34th EuroForth Conference

September 14-17, 2018

Doubletree/Hilton Queensferry Hotel
Edinburgh
Scotland

Preface

EuroForth is an annual conference on the Forth programming language, stack machines, and related topics, and has been held since 1985. The 34th EuroForth finds us in Edinburgh for the first time. The two previous EuroForths were held on Reichenau Island, Germany (2016) and in Bad Vöslau, Austria (2017). Information on earlier conferences can be found at the EuroForth home page (`http://www.euroforth.org/`).

Since 1994, EuroForth has a refereed and a non-refereed track. This year there were three submissions to the refereed track, and two were accepted (67% acceptance rate). For more meaningful statistics, I include the numbers since 2006: 27 submissions, 19 accepts, 70% acceptance rate. Each paper was sent to three program committee members for review, and they all produced reviews. The reviews of all papers are anonymous to the authors: This year two submissions were co-authored by program committee members, one of them by the program chair; the papers were reviewed and the final decision taken without involving the authors. Ulrich Hoffman served as secondary chair and organized the reviewing and the final decision for the paper written by the program chair. I thank the authors for their papers and the reviewers and program committee for their service.

One paper was submitted to the non-refereed track in time to be included in the printed proceedings.

These online proceedings (`http://www.euroforth.org/ef14/papers/`) also contain papers and presentations that were too late to be included in the printed proceedings. Also, some of the papers included in the printed proceedings were updated for these online proceedings.

Workshops and social events complemented the program. This year's EuroForth was organized by Paul E. Bennett.

Anton Ertl

Program committee

Contents

A descriptor based approach to Forth strings

Ulrich Hoffmann (FH Wedel University of Applied Sciences), Andrew Read

August 2018

uh@fh-wedel.de, andrew81244@outlook.com

Abstract

We have developed a novel, descriptor based based approach to Forth strings based on the established technique of array slices, most recently exemplified by the Go programming language. We have done this with the goal of creating a strings package that is suitable be incorporated into the Forth kernel. We illustrate the convenience and potential utility of our approach with a lightweight regular expression matcher. We argue that the descriptor based approach provides much of the functionality that has been traditionally obtained with string stacks, but more simply and closely integrated with Forth, and therefore more easilty adopted as a foundational layer in the kernel. We see a potential role for our approach on any Forth system needing more than the bare minimalism of `c-addr u` strings.

1 Introduction

Forth is more than 40 years old, yet strings remain a space for innovation. We are seeking to develop a strings package that can be implemented as a foundational layer in the Forth kernel upon which the string processing requirements of the Forth kernel can be based. Our descriptor-based approach has been informed by the technique of array slices, as most recently exemplified in the Go programming language.

Our paper begins with brief examples of array slices drawn from other computer languages. Moving on to Forth, we review Anton Ertl's main observations from his EuroFORTH 2013 paper "Standardize Strings Now!" [5]. We consider some other contributions to Forth strings that have appeared over the years, before stating our goals and evaluation criteria, describing our approach and illustrating it with a lightweight regular expression matcher that we have developed from original code by Rob Pike and Brian Kernighan. We discuss the advantages and limitations of our approach and make a brief comparative analysis with strings stacks. Finally we consider the suitability of our descriptor based approach for different Forth environments.

2 Array slices

The Go programming language has both arrays and slices. Arrays are conventional data-structures holding a collection of elements of the same type. Go arrays are value types: assigning a Go array to a new variable creates a copy of the array. Slices are reference-type wrappers for arrays. Slices refer segments of an array that may be manipulated without copying or modifying the underlying data [1]. Multiple slices may reference the same array, potentially to overlapping ranges within it. Go arrays themselves have a fixed size but slices vary in size. Slice functions might create new slices, which reference a newly allocated array with larger capacity and copy over the elements. The old array might later be garbage collected if not referenced by other slices.

Other examples include: Algol 68, which permitted slices as references to subsections of arrays specified between lower and upper bounds, Fortran 77, which offered array slices with sophicticated capabilities for slicing multi-dimensional arrays in any direction, and Sinclair BASIC (ZX80/81 and ZX Spectrum), which offered simple but convenient slicing of character strings [2, 3, 4].

3 Standardize Strings Now!

Anton Ertl began "Standardize Strings Now!" by arguing that the desirable properties for Forth strings are (i) ease of use and (ii) integration with the rest of Forth.

The first part of Anton's article discussed these requirements as they relate to memory management of the character buffer. One barrier to efficient string handling is the allocation and deallocation of character buffers as strings are consumed by words on the stack. Anton illustrated the pitfalls of either ignoring the problem altogether (leading to memory leaks or to the recruitment of the ever-unpopular garbage collector) or attempting to manage allocation and deallocation concurrently with the manipulation of strings (leading to difficult code and reliance on `allocate`, a word that many small systems prefer not to support). Anton explained how region-based memory allocation can assist by collecting related data into a single region that can easily be freed as a block.

The second part of Anton's article discussed the convenient representation of strings. Anton highlights some of the current issues: the `c-addr u` format is flexible because it represents strings of arbitrary length and content and allows sub-strings to be taken without copying the buffer. A disadvantage of this approach is that it takes two cells to represent each string and this becomes cumbersome with several strings. The counted string format needs only one cell on the stack (a pointer to the count) but sub-strings cannot be taken without copying the buffer, and traditional implications of the counted cell format can only represent strings with up to 255 bytes.

A practical illustration of the stack depth difficulty with the `c-addr u` representation is illustrated by Anton's example of a regular expression matching word

```
: search-regexp ( c-a1 u1 c-a2 u2 -- c-a1 u3 c-a4 u4 c-a5 u5 true | false )
\ Search for regexp c-a2 u2 in string c-a1 u1
\ if the regexp is found, c-a1 u3 is the substring before the first match,
\ c-a4 u4 is the first match, and c-a5 u5 is the rest of the string, and
\ TOS is true, otherwise return false
```

A successful match returns 7 parameters on the stack (3 strings and a flag). Anton suggested the possibility of implicit parameters and context-wrappers among other techniques for reducing stack depth, and concluded in general by favouring the `c-addr u` format over most alternatives.

4 Other contributions

Strings stacks with special features for handling strings have been around since the early-days of Forth. For example Klaus Schliesiek's 1986 string stack implementation, later modernized and updated for Forth-94 and Forth-2012 compliant systems by Ulli Hoffmann [8]. One attraction of string stack is their capability for handling multiple strings without cluttering the parameter stack. For example:

```
: "delimiter-join (" s1 s2 ... sn delim -- s ) ( n -- )
\ Concatenate the n strings sn, ... s2, s1 with
\ sn at the beginning of the resulting string s
\ interspersed with the delimiter string delim.
\ " ef" " cd" " ab" 3 " /" "delimiter-join results in ab/cd/ef.

: "delimiter-split (" s0 delim -- s1 ... sn ) ( -- n )
\ Split the text s0 on occurances of the delimiter string delim.
\ s1 to sn are the resulting parts. sn is the closest to the beginning of s0.
\ The delimiter is removed. n is the number of parts.
```

Brad Rodriguez developed PatternForth to provide SNOBOL4-like string processing and pattern matching functions. PatternForth, in a different context, incorporates the concept of string descriptors [13]. More recently Carsten Strotmann has suggested that REXX Parse provides a worthwhile model that may be useful for inspiring future work with Forth strings [6]. Ulli Hoffmann has also demonstrated a very light wordset for string handling that represents strings entirely on the parameter stack as a count followed by characters one-cell-at-a-time. This approach that has the advantage of minimal dependency on the rest of the Forth dictionary [7].

