A FRAMEWORK FOR DATA STRUCTURES IN A TYPED FORTH

Federico de Ceballos

Universidad de Cantabria
federico.ceballos@unican.es

September, 2007

Strong Forth as a typed Forth

In Strong Forth the interpreter knows the types associated with the elements in the stack. The compiler takes care that a word's behavior is consistent with its declared specification. This has to be unique.

```
SWAP ( SINGLE SINGLE -- 2ND 1ST )
```

Because every word has a declared input diagram, overloading is possible.

```
SWAP ( DOUBLE DOUBLE -- 2ND 1ST )
```

If you do need complete control, you can use a cast.

```
CHAR 0 .                    \ prints 0
CHAR 0 CAST SINGLE .        \ prints 48
```

Strong Forth's type hierarchy

The language comes with several families of types:

```
SINGLE      INTEGER           UNSIGNED
                              SIGNED
                              CHARACTER
            LOGICAL           FLAG
            TOKEN
            MEMORY-SPACE
            FILE

DOUBLE      INTEGER-DOUBLE    UNSIGNED-DOUBLE
                              SIGNED-DOUBLE
            CONTROL-FLOW
            DATA-TYPE         STACK-DIAGRAM
            DEFINITION        COLON-DEFINITION

FLOAT
```

Strong Forth's type hierarchy (2)

```
SINGLE      ADDRESS          DATA
                             CONST
                             PORT
                             CODE
                             CADDRESS         CDATA
                                              CCONST
                                              CPORT
                                              CCODE
                             SFADDRESS        SFDATA
                                              SFCONST
                                              SFCODE
                             DFADDRESS        DFDATA
                                              DFCONST
                                              DFCODE

DOUBLE      FAR-ADDRESS      CFAR-ADDRESS
                             SFFAR-ADDRESS
                             DFFAR-ADDRESS
```

Forth's advantages

Usability.

Interactive development.

Efficiency (both in terms of development time and generated code).

Possibility of accessing low-level resources.

Little constraints.

Intimate knowledge of your development environment.

Typeless. (A language feature, not a bug.) ;-)

Variations from the Strong Forth model

Strong Forth is aimed to embedded systems with multiple address spaces, we are targeting a PC with a single address space.

Strong Forth tries to keep the Forth *flavour* as much as possible, we plan to take advantage of the new possibilities offered.

The idea of having different base pointers in order to navigate through different data sizes is abandoned.

Data definitions in Forth

```
VARIABLE FIRST

2VARIABLE SECOND

FVARIABLE THIRD

SFVARIABLE FOURTH

DFVARIABLE FIFTH

CREATE SIXTH 3 CELLS ALLOT

23 VALUE THIS


CLASS POINT
   VARIABLE X
   VARIABLE Y
END-CLASS

POINT BUILDS MY-POINT
```

Data definitions in Forth

```
23 FIRST !

SECOND 2@

1.23E THIRD F!

1.23E FOURTH SF!

1.23E FIFTH DF!

1 2 3  SIXTH  TUCK !  CELL+ TUCK !  CELL+ !


45 TO THIS


12 MY-POINT X !
14 MY-POINT Y !

14 12 MY-POINT  TUCK X !  Y !        \ An error !!!
```

Basic data types

The Strong Forth system provides the following basic data types from which the rest are derived:

SINGLE	**A single-cell data type, used to hold a number or an address.**
DOUBLE	**A double-cell data type, used to hold a double number or else a couple of single-cell items working together.**
FLOAT	**A data type wide enough to hold a floating point number. In a 16-bit system with 64-bit reals, a FLOAT would occupy four cells.**

In addition, we are using the following basic type:

TRIPLE	**A data type three cells wide, used to hold three single-cells items working together.**

User data types

```
CHAR        1 BYTE

INT         1 CELL
UINT        1 CELL (UNSIGNED)
LONG        2 CELLS
ULONG       2 CELLS (UNSIGNED)

REAL        1 FLOAT
COMPLEX     2 FLOATS
VECTOR      3 FLOATS
QUATERNION  4 FLOATS
```

User data types

```
C&          An address to a read-only element.
D&          An address to an element (derived from the type above).

C[]         An address to an array of read-only elements, together
            with the number of elements.
D[]         An address to an array of elements, together with the
            number of elements (derived from the type above).

C[,]        An address to a two dimensional array of read-only
            elements, together with the number of rows and columns.
D[,]        An address to a two dimensional array of elements,
            together with the number of rows and columns (derived
            from the type above).
```

C& points to an element that can be read. D& points to an element that can also be modified. D& is derived from C&, as you can do with a normal pointer everything you would do with a pointer to constant plus a few new things.

Categories

All categories allocate enough memory to hold a value of the given type.

VAR	**No action associated with the word. When it is executed, its address is returned.**
AUTO	**When the word is executed, its address is returned and the @ word is applied to it. After the definition, the ! word is applied to its address.**
CONST	**When the word is executed, its constant address is returned and the @ word is applied to it. After the definition, the ! word is applied to its address as if it were not constant.**

CONST words cannot be used inside a struct. AUTO words are not initialised inside a struct.

