1 \;)(\; w_1', w_2' \; \text{---} \; w_1', w_2', w_1' \;)$ Rename (7)

$(\; w_1, w_2 \; \text{---} \; w_1, w_2 \;)(\; w_1' \; \text{---} \; w_1', w_1, w_1' \;)$ Match wildcards (8)

$(\; w_1, w_2 \; \text{---} \; w_1 \;)(\; \text{---} \; w_2, w_1, w_2 \;)$ Match wildcards (8)

$(\; w_1, w_2 \; \text{---} \; w_1, w_2, w_1, w_2 \;)$ Combine (2)

4. Let us assume the following signatures for FORTH words:

`DROP`	$(\; w \; \text{---} \;)$
`OVER`	$(\; w_1, w_2 \; \text{---} \; w_1, w_2, w_1 \;)$
`SWAP`	$(\; w_1, w_2 \; \text{---} \; w_2, w_1 \;)$
`ROT`	$(\; w_1, w_2, w_3 \; \text{---} \; w_2, w_3, w_1 \;)$

We can show that the sequence `OVER ROT DROP` has the same type signature as the word `SWAP`:

$(\; w_1, w_2 \; \text{---} \; w_1, w_2, w_1 \;)(\; w_1, w_2, w_3 \; \text{---} \; w_2, w_3, w_1 \;)(\; w \; \text{---} \;)$

$(\; w_1, w_2 \; \text{---} \; w_1, w_2, w_1 \;)(\; w_1', w_2', w_3' \; \text{---} \; w_2', w_3', w_1' \;)(\; w \; \text{---} \;)$ (7)

$(\; w_1, w_2 \; \text{---} \; w_1, w_2 \;)(\; w_1', w_2' \; \text{---} \; w_2', w_1, w_1' \;)(\; w \; \text{---} \;)$ (8)

$(\; w_1, w_2 \; \text{---} \; w_1 \;)(\; w_1' \; \text{---} \; w_2, w_1, w_1' \;)(\; w \; \text{---} \;)$ (8)

$(\; w_1, w_2 \; \text{---} \;)(\; \text{---} \; w_2, w_1, w_1 \;)(\; w \; \text{---} \;)$ (8)

$(\; w_1, w_2 \; \text{---} \; w_2, w_1, w_1 \;)(\; w \; \text{---} \;)$ (2)

$(\; w_1, w_2 \; \text{---} \; w_2, w_1 \;)(\; \text{---} \;)$ (8)

$(\; w_1, w_2 \; \text{---} \; w_2, w_1 \;)$ (2)

6.6 Multiple Signatures

It is possible for a FORTH word to have more than one acceptable signature. Indeed there are many words in FORTH that require more than one signature. For this reason we have introduced the "+" symbol to indicate the existence of another possible signature for the same word.

Let us take the FORTH word `AND`, there are two functions associated with this word. The first is that of a logical (Boolean) `AND`, while the second is that of a binary (bitwise) `AND`. The signature for a Boolean `AND` is $(\; \mathit{flag}, \mathit{flag} \; \text{---} \; \mathit{flag} \;)$, while the signature for a bitwise `AND` is $(\; \mathit{logical}, \mathit{logical} \; \text{---} \; \mathit{logical} \;)$, thus the true signature is:

$$\text{sig}(\texttt{AND}) = (\; \mathit{flag}, \mathit{flag} \; \text{---} \; \mathit{flag} \;) + (\; \mathit{logical}, \mathit{logical} \; \text{---} \; \mathit{logical} \;)$$

The correct signature will be used in composition due to the naming of a known type. Let us assume that the FORTH word `IF` has the signature $(\; \mathit{flag} \; \text{---} \;)$. When we come to compose the sequence `AND IF` we will know (from the signature of `IF`) that the Boolean `AND` signature is required.

Notice that we have also introduced the notation sig(α) to indicate all of the possible signature compositions of the phrase α.

6.7 Pass by reference

We indicate a pointer to a known type by writing $*^n k$. Where the $*^n$ is used to indicate n levels of indirection and the k is the known type being referenced. For simplicity we write $*k$ to indicate $*^1 k$. The notation $*^0 k$ is the same as the basic type k without indirection.

A possible definition of the FORTH word `@` would be ($*w$ --- w), however we have not defined the pointer type to be able to point to wildcard types. Hence the actual signature for `@` is:

$$\sum_{k \in \mathbb{K}} (\ *k \texttt{ --- } k\ )$$

This produces a collection of signatures, (one for every entry in $\mathbb{K}$). The correct signature will be selected when this word is being composed.

6.8 Control Structures

We use the ideas of multiple signatures (and summation) to show all of the possible paths through a control sequence. This is best shown by example.