5 Goal and evaluation criteria

Our motivation in developing this strings package is connected with the our concept of a "New Synthesis" for Forth [12]. We seek to develop an approach to strings that can be built into the Forth kernel and used to support the string handling requirements of the kernel, such as I/O operations, input stream processing and maintenance of the Forth dictionary. The New Synthesis is a concept under development, nevertheless some criteria suggest themselves based on these ambitions.

The strings package must be programmed before the kernel is completed, so it must be developed using an elementary Forth vocabulary. The string handling facilities of the kernel will be developed out of our package, so logically we cannot rely on those facilities in developing the package itself. We do not want the strings package to commit us to fixed decisions elsewhere in the kernel, such as choices about memory management. We would prefer the compilation size of our package to be small (although not necessarily minimal, recognizing that all design decisions in a Forth system are eventually tradeoff decisions). We must ensure adequate performance, and experience suggests that we must therefore avoid any approach that relies on copying string data. Ideally there should be some "killer application" for our package within the kernel itself, and for us this is the opportunity to parse the input stream with regular expressions (see section 6). Finally, the strings package should be generally usable and useful for applications running on our kernel that we cannot yet anticipate.

6 A descriptor based approach to strings

6.1 Outline

Our strings are single-cell references on the parameter stack. Each reference (our 'string') points to a small block of string meta-data (our 'descriptor') located in memory. The descriptor, among other information, holds a flexible onward reference to an ASCII character buffer where the string data resides.

The descriptor based approach allows strings to be duplicated or sub-strings to be created without any copying of character data. We have also established two classes of strings: 'permanent' strings and 'temporary' strings, which function similarly, but which have different consequences for memory management.

Before describing our approach in more detail we provide an illustration of usage. String operators in our wordset are prefixed with `$` (e.g. `$initialize, $make, $len, $s`) while strings themselves are suffixed with `$` (`a$, pad$`, etc.).

```
\ reserve space for 10 string descriptors
10 $initialize

\ Establish an ASCII character buffer with 9 character of text
\ and 20 characters of extra capacity
S" Veni vidi01234567890123456789"

\ Create a permanent string and leave its descriptor on the stack
\ : $make ( c-addr len size flag -- s$)
\ permanent = -1
9 swap permanent $make

\ Create another string, this time a temporary string with no extra capacity
\ temporary = 0
S"  vici" dup temporary $make

\ Concatenate the two strings (see the wordlist that follows)
\ : $+ ( s$ r$ -- s$)
$+

\ Enquire as to the length of the new string
\ : $len ( s$ -- s$ n)
$len CR .
14

\ Provide a legacy reference to the string, and type it out
\ : $s ( s$ -- c-addr u)
$s CR type
Veni vidi vici
```

6.2 Overview of the descriptor approach

Figure 1 illustrates the descriptor approach. Our descriptors are structures with four cell-width fields: the address field points to the start address of the character buffer, the size field holds the total size of the character buffer in bytes, the start field holds the position within the character buffer of the first character of the string (counted from zero for the first byte of the buffer), the length field holds the count of characters in the string. A single flag bit[1] specifies whether the string is permanent or temporary (see the following section on Memory management). Empty strings are represented by descriptors with a length field of zero.

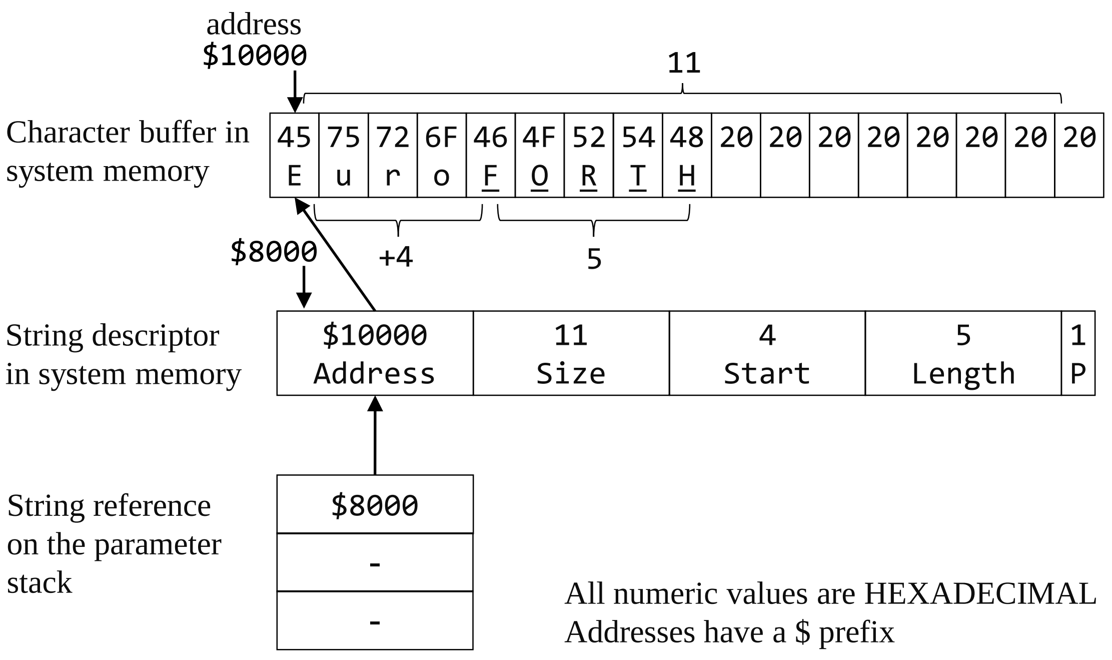

address
$10000
11
Character buffer in system memory
45 75 72 6F 46 4F 52 54 48 20 20 20 20 20 20 20 20
E u r o F O R T H
+4
5
$8000
String descriptor in system memory
$10000 Address
11 Size
4 Start
5 Length
1 P
String reference on the parameter stack
$8000
-
-
All numeric values are HEXADECIMAL
Addresses have a $ prefix

Figure 1: The descriptor approach illustrating a reference to the characters "FORTH" (underlined) held in a character buffer. In this example the permanent flag of the descriptor has arbitrarily been set to TRUE. Memory addresses and data are purely illustrative.

Strings are duplicated by creating a replica descriptor without copying any character data, as illustrated in figure 2. Sub-strings are taken by modifying the start and length fields. A string can be appended to by copying data into the character buffer and incrementing the length field, provided that the size of the character buffer has sufficient capacity. Traditional `c-addr u` references can be obtained by computing the address of the first character in the string (address + start) and also providing the length. More complex manipulations, such as inserting and deleting characters within a string, can be carried out by modifying the character buffer, again subject to the constraint of the size of the character buffer.

6.3 Memory management

Memory management needs to address two separate issues: management of the character buffers and management of the descriptors themselves.

6.3.1 Character buffers

In our implementation we explicitly do not memory manage the character buffers. To create a string descriptor it is necessary that the proposed character buffer already exist in memory. When a string descriptor is recycled (see below), the character buffer remains intact. In consequence, character buffers need to be memory managed separately. We discuss the merits and limitations of this approach later.

[1]In our implementation the permanent/temporary bit is actually the MSB of the size field, so that strings are limited in size to 2^31 bytes on a 32 bit system. We require that the MSB of the length and start fields must be zero. However different structures would be equally feasible.

