The Cell Type

Peter J Knaggs and Bill Stoddart
Teesside Polytechnic
Middlesbrough, Cleveland, England.

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 to be allowed.

However this can also be misleading and confusing. Here we present a theory that allows us to not only type the arguments to a function, but additionally to check that the arguments are correct for any given function.

Introduction

The ANS ASC X3/X3J14 Technical Committee define 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. [Bas91]

It is allowable to store an execution token in a single-cell variable. Hence it is possible to fetch the contents of a single-cell variable and execute the resulting token. However, if the single-cell variable were to be holding an integer value then the system would treat this value as if it where an execution token. This is obviously a type clash.

Stack Types

We propose a system that can be used to check the type requirements of a sequence of words at compile time. This will also have 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 meats with its stack requirements. This is not the same as saying that the program is complete or correct in operation. We only say that the code is type correct [Pöi90] [Sto91].

Notation

We give each word a type signature, we use the notation (s_1 --- s_2) to indicate a words type signature. It would be possible to define a word with a signature of (*a*, *b*, *c* --- *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 our system it is be possible to prove that the sequence of words that makes up the new word will actually perform this type transformation.

Rules

In order that we can discover whether a sequence of type signature will perform the type transformation that we require we use the following rules for manipulating the signatures.

Composition Rules

The Composition Rules are used to rewrite two signatures into one new signature. We will use the notation (s_1 --- s_2) (t_1 --- t_2) to indicate two adjacent type signatures.

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.

$$\frac{(\ s_1 \text{ --- } s_2\ )\ (\ t1 \text{ --- } t_2\ ),\ \#s_2 = 0}{(\ t_1, s_1 \text{ --- } t_2\ )}$$

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.

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

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

$$\frac{(\ s_1 \text{ --- } s_2\ )\ (\ t_1 \text{ --- } t_2\ ),\ \text{last } s_2 <> \text{last } t_1}{0}$$

Reduction Rules

Reduction rules are 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.

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

The remaining rules are used when handling wildcard types. A wildcard type can match any known type. We indicate a known type as being a member of the set κ, and a wildcard type as being a member of the set 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, and rename any additional occurrences of the wildcard in the second signature with the known type from the first signature.

$$\frac{(\ s_1 \text{ --- } s_2\ )\ (\ t_1 \text{ --- } t_2\ ),\ \text{last } s_2\ \varepsilon\ \kappa,\ \text{last } t_1\ \varepsilon\ W}{(\ s_1 \text{ --- front } s_2\ )\ (\ (\ \text{front } t_1 \text{ --- } t_2\ )\ [\ \text{last } s_2\ /\ \text{last } t_1\ ]\ )}$$

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.

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

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

$$\frac{(s_1 \text{ --- } s_2)(t_1 \text{ --- } t_2),\ \text{ran}(s_1\ \text{U}\ s_2)\ \text{inter ran}(t_1\ \text{U}\ t_2)\ \text{inter W} <> \{\}}{(\ s_1 \text{ --- } s_2\ )(\ (\ t_1 \text{ --- } t_2\ )\ [\omega'\ /\ \omega\ ])}$$

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, and rename all remaining occurrences of the wildcard in the second signature with the wildcard from the first signature, providing that the wildcard does not already exist in the second signature (there is not a name clash).

$$\frac{(s_1 \text{ --- } s_2)(t_1 \text{ --- } t_2),\text{last } s_2 \varepsilon W,\text{last } t_1 \varepsilon W,\text{last } s_2 \text{ not}\varepsilon\ \text{ran}(t_1\ \text{U}\ t_2\ )}{(\ s_1 \text{ --- front } s_2\ )\ (\ (\ \text{front } t_1 \text{ --- } t_2\ )\ [\text{last } s_2\ /\ \text{last } t_1\ ]\ )}$$

Simple Example

Let us assume the following signatures for Forth words:

DROP (ω ---)

OVER (ω_1,ω_2 —— $\omega_1,\omega_2,\omega_1$)

SWAP ($\omega_1,\omega_2,\omega_3$ —— $\omega_2,\omega_3,\omega_1$)

Prove that the sequence **OVER ROT DROP** is equivalent to the word **SWAP**:

Assume:(ω_1,ω_2——$\omega_1,\omega_2,\omega_1$)($\omega_1,\omega_2,\omega_3$——$\omega_2,\omega_3,\omega_1$)($\omega$——)

Rule 7: (ω_1,ω_2——$\omega_1,\omega_2,\omega_1$)($\omega'_1,\omega'_2,\omega'_3$——$\omega'_2,\omega'_3,\omega'_1$)($\omega$——)

Rule 8: (ω_1,ω_2——ω_1,ω_2)(ω'_1,ω'_2——$\omega'_2,\omega_1,\omega'_1$)($\omega$——)

Rule 8: (ω_1,ω_2——ω_1)(ω'_1——$\omega_2,\omega_1,\omega'_1$)($\omega$——)

Rule 8: (ω_1,ω_2——)(——$\omega_2,\omega_1,\omega_1$)($\omega$——)

Rule 2: (ω_1,ω_2——$\omega_2,\omega_1,\omega_1$)($\omega$——)

Rule 8: (ω_1,ω_2——ω_2,ω_1)(——)

Rule 2: (ω_1,ω_2——ω_2,ω_1)

Multiple Signatures

There are two functions associated with the Forth word **AND**. 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 (flag, flag --- flag), while the signature for a bitwise **AND** is (logical, logical --- logical). So the signature for the word **AND** is:

sig(AND) = (flag, flag --- flag) + (logical, logical --- 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 (flag ---). 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 (X), to indicate all of the possible signature compositions of the phrase **X.**

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.

Control Structures

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

(flag ---) (sig (A) + sig (B))

For a more complex control structure, such as **BEGIN** A **WHILE** B **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:

$$\sum_{i=0}^{\infty} (\text{sig(A) (flag ---) sig(B)})^i \text{ sig (A) (flag ---)}$$

Stacrobatics

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.

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 signature to be (**int**). The word --- is used to move from describing the current stack, to describing the signature he would like (**token**). Finally the word >> replaces the current stack signature with the required signature. This is similar to C's casting mechanism.

Strong vs Weak 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. While in a weakly typed system all the variables will be defined to hold any of the known types.

This can be seen by examining the following code:

```
X @ EXECUTE
```

In a strongly typed system this would have a signature of:

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

Which shows up the type clash. In order to compile this code the programmer would have to cast the top element of the stack:

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

While is a weekly typed system it would have a signature of:

(--- **k*) (**token --- token*) (*token* ---)

As the **X** returns a referenced to a 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.

Conclusion

The lack of typing in Forth makes it a versatile, if unusual, tool. However, we have noted that where this ability is abused we can, unwittingly, write code that may cause the system to perform different actions to those we expected.

We have seen how a typing system could be added to Forth at compile time. This system will allow us to reason about the code that we are writing. It will stop the abuse of the typeless stack, whilst maintaining the abilities of a typeless stack.

This system would allow us to say that a program is correct in its handling of the stack. It can not say that the program will perform as required, but simply that the stack manipulation will not be the cause of any problem.

Bibliography

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

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

[Bas91] ANS ASC X3/X3J14 Technical Committee, "X3J14 Basis Document", revision 15.