Manipulators

A manipulator is a state smart word that fetches the next word in the input stream (that has to be an instance of one of the categories given above) and

```
SIZEOF    Returns the size of the object measured in address
          units.

ADDR      Returns the starting address of the object.

TO        Applies the word ! to the object.
```

Examples:

```
+10 CONST INT 5*2

VAR INT FIRST
-5 AUTO INT SECOND

SIZEOF FIRST                    \ returns 4 in a 32-bit system
+10 FIRST !  FIRST @            \ returns +10
+10 TO SECOND  SECOND           \ returns +10
+10 ADDR SECOND !  SECOND       \ returns +10
```

One dimensional arrays

A vector is a collection composed of a fixed number of elements of the same base type. Each of the elements can be used to hold a piece of data and the vector can be manipulated as a whole.

```
10 [] INT DISTANCES  \ defines the vector

SIZEOF DISTANCES     \ returns 40 in a 32-bit machine

DISTANCES .          \ prints the value of all elements
```

The following words should be provided for an array of any of the basic types:

```
SIZE ( d[] -> type -- uint )
     Returns the number of elements in the array.

@    ( d[] -> type uint -- 2nd )
     Fetches one of the elements in the array. An exception is thrown is
     the index is equal or greater than the number of elements in the
     array.

!    ( type d[] -> 1st uint )
     Sets one of the elements in the array. An exception is thrown is the
     index is equal or greater than the number of elements in the array.
```

One dimensional arrays

HEAD (c[] -> *type* uint -- 1st)
Returns an array with some of the first entries in the original array. If the number is greater than the entries in the array, the whole array is returned.

TAIL (c[] -> *type* uint -- 1st)
Returns an array with some of the last entries in the original array. If the number is greater than the entries in the array, the whole array is returned.

-HEAD(c[] -> *type* uint -- 1st)
Returns an array in which some of the first entries in the original array have been removed. If the number is greater than the entries in the array, an empty array is returned.

-TAIL(c[] -> *type* uint -- 1st)
Returns an array in which some of the last entries in the original array have been removed. If the number is greater than the entries in the array, an empty array is returned.

CLONE(c[] -> *type* -- d[] -> 2nd)
Returns a new array with the same data as the given one.

Two dimensional arrays

The following words are modelled after their one dimensional counterparts:

```
SIZE ( d[,] -> type -- uint uint )

@    ( d[,] -> type uint uint -- 2nd )

!    ( type d[,] -> 1st uint uint )

HEAD ( d[,] -> type uint -- 1st )

TAIL ( d[,] -> type uint -- 1st )

-HEAD( d[,] -> type uint -- 1st )

-TAIL( d[,] -> type uint -- 1st )

CLONE( d[,] -> type -- 1st )
```

In addition, we have the following word:

```
ROW  ( d[,] -> type uint -- d[] -> 2nd1 )
     Returns an array with one row of the original array.
```

Strings

A string can be defined as a one dimensional array of characters as in:

```
50 [] CHAR ADDRESS
```

However, as the Forth language allows for direct use of strings, some convenience can be provided.

The keyword STRING defines a constant array of chars with the length of its initialiser:

```
" This is some text" STRING MY-TEXT
```

Structures

```
STRUCT POINT-2D
      VAR INT X
      VAR INT Y
END

POINT-2D  A basic type with an associated size of 8 bytes (in a 32
          bit machine).

X         ( C& -> POINT-2D -- 1ST -> INT)

          The returned address is the same as the parameter.

Y         ( C& -> POINT-2D -- 1ST -> INT)

          The returned address is the parameter incremented in 4
          bytes.
```

Using structures

```
STRUCT DATE
   VAR UINT DAY
   VAR UINT MONTH
   VAR UINT YEAR
END

: ! ( UINT UINT UINT D& -> DATE ) >R
    R@ YEAR !  R@ MONTH !  R> DAY ! ;
: @ ( C& -> DATE -- UINT UINT UINT ) >R
    R@ DAY @  R@ MONTH @  R> YEAR @ ;

: 00 ( UINT ) <# # # #> TYPE ;
: . ( C& -> DATE ) @ ROT 00  '/' EMIT  SWAP 00  '/' EMIT  . ;

15 9 2007 CONST DATE TODAY

TODAY MONTH .                \ prints 9
```

Derived structures

A new structure can be a subtype of another type. In this case, the newer one has the size of the old one (augmented with the size of new components) and can use any word that apply to the old one (unless a new overloaded one is defined).

```
STRUCT POINT-2D
      VAR INT X
      VAR INT Y
END

DERIVED POINT-2D POINT-3D
      VAR INT Z
END

: . ( C& -> POINT-2D ) DUP X .  Y . ;
: . ( C& -> POINT-3D ) DUP .  Z . ;
```

Privacy matters

In an OOP language, the following concepts may be available:

```
PRIVATE    A method that can be used only inside the class.

PROTECTED  A method than can be used inside the class and also
           inside derived classes.

PUBLIC     A method without privacy constraints.

FRIEND     An external method that can access all method in the
           class.
```

In Forth, the use of word lists or packages caters for all these possibilities and then some more.

Future lines of work

The distinction between normal manipulators (that increment the data pointer) and manipulators inside a structure (that increment the structure size) can be complemented with USER manipulators (the increment the offset into the user data space).

```
USER VAR INT UNO

USER AUTO LONG DOS
```

If it were possible to derive structures from a TAGGED one, the tab could be used at runtime to choose the exact function that should be called. This modification is not trivial.

```
TAGGED FIRST
      VAR INT X
END

DERIVED FIRST SECOND
      VAR INT Y
END
```