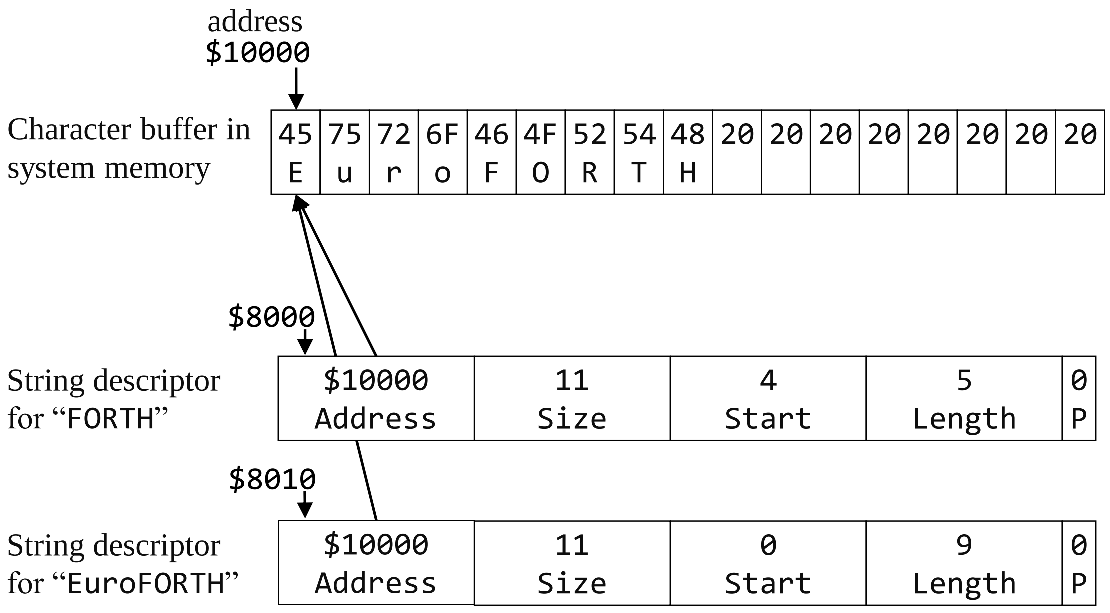


Figure 2: Two descriptors that point to different substrings in the same character buffer. The second descriptor may have been created as a duplicate of the first and later modified. Note that changes to the string buffer would affect both strings.

6.3.2 String descriptors

We reserve memory for a fixed number of string descriptors at initialization[2] (a pool approach). We need to identify those string descriptors within the pool that are unallocated and available for creating new strings. In our first iteration, we opted to use a scheme with an additional stack stack set up in memory to hold the addresses of unallocated descriptors. The inefficiency of this approach was pointed out by the anonymous academic reviewers who suggested a revised approach that we have now adopted. This is a linked list in which the address field each unallocated descriptor points to the address of the next unallocated descriptor. A global variable within our strings package holds the address of the first free descriptor and the list is terminated with a zero pointer. The descriptor pool is illustrated in figure 3.

6.3.3 Recycling descriptors

As noted above, our strings are represented by a single-cell reference on the stack. Applications may discard these references as strings are no longer required. Without an appropriate mechanism to recycle the descriptors of discarded strings the pool of available descriptors would eventually become depleted.

We differentiate between permanent and temporary strings, an assignment made when a string is created. Permanent strings are never recycled, temporary strings are subject to the recycling mechanisms described below. Having both permanent and temporary strings allows flexibility for strings that have different intended uses. Looking ahead, we might create a permanent string to refer to a repeatedly-used regular expression, whilst we might create a temporary string to refer to characters in some temporary buffer.

The recycling word is `$drop (s$ --)`. `$drop` take the string `s$` from the top of the stack and recycles it. When a string descriptor is recycled it is replaced in the pool of available descriptors. We also take the precaution of “spoiling” a recycled string (i.e. setting its size, length and start position to zero), with the intention of making it harder to inadvertently keep using a descriptor after recycling it. As noted already, when a string is recycled by `$drop` the character buffer to which it referred is left untouched. Also `$drop` detects permanent strings and does not recycle them.

[2]In our implementation we use ALLOCATE to reserve the necessary storage for string descriptors, but as this is a once-only procedure at initialization time it would be equally possible to reserve the necessary storage with ALLOT, or in some platform-dependent manner.

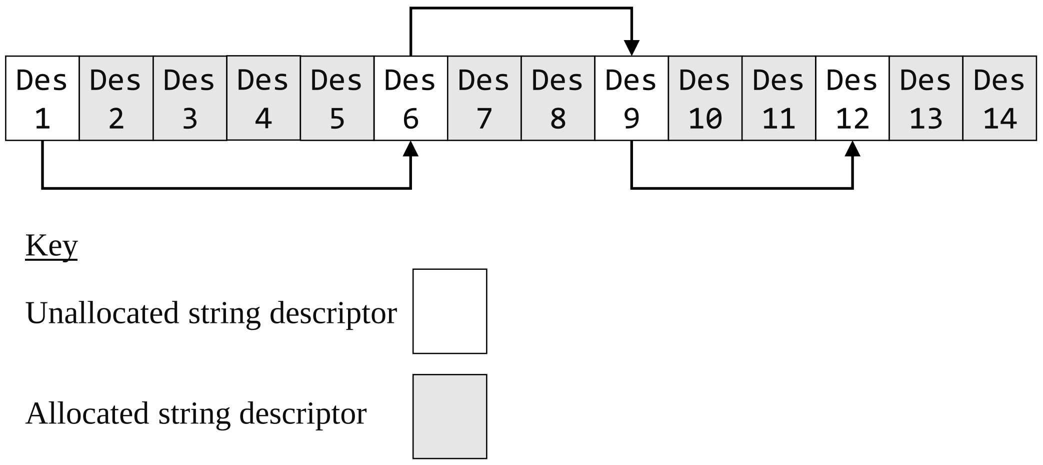


Figure 3: Illustration of the descriptor pool and stack of available descriptors. In this example 10 descriptors have been allocated out of 14 in the pool. References to the 4 unallocated descriptors are held in a linked list.

At this point it may be worth comparing the actions of `$drop` with `drop`. Both `$drop` and `drop` will, if there is a string descriptor on the top of the stack, remove that string descriptor from the top of the stack. However `$drop` will also recycle the descriptor (assuming that it was a temporary string). By contrast `drop` will not recycle the descriptor, which will remain intact.

Words in our string wordset that consume strings as parameters will recycle their descriptors. For example

```
: $+ (s$ r$ -- s$)
\ Append the contents of string r$ to s$ and return s$.
\ The length of the string is always truncated to fit within the size of s
\ r$ is internally passed to $drop for recycling
```

In cases where we do not want to consume a string parameter, the convention is that we do not consume it. For example

```
$len ( s$ -- s$ n)
\ Return the length of a string in count of characters
```

To summarize the convention that we have adopted: the recycling of a string descriptor can be forced with `$drop`, otherwise string descriptors are recycled automatically when they are consumed by a word in our string wordset. However those strings designated at the time of creation as permanent are not recycled in these cases.

6.3.4 Duplicating strings

With our approach it is possible to duplicate strings without copying the underlying character data. The duplication word is `$dup ( s$ -- s$ r$)`. `$dup` takes the string descriptor `s$` and creates a new string descriptor `r$`. `r$` has the same size, length, and start position as `s$`, and it also refers to the same character data. However r$ and s$ refer to separate, distinct descriptors.