Let us take the FORTH statement: `IF` α `ELSE` β `THEN`. We must compose the signature for both cases of the `IF` condition. Hence for a true condition the sequence (*flag* ---)sig(α) exists, while for a false condition the sequence (*flag* ---)sig(β) exists. These two signature can be written as one multiple signature:

$$(\ \mathit{flag} \texttt{ --- }\ )\,(\mathrm{sig}\,(\alpha) + \mathrm{sig}\,(\beta))$$

For a more complex control structure, such as `BEGIN` α `WHILE` β `REPEAT`, we have no way of knowing how many times the loop will be executed. We must therefore produce a multiple type signature for all the possible different number of iterations:

$$\sum_{i=0}^{\infty} (\mathrm{sig}\,(\alpha)\ (\ \mathit{flag} \texttt{ --- }\ )\mathrm{sig}\,(\beta))^i\ \mathrm{sig}\,(\alpha)\ (\ \mathit{flag} \texttt{ --- }\ )$$

However, it is normally the case that a loop of this form is "balanced" in terms of its stack arguments. In the case of a balanced stack, the loop can be simplified to a single term. If the sequence sig(α)(*flag* ---)sig(β) can be reduced to a signature of (s --- s), (ie a balanced signature) we can reduce this signature to:

$$(\ s \texttt{ --- } s\ )\mathrm{sig}(\alpha)(\ \mathit{flag} \texttt{ --- }\ )$$

In order to fully satisfy ourselves that a program is complete, we must follow though every single path of execution. We can say that a word definition (or program) is type correct if its expected input and output types can be reduced to the single signature holding the same input and output types.

For example: Let us define a FORTH word `EXAMPLE` which takes an input signature of s and is expected to produce an output signature of t. The word is type correct if

$$\mathrm{sig}(\texttt{EXAMPLE}) = (\ s \texttt{ --- } t\ )$$

It should be noted that we are currently unable to check words that make use of the `EXECUTE` word. For example, given the definition:

```
: TEST ( char --- ) 'TEST @ EXECUTE ;
```

it would be possible for us to check that the body of `TEST` is correct. However the action of `TEST` is to execute the definition, the execution token of which, is stored in the variable `'TEST`. As we do not know the signature of this definition, we can not check against the specification given, ie (*char* ---).

This problem may be overcome by expanding the rôle of the execution token to include a type signature within a type signature. Ie, the signature (*char*, (*char* ---) ---) indicates a character and an execution token are expected where the execution token has the signature of (*char* ---). There are a number of ways in which one might resolve this restriction, several of which are currently under investigation.

6.9 Casting

There are occasions when a programmer will want to convert the type of a stack item that is not catered for by the default matching type signatures. We have introduced a notation that will allow the programmer to alter the current type signature at compile time.

Let us assume that the programmer would like to convert a single-cell integer into an execution token. He would have to add the following line to his code:

```
<< int --- token >>
```

Where the FORTH word `<<` enters into a "*alter type signature*" mode. He then gives a representation of what he expects the current stack type signature to be (`int`). The word `---` is used to indicate the end of the current stack and the start of a description indicating what he would like the current stack type signature to become (`token`). Finally, the word `>>` replaces the current type signature with the required signature.

Obviously, there are many checks we can make at this point. The number of stack items expected (between `<<` and `---`) must be equal to (or less than) the actual number of item calculated to be on the stack (and of the correct type). The number of stack items given in the expected part must be the same as the number given in the wanted part (between the `---` and `>>`), thus protecting the compilers image of the stack.

6.10 Strong vs Weak Typing

6.10.1 Strong Typing

In a strongly typed system, every variable will have a known type associated with it. Hence a single-cell variable that has been defined to hold an integer could not hold a token as that would lead to a type clash.

A strongly typed system would be difficult to implement compared to a weakly typed system. It would have to keep track of the type associated with each memory cell in the system where as a weakly typed system would not retain this information. Due to the nature of types in FORTH, a strongly typed system would require the programmer to give additional information. See sections 4.6 and 7.3.1 for a discussion on FORTH's type structure.

In a strongly type system, the programmer would have the benefit of peace of mind, insomuch as he knows that the system will report an error if he attempts to develop code that uses the stack in what would be considered the wrong way.

This can be seen by examining the following code:

```
X @ EXECUTE
```

This would have the following type signatures:

(--- **int*)(**int* --- *int*)(*token* ---)

This would obviously clash on the (**int* --- *int*)(*token* ---) section.

In order to compile this code the programmer would have to write:

```
X @ << int --- token >> EXECUTE
```

To convert the *int* returned from the `@` into the *token* that is excepted by `EXECUTE`. Thus the programmer has to explicitly instruct the compiler to make the conversion and allow this code.

6.10.2 Weak Typing

In a weakly typed, system all memory cells will be defined to hold any of the known types. This is simpler to implement, however it does not bring with it the same peace of mind that a strongly typed system would.

If we take the same code as before:

```
X @ EXECUTE
```

which will now have a type signature of:

$$(\ \texttt{---}\ *k\ )(\ *token\ \texttt{---}\ token\ )(\ token\ \texttt{---}\ \ )$$

We can see that in the weakly typed system the `X` returns a referenced known type ($*k$) this will be matched with the referenced token ($*token$) type required by `@`. Thus, this code will be acceptable to a weakly typed system. Hence, a weakly typed system can aid in program construction but will not be able to catch misusage of variables.