Compare the action of `$dup` with `dup`: both `$dup` and `dup` will, if there is a string descriptor on the top of the stack, place a second string descriptor on the top of the stack. However `$dup` actually creates a new descriptor. In this case if `s$` is subsequently modified by taking a sub-string with `$sub`, or by adding another string with `$+`, or recycled with `$drop`, `r$` will not be affected. Of course `s$` and `r$` still refer to the same underlying character data and any operations that affect that character data directly (such as inserting or deleting characters) will affect both `s$` and `r$`.

drop: remove the descriptor from the stack but do not recycle it	**$drop**: remove the descriptor from the stack and recycle it (if it is a temporary string)
dup: duplicate the reference to the same descriptor	**$dup**: create a new descriptor and provide a reference to it

Table 1: Comparison of drop and $drop, and dup and $dup

Table 1 compares the actions of `drop` and `$drop`, and `dup` and `$dup`.

6.4 Wordlist

To complete the description of our descriptor based approach to strings we now list the main words currently in our wordlist.

```
: initialize ( N --)
\ Initialize the string descriptor system with space for N strings

: $make ( c-addr len size flag -- s$)
\ Make a new string descriptor referencing character data at c-addr
\ The character buffer contacts len bytes of valid data, starting at c-addr
\ and has total capacity of size bytes.
\ If size > len then the string has spare capacity to be extended
\ flag = TRUE for a permanent string; FALSE for a temporary string

: $drop ( s$ --)
\ Recycle a the descriptor s$, unless s$ is a permanent string
\ The character data itself is not deallocated

: $s ( s$ -- c-addr u)
\ provide a legacy reference to a string

: $len ( s$ -- s$ n)
\ return the length of a string in characters

: $size ( s$ -- s$ n)
\ return the size of the character buffer holding the string

: $dup ( s$ -- s$ r$)
\ Copy the string descriptor s$ to a new string descriptor r$
\ Both $s and $r reference the same character data in memory
\ but can take different cuts

: $sub ( s$ a n -- s$)
\ Modify s$ to reference the substring starting at position a
\ and running for n characters
\ a is calculated as an offset from the current start position,
\ not the start of the buffer
\ a is permitted to be negative; n should be positive

: $app ( s$ c-addr u -- s$)
\ Append the text characters from c-addr u to s$ and return the augmented string
\ The length of the string is always truncated to fit within size

: $+ ( s$ r$ -- s$)
\ Append the contents of string r$ to s$ and return s$.
\ The length of the string is always truncated to fit within size

: $= ( s$ r$ -- s$ r$ flag)
\ Compare the character strings s$ and r$ and return true if they are equal
\ Note, this compares the characters in the buffer, not the descriptors

: $rem ( s$ a n -- s$)
\ Remove n characters from s$ starting at position a
\ Following characters within the character buffer are moved as necessary

: $ins ( c-addr u s$ a -- s$)
\ Copy the text characters from c-adder u into s$ at position a
\ Following characters within the character buffer are moved as necessary
\ The length of the string is always truncated to fit within size
```

7 Illustrative application - a regular expression matcher

In section 2 we mentioned Anton Ertl's illustration of the cumbersome stack signature for a regular expression matcher and suggested solutions. The descriptor based strings approach offer an alternative and explains some of our design decisions. Thanks to single cell string references, the regular expression matcher accepts 2 parameters and returns no more than 4. Placing the length, start and size within the string descriptor allows us to partition the original string according to the results of the regular expression search without any string copying.

The regular expression matcher is less that 250 lines long, including full comments. This code originated as a direct port of a C program written in 1998 by Rob Pike and Brian Kernighan [11]. For the sake of brevity and simplicity we do not attempt to support more complex regular expressions, but the code is straightforward enough that extensions could be added by a user if needed. We have however extended the basic set of matches to include specific regular expressions focused on parsing the Forth input stream, see table 2.

The wordlist is briefly illustrated below. `$regex` calls `match`, which takes `c-addr u` strings as input parameters and returns the character location of the match as the output. `match` can be used independently of the descriptor based string library, but in that case the application must perform additional manipulation to extract the matched and unmatched portions of the string.

```
: $regex ( s$ r$ -- a$ b$ s$ TRUE | FALSE)
\ Search for regex r$ in string s$ if the regexp is found, a$ is the
\ substring before the first match, b$ is the first match
\ s$ (modified) is the rest of the string and the TOS is true;
\ otherwise return false and preserve s$ unmodified
\ r$ is $drop'ed (recycled unless defined to be a permanent string)
\ a$, b$ and s$ all reference portions of the same character data in memory

: match ( addrT uT addrR uR -- first len TRUE | FALSE )
\ search for regexp (addrR uR) anywhere in text (addrT uT)
\ return the position of the start of the match, the length of the match,
\ and TRUE or FALSE if there is no match
```

8 Discussion

8.1 Some design considerations

8.1.1 Different descriptors pointing to substrings in the same mutable string

Figure 2 illustrates two separate descriptors pointing to different, but overlapping, substrings in the same mutable string buffer. This seems like a recipe for disaster. However there are advantages if used with care, and we have one "use case" that we believe is compelling. Let the input stream be represented by a string descriptor, and proceed to parse it with our regular expression matching word, `$regex`. A successful match returns three descriptors, all of which point to non-overlapping sections of the original input stream buffer. This is very efficiently achieved without any string copying. Furthermore, the string descriptor that represents the input stream is automatically adjusted to the unmatched portion, also without any string copying. We anticipate there could be other instances within the kernel where the ability to split and alter strings without copying may be an advantage.

8.1.2 Automatic recycling of consumed string descriptors

Our approach is to automatically recycle the relevant descriptors when strings are consumed by string manipulation words. The alternative would have been to not do this and leave the user to recycle all strings at the end of their lifetime with `$drop`. The potential advantage is reduced need for string duplication with `$dup` in cases where a string will still be needed after a string manipulation word consumes it as a parameter.

We felt the over-riding consideration was to make management of the descriptors as trouble-free for the user as possible and that this was best achieved by the approach we have taken. In addition, it is a simple matter to

Regex	**Match**
^	Beginning of string
!	Beginning of string disregarding succeeding whitespaces
$	End of string
.	Any character, including newline
\	Quote or special character
a*	Zero or more a's
a+	One or more a's
a?	Zero or one a's (i.e. optional a)
~a	Not a (i.e. any character other than a)
\t	Tab
\n	Linefeed
\r	Carriage return
\s	Any whitespace
\S	Any non-whitespace
\d	Any decimal digit
\h	Any hexadecimal digit (case insensitive)

Table 2: Regex's supported by our regular expression matcher

arrange for a string manipulation word not to consume its argument when it is anticipated that the string will be needed again, and this is illustrated by the stack signatures of `$len` and `$size`.

We have provided for permanent strings that are totally immune to recycling. Our motivation for permanent strings is the opportunity to assign certain components within the kernel to descriptor based strings, where it would be convenient for system operations not to have them recycled. For example we anticipate building a kernel in which the input stream and regular expressions for certain input types (such as decimal and hexadecimal numbers) are represented by permanent strings.

8.2 Relationship to the string stack approach

Our approach can be compared to a non-copying string stack as illustrated in figure 4. The similarity is that the string is ultimately represented by an abstract datatype that holds more information than simply the length and address of the string. In string-stack system these datatypes reside on the stack directly. In our system the abstract datatypes have been placed in a pool of descriptors, while the references to those descriptors are placed on the parameter stack.

It can be argued that the string descriptor approach is simpler and better integrated with Forth. Firstly, there is no need for a separate string stack with its own suite of stack handling words. Secondly, both string and non-string arguments can combined in the signatures of string-handling words. The "secret weapon" of the descriptor-based approach that gives string-stack like functionality on the parameter stack is the subtle differences in operation between `$dup` and `dup`, and `$drop` and `drop`.

Another relative convenience of the descriptor based approach as compared with a string stack is that it is possible to convert `c-addr n` strings into descriptor strings, or vice-versa, with a lightweight function call that does not modify or copy the string buffer. Finally, the descriptor based approach leaves the allocation and deallocation of string buffers to the application which is likely to be best placed to optimize resources based on its needs, rather than allocating an entire block of memory for a string stack at initialization.

However string stacks do have the advantage of being able to conveniently hold and manipulate multiple separate strings (for example to split or join). The descriptor approach could potentially assist in this case as noted below.

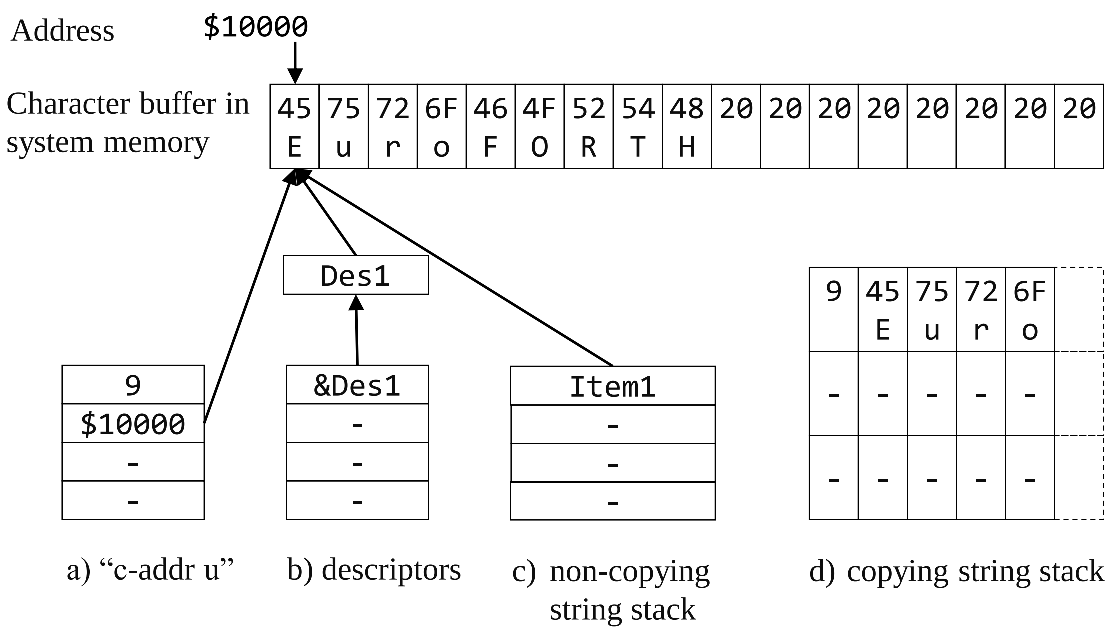


Figure 4: Comparison of alternative string representations. The "c-addr u" format holds the size and address of the string on the parameter stack. The descriptor approach holds the relevant string information in a descriptor that is pointed to by a single cell reference on the parameter stack. A non-copying string stack sets up a new stack to hold essentially the same information as is found in a string descriptor. In all of these three methods the character data itself is located in a separate character buffer that must be separately managed. Finally a copying string stack holds the length and character data directly on a separate stack and is responsible for managing the allocation and deallocation of that character data. Multiple string stack implementations exist and these examples are illustrative only.

We imagine that our descriptor-based approach could be extended to other abstract datatypes that have traditionally often been implemented on separate stacks. The most obvious candidates might be arbitrary-size integers or floating point numbers. Willi Striker has demonstrated that an efficient floating point arithmetic wordset can be obtained by holding floating point numbers as references on the parameter stack [9], and espoused as similar approach: "Everything on the stack is either an integer or a reference!" [10]. Another candidate datatype to benefit from a descriptor based approach might be collections, such as collections of strings themselves. This could assist in managing string intensive applications.

8.3 Response to Standardize Strings Now!

8.3.1 Memory management of character buffers

We have not solved the first issue that Anton raised: how to manage memory manage the character buffer. We explicitly leave allocation and deallocation of character buffer out of scope of our word-set. Our justification for this is that we want to implement our package in the Forth kernel at a foundational level and prefer to be adaptable to whatever kind of kernel-level character buffer any particular system adopts. These could vary widely.

We do provide an approach to the buffer allocation problems around concatenating strings. By separating the length of a string from the size of its buffer we allow for strings to be created ready with spare capacity. Whilst this approach still requires the programmer to anticipate the required capacity in advance, this appears to be a sensible compromise position between having no spare capacity at all (traditional Forth strings) and a some fully-dynamic but complicated approach. As pointed out in our introduction, this idea is not arbitrary but has the credential of being a core feature of array handling in Go and several other languages before it [1].

8.3.2 Convenient string representation

Our strings are conveniently held on the parameter stack as single cells, but do not suffer from any of the limitations of counted strings. Do the descriptors themselves create an inconvenience or risk of memory leaks? We argue not. Our system manages its own pool of descriptors and enforces automatic recycling when strings are consumed by string manipulation words, so the user has minimal work to do in remembering to $drop (rather than drop) strings that are no longer required. Errors in this regard can be minimized with stack comments and a brief code review.

8.4 Comparison with Go slices

A Go slice is a descriptor of an array segment. It consists of a pointer to the array, the length of the segment, and its capacity (the maximum length of the segment). In this, our descriptors are similar to Go slices. However Go slice functions may cause arrays to be reallocated, copied and otherwise memory managed. By contrast our words cannot, by design, affect the character buffers referenced by the descriptors. This is significantly less functionality, but consistent with our approach of leaving memory management elsewhere in the kernel.

8.5 Limitations

Naturally we acknowledge limitations introduced with our approach. The most significant limitation is the overhead introduced. Although our package is lightweight and performance overhead is therefore unlikely to be an issue, especially considering that I/O is the typical bottleneck in most string applications, extra memory is required and this must be reserved in advance for both the pool of descriptors and the string handling words themselves. Many small systems simply have no need for regular expressions or sophisticated string handling. It is enough to simply process the input steam and display messages to the user. In such cases the `c-addr u` format is both effective and efficient.

Another limitation is the fixed size of the descriptor pool. However with our stack-based approach to organizing free descriptors, the memory reserved for descriptors does not need to be contiguous and so it would not be too difficult to add further descriptors to the pool after initialization if to do so would be worthwhile.

9 Conclusion

We have presented a descriptor based approach to strings that draws on familiar ideas from other programming languages, but which we are not aware has been tried in Forth before. Although we have positioned this paper in part as a response to Anton Ertl's "Standardize Strings Now!", we do not necessarily suggest our approach as candidate for string standardization. Other approaches also have their merits, and the standard must weight common usage more highly than special innovations. As we also note above, our approach would also likely be a cost rather than a benefit to small systems.

However on systems where the string problem is relevant, we believe our approach offers an elegant and useful "middle-way" between the stark minimalism of `c-addr u` and the full-blown weight and complexity of string stacks with dynamic buffer management. Our intention is that our strings package be implemented at kernel level and used to support kernel-level string operations. To demonstrate the combination of easy and utility that our approach offers, we have developed a powerful yet lightweight descriptor-based regular expression matcher.

We acknowledge a debt to programming pioneer Rob Pike both for the exemplification of the slice mechanism in the Go Programming language and for his remarkably simple C regular expression matcher upon which our own is based. We wish to thank the anonymous academic reviewers for the helpful comments that greatly improved our paper.

References

[1] Go Slices: usage and internals, The Go Programming Language, January 2011, https://blog.golang.org/go-slices-usage-and-internals

[2] Sinclair ZX Spectrum BASIC Programming Manual, Chapter 8, 1982, Sinclair Research

[3] Informal Introduction to Algol 68, Section 2.5, Lindsey and van der Meulen, 2nd ed. 1977

[4] Fortran 77 Language Reference Manual, Chapter 2, Sun Microsystems, 2001

[5] Standardize Strings Now! Anton Ertl, TU Wien, EuroFORTH 2013

[6] REXX Parse (informal presentation), Carsten Strotmann, EuroFORTH 2013

[7] "Stack of Stacks", Ulrich Hoffmann, Forth Tagung 2017, https://wiki.forth-ev.de/doku.php/events:tagung-2017

[8] String Stack, Ulrich Hoffmann, first committed 2015, https://github.com/uho/stringstack

[9] Forth Floating Point Word-Set without Floating Point Stack, Willi Stricker, EuroFORTH 2013

[10] Video presentation (minutes 14:00 to 18:00), Willi Stricker, EuroForth 2013

[11] Article, http://www.cs.princeton.edu/courses/archive/spr09/cos333/beautiful.html, Rob Pike and Brian Kernighan, 1998

[12] RFC: Forth, a New Synthesis, Andrew Read and Ulrich Hoffmann, EuroForth 2017

[13] PatternForth: A Pattern-Matching Language for Real-Time Control, Brad Rodriguez, Bradley University, 1989

Closures — the Forth way

M. Anton Ertl[*]
TU Wien

Bernd Paysan
net2o

Abstract

In Forth 200x, a quotation cannot access a local defined outside it, and therefore cannot be parameterized in the definition that produces its execution token. We present Forth closures; they lift this restriction with minimal implementation complexity. They are based on passing parameters on the stack when producing the execution token. The programmer has to explicitly manage the memory of the closure. We show a number of usage examples. We also present the current implementation, which takes 109 source lines of code (including some extra features). The programmer can mechanically convert lexical scoping (accessing a local defined outside) into code using our closures, by applying assignment conversion and flat-closure conversion. The result can do everything one expects from closures, including passing Knuth's man-or-boy test and living beyond the end of their enclosing definitions.

1 Introduction

The addition of locals, quotations[1] and `postpone` to Forth leads to the question of how these features work together. In particular, can a quotation access the locals of outer definitions (i.e., is the quotation a closure)? The Forth200x proposal for quotations chose not to standardize this, because there is too little existing practice in the Forth community; however, it does encourage system implementors to experiment with providing such support, and this is what we did for the present paper.

In other programming languages, particularly functional programming languages, access to outer locals is a valuable feature that increases the expressive power[2] of these languages.

Also, can a local be `postpone`d, and if yes, what does it mean? Forth-94 chose to not standardize this.

However, implementing these features in its most powerful and convenient form requires garbage collection, which is not really appropriate for Forth. So we have to find a good compromise between expressive power and convenience on one hand, and ease of implementation on the other. The contribution of this paper is to propose such a compromise.

In this paper, we first present the principles and syntax of our new features (Section 2); next we give usage examples for these features (Section 3), as well as alternatives that do not use them; next we give an overview of the implementation (Section 4); then we discuss the relation between our flat-closure feature and lexical scoping (Section 5); we also give some microbenchmark results that give an idea of the performance of our implementation (Section 6); finally, we discuss related work (Section 7).

2 Closures: Principles and Syntax

2.1 Overview and principles

Quotations have been accepted into the next version of the Forth standard in 2017, but they do not define what happens on access to locals of enclosing definitions. Consider the following minimal example:

```
: foo {: x -- xt :}
  [: x ;] ;

: bar {: x -- xt1 xt2 :}
  [: x ;]
  [: to x ;] ;

5 foo 6 foo
execute . execute . \ prints 6 5

5 bar over execute . \ prints 5
6 swap execute
execute . \ prints 6
```

Some people may wonder what this means. It is not necessary to know this to understand most of this paper (we use a different syntax), but in case you really want to know, the rest of this paragraph

[*] `anton@mips.complang.tuwien.ac.at`

[1] Nameless colon definitions inside other colon definitions, http://www.forth200x.org/quotations.txt

[2] The expressive power refers not just to what can be expressed (all interesting languages are Turing-complete and can compute the same things, given enough resources), but also to the ease and versatility of expression.

explains it. Following the example of Scheme[3], every invocation of `foo` (or `bar`) creates a new instance of the local `x`, and an xt (two xts for `bar`) for the quotation. Calling this xt (these xts) accesses the instance of `x` that was created in the invocation of `foo` that produced the xt. Yes, this means that different invocations of `foo` produce different xts.

Terminology: In the programming language literature, a nested definition (or quotation) that accesses a local of an enclosing definition is called a **closure**. This is also the name of a data structure used for implementing this feature. We provide the data structure, and call it **closure**, but leave closure conversion (the process by which other programming language compilers translate from source-code closures to data-structure closures) to the programmer. In most of the rest of this paper, **closure** refers to the data structure, and it's Forth source code representation.

We do not support the syntax shown above. Instead, in the minimal version of our syntax, these words can be written as follows:

```
: foo {: x -- xt :}
  x [{: x :}d x ;] ;

: bar ( x -- xt1 xt2 )
  align here swap , {: xa :}
  xa [{: xa :}d xa @ ;]
  xa [{: xa :}d xa ! ;] ;
```

`[{:` starts a closure and a definition of passed-in locals of the closure.

The decisive difference between a closure and a quotation that starts with a locals definition is that the locals of the closure are initialized from the values that are on the data (and FP) stack at the time when the quotation's xt is pushed on the data stack, while a quotation with locals at the start would take the locals from the stack when the xt is `executed` (maybe much later). In this way, the closure gets data from its enclosing definition that it can use later.

The other difference is that the locals definitions in these closures end with `:}d`, and that means that the memory needed for the closure is stored in the dictionary (`alloted` space).

These examples demonstrate the principles of our approach:

Explicit memory management of closures: Closures can live longer than the enclosing definition. The programmer decides where the memory for closure is allocated, and how it is reclaimed. The memory can be allocated and reclaimed like locals, allocated with `allocate` and reclaimed explicitly with `free`, allocated in the dictionary, or allocated with some user-defined allocator (such as the Forth garbage collector[4], or region-based memory allocation [Ert14]).

Copying locals into closures: Locals in a closure are a separate copy of the outer local when used in the way shown above. For read-only locals, this is no problem.

This approach of creating copies of values of read-only locals is known as **flat-closure conversion**. In other programming languages, the compiler performs flat-closure conversion implicitly (or uses a different implementation approach); in our Forth extension, the programmer performs it explicitly.

Explicit management of writable locals: For writable locals, we usually do not want separately modifyable copies, but want to access one *home location*. In our approach, home locations are allocated (and memory managed) explicitly (with `align here swap ,` in the `bar` example). The addresses of these home locations are read-only and copied into the closures, like other read-only values. The home locations are accessed with memory words, such as `@` and `!`, as shown in the `bar` example. This approach is called **assignment conversion**.

Our syntax is more verbose, but also more flexible than simply allowing access to outer locals: The locals in the closures can have a different name from the corresponding locals in the enclosing definition, and actually, there is no need to define a value as a local in in the enclosing definition. E.g., we could also define these words as follows, and achieve the same effect:

```
: foo ( x -- xt )
  [{: x :}d x ;] ;

: bar ( x -- xt1 xt2 )
  align here swap ,
  dup  [{: xa :}d xa @ ;]
  swap [{: xa :}d xa ! ;] ;
```

2.2 Closure words

These words are used for defining and memory-managing closures (without conveniences for dealing with read/write locals).

[3]The most popular of the early programming languages that got this right.

[4]http://www.complang.tuwien.ac.at/forth/garbage-collection.zip; however, the current version of the garbage collector does not recognize closures as live by seeing their xt, because the xts do not point to the start of the memory block.

`[{:` (C: -- closure-sys) Compilation: Start a closure, and a locals definition sequence.

`:}d` (C: closure-sys -- quotation-sys colon-sys) Compilation: End a locals definition sequence.

Enclosing definition run-time: Take items from the data and FP stack corresponding to the locals in the definition sequence, create a closure in the dictionary. The `;]` that finishes the closure pushes the xt of that closure.

`:}h` (C: closure-sys -- quotation-sys colon-sys) Like `:}d`, but the closure is `allocated` (the `h` stands for *heap*).

`:}l` (C: closure-sys -- quotation-sys colon-sys) Like `:}d`, but the closure is created on the locals stack[5] in the enclosing scope. I.e., it lives as long as a local defined in the same place.

`:}*` (C: closure-sys xt -- quotation-sys colon-sys) A factor of `:}d :}h :}l`, usable for defining similar words for other allocators. The passed xt has the stack effect `( u -- addr )` and allocates `u` address units (bytes) of memory.

`:}xt` (C: closure-sys -- quotation-sys colon-sys) Similar to `:}*`, but the xt is pushed at the enclosing definition run-time, before `[{:`. Usage example: `['] allocd [{: x :}xt x ;]`

`>addr` (xt -- addr) `Addr` is the address of the memory block of the closure identified by `xt`. Typical use: `( xt ) >addr free throw`.

2.3 Gforth features

This subsection describes some Gforth features that make the closure words nicer to use, or that are used in the examples in the rest of the paper.

Locals definers

Gforth cannot just define cell-sized locals, but also, e.g., FP locals, by putting `f:` before the local. An old [Ert94], but (up to now) little-used feature is *variable-flavoured locals* where using a local pushes the address of its location on the data stack, and accesses to the values are performed with words like `@ !`. Variable-flavoured locals are defined by putting one of `w^ f^ d^ c^` before the name of the local (for a cell, a float, a double, or a char respectively). Given that writable locals in closures are based on passing the address of the home location of the local around, this feature finally becomes interesting. Example:

[5]Or on the return stack on systems that keep locals there.

```
{: f: r w^ x :}
r f. 1e to r
x @ . 1 x !
```

This code fragment first defines a value-flavoured FP local `r`, and then a variable-flavoured local `x`, then shows a read and a write access to `r`, then a read and a write access to `x`.

VFX Forth supports defining local buffers, which can also be used for defining home locations for read/write locals that live until the definition is exited.

Gforth also has a *defer-flavoured* locals definer: if you define a local `x` with the definer `xt:`, an ordinary occurence of `x` `executes` the xt in `x`; you can also use `is` and `action-of` on `x`. Example:

```
['] . {: xt: y :}
5 y \ prints 5
['] drop is y
```

Convenient `postponeing`

Instead of writing a long sequence of `postpones`, e.g.,

```
postpone a postpone b postpone c
```

you can write

```
]] a b c [[
```

An implementation of this feature in standard Forth is available at http://theforth.net/package/compat/current-view/macros.fs.

Modifying words

`Set-does> ( xt -- )` is a modern variant of `does>`. It changes the last defined word to first push its body address, and then perform the xt. E.g., instead of

```
: myconst ( n -- )
  create ,
does> ( -- n )
  @ ;
```

you can write

```
: myconst ( n -- )
  create ,
  ['] @ set-does> ;
```

`Set-optimizer ( xt -- )` changes the last defined word `w` such that it `executes` xt whenever `compile,` is called with the xt of `w` as parameter. You can use this to generate better code for `w`. E.g., you can have myconst generate better code:

```
: myconst ( n -- )
  create ,
  ['] @ set-does>
  [: >body @ ]] literal [[ ;] set-optimizer ;
```

2.4 Auxiliary closure words

The following are convenience features. One can eliminate them from code without requiring deep changes, but the code becomes longer and less readable.

Home location conveniences

We can use variable-flavoured locals to create home locations that live until the end of the definition, but for longer lifetimes, allocating home locations of multiple locals is inconvenient: If they are allocated separately, this may cost extra memory and require extra effort on deallocation; if they are allocated at once, we have to get individual home location addresses with address computations or with structure words.

Our current implementation reuses some of the existing code to provide the following convenience for creating home locations:

```
<{: w^ a f^ b :}h a b ;>
```

This creates a home location for cell `a` and float `b` on the heap, and then (between `:}h` and `;>` pushes the addresses on the stack; finally, the `;>` pushes the address of this home location block so that it can be `free`d at the end of the lifetime.

For implementation simplicity reasons, locals from outside cannot be used inside `<{:...;>`, and the locals defined inside cannot be used outside. That's why the addresses of the home locations are passed on the data stack to the outside.

If we did not have `<{:...;>`, one would have to write the following code to replace the code above:

```
0 cell+ faligned float+ allocate throw
dup cell+ faligned over
```

So, while `<{:...;>` is more cumbersome than one would like, it is better than nothing; and it is very simple to implement.

`Postpone` locals

Given a local `x`, `postpone x` is equivalent to `x postpone literal`. This is especially convenient in combination with `]]...[[` (see below).

However, the generated code compiles the value that `x` had when the `postpone` runs, not the value `x` has at run-time, so the following example will produce results that some may not expect:

```
: foo
  7 {: a :} postpone a 8 to a ; immediate
: bar foo ;
bar . \ prints 7
```

Therefore we recommend that one should not apply `postpone` and `to` to the same local. It would be relatively easy to warn of this combination, but, for now, our implementation does not.

Allocation

These are variants of existing memory allocation words that fit the stack effect expected by `:}*` and `:}xt`.

`alloch` (size -- addr) A variant of `allocate` with a different stack effect.

`allocd` (size -- addr) A variant of `allot` with a different stack effect.

3 Closure Usage and Alternatives

This section gives some examples for uses of closures. We also show alternatives that do not use these features (sometimes before, sometimes after the usage examples), so you get a better impression of whether closures provide benefits for the example, and what they are.

In stack effect comments, we use `...` to indicate additional data and/or FP stack items. For a stack effect comment `( ... x y -{}- ... z )`, the number of stack items represented by `...` normally does not change.

3.1 Numerical integration

Higher-order words are words that take an xt and call it an arbitrary number of times.

A classical use of words that take an xt (in other languages, a function) as argument is numerical integration (also known as *quadrature*):

```
numint ( a b xt -- r )
\ with xt ( r1 -- r2 )
```

This approximates $\int_a^b \mathrm{xt}(x)dx$.[6] Now consider the case that we want to compute $\int_a^b 1/x^y dx$ for a given a, b, and y, and want to have a word for this:

```
: integrate-1/x^y ( a b y -- r )
  [{: f: y :}l ( r1 -- r2 ) y fnegate f** ;]
  numint ;
```

So the stack element y is consumed (and stored in the local y during closure construction, and then

[6] A practical word would have one or more additional parameters that influence the computational effort necessary and how close the result is to the actual value of the integral.

used during the repeated calls to the closure performed by `numint`.

Another way in which we might express this computation is:

```
: 1/x^y ( y -- xt )
 [{: f: y :}h ( x -- r ) y fnegate f** ;] ;

( a b y ) 1/x^y dup numint >addr free throw
```

`1/x^y` takes y and produces an xt. The xt takes x and produces the result. This technique of splitting a function with multiple arguments into a sequence of functions, each with one argument is called currying. It allows a more uniform treatment of functions, which is useful in conjunction with higher-order functions, and is therefore common in functional programming.[7]

A difference between these variants is that in the latter the local y lives after the definition returns in which it was defined. Therefore, we used `:}l` in the first variant, but `:}h` (and `>addr` and `free`) in the second.

A Forth-specific alternative is to pass y on the (FP) stack rather than through a local. In order to do that, `numint` has to be modified to have the following stack effect:

```
numint ( ... a b xt -- ... r )
\ with xt ( ... r1 -- ... r2 )
```

I.e., `numint` has to ensure that xt can access the values on the stack represented by `...`. Now we can write:

```
: integrate-1/x^y ( a b y -- r )
  frot frot ( y a b )
  [: ( y x -- y r2 )
    fover fnegate f** ;]
  numint fswap fdrop ;
```

The stack handling takes some getting-used-to. For a single level of higher-order execution, as used here, this is manageable.

If we want something like the currying variant, this could look like this:

```
: 1/x^y ( y x -- y r )
  fover fnegate f** ;

( a b y ) frot frot ' 1/x^y numint
fswap fdrop
```

We don't get a properly curried function here, but instead a function that reads the the extra argument from the (FP) stack without consuming it, the same as the quotation in the other pass-on-the-stack variant.

If you need several functions with such extra arguments in one computation (for both pass-on-the stack variants), the functions have to be written specifically for the concrete usage (e.g., one reads the second and third stack item, while another reads the fourth stack item, etc.), not quite in line with the combinatorial nature of currying.

In any case, it is a good practice to design higher-level words such that the called xts have access to the stack below the parameters: Move the internal stuff of the higher-level word elsewhere (return stack or locals) before `execute`ing xts.

3.2 `Sum-series`

Franck Bensusan posted a number of use cases[8], among them one for writing a word that computes $\sum_{i=1}^{20} 1/i^2$, as an example of computing specific elements of a series.

This can be written as follows, factoring out reusable components, and going all-in with locals:

```
: for ( ... u xt -- ... )
   \ xt ( ... u1 -- ... )
   {: xt: xt :} 1+ 1 ?do i xt loop ;

: sum-series ( ... u xt -- ... r )
   \ xt ( ... u1 -- ... r1 )
   0e {: f^ ra :}
   ra [{: xt: xt ra :}l ( ... u1 -- ... )
         xt ra f@ f+ ra f! ;] for ra f@ ;

20 [: ( u1 -- r )
     dup * 1e s>f f/ ;] sum-series f.
```

In accumulating/reducing words like `sum-series`, we need to update a value in every iteration. In this variant, we update a local. A variant without closures differs in the following definition:

```
: sum-series ( ... u xt -- ... r )
  \ xt ( ... u1 -- ... r1 )
  0e swap [: ( ... xt r1 u1 -- ... xt r2 )
    {: f: r :} swap dup >r execute r> r f+
  ;] for drop ;
```

This puts `r` in a local in the quotation in order to get it out of the way. This is not needed for the particular way we use the word, but it allows to use `sum-series` in other contexts, too. It is the price we pay for being able to use this as a higher-order word without needing closures.

An in-between variant that is better than either variant above is:

[7] Interestingly, working with higher-order and curried functions allows a programming style that avoids local variables; still, general locals are useful in implementing curried functions. There are alternatives, however [Bel87].

[8] news:<8ea09174-ddac-4d5b-b906-df3bd4f07932@googlegroups.com>

```
: sum-series ( ... u xt -- ... r )
  \ xt ( ... u1 -- ... r1 )
  0e [{: xt: xt :}l ( ... u1 r1 -- ... r2 )
        {: f: r :} xt r f+ ;] for ;
```

This passes the xt through the closure mechanism, and the intermediate result on the stack.

3.3 Man or boy?

Knuth's man-or-boy test [Knu64] is an Algol 60 function that has no purpose other than to test whether a compiler implements lexical scoping correctly. In Algol:

```
begin
  real procedure A(k, x1, x2, x3, x4, x5);
  value k; integer k;
  real x1, x2, x3, x4, x5;
  begin
    real procedure B;
    begin k := k - 1;
          B := A := A(k, B, x1, x2, x3, x4)
    end;
    if k <= 0 then A := x4 + x5 else B
  end;
  outreal(A(10, 1, -1, -1, 1, 0))
end;
```

In Forth[9]:

```
: A {: w^ k x1 x2 x3 xt: x4 xt: x5 | w^ B :}
  recursive
  k @ 0<= IF  x4 x5 f+  ELSE
    B k x1 x2 x3 action-of x4
    [{: B k x1 x2 x3 x4 :}L
      -1 k +!
      k @ B @ x1 x2 x3 x4 A ;] dup B !
      execute  THEN ;
10 [: 1e ;] [: -1e ;] 2dup swap [: 0e ;] A
f.
```

This example allocates all locals and all home locations on the locals stack.

Given the purpose of this example, we did not try to find an alternative without closures.

3.4 testr

McCarthy [McC81] presents the following Lisp function (in M-expression syntax) by James R. Slagle, which revealed that the Lisp implementation of the time did not implement lexical scoping:

```
testr[x,p,f,u] <-
  if p[x] then f[x]
  else if atom[x] then u[]
  else testr[cdr[x],p,f,
           lambda:testr[car[x],p,f,u]].
```

[9]Call Gforth with `gforth -l128k`

> The object of the function is to find a subexpression of x satisfying p[x] and return f[x]. If the search is unsuccessful, then the continuation function u[] of no arguments is to be computed and its value returned. ([McC81])

To implement this in Forth, we use the following words for accessing S-Expressions:

atom (s-expr -- f) is the s-expression an atom (true) or a pair (false)?

car (s-expr -- s-expr) the first half of a pair

cdr (s-expr -- s-expr) the second half of a pair

In Forth with closures, the equivalent is:

```
: testr {: x p f u -- s :} recursive
  \ x is an s-expression
  \ p is an xt ( s-expr -- f )
  \ f is an xt ( s-expr1 -- s-expr2 )
  \ u is an xt ( -- s-expr )
  \ s is an s-expression
  x p execute if x f execute exit then
  x atom if u execute exit then
  x cdr p f
  x p f u [{: x p f u :}l
    x car p f u testr ;] testr ;
```

This could also be written using `xt:`, but the number of required `action-of`s would exceed the number of eliminated `execute`s.

The reason for dealing with the unsuccessful search by calling u is that f can return any S-expression, so there is no way to indicate an unsuccessful search through the return value. Of course, in Forth, we have the option of returning such an indication as additional return value, so we can implement `testr` without closures:

```
: testr1 {: x p -- s1 f :} recursive
  x p execute if x true exit then
  x atom if nil false exit then
  x cdr p testr1 dup if exit then
  x car p testr1 ;
```

```
: testr {: x p xt: f xt: u -- s :}
  x p testr1 if f exit then
  drop u ;
```

3.5 Defining words

The `create...does>` feature of Forth has a number of problems:

- It does not allow optimizing read-only accesses to the data stored in the word.

- When multiple cells (or other data) are stored in the word, it becomes hard to follow across the `does>` boundary what is what.
- First `create` produces a word with one behavior, then `does>` changes the behaviour (and this can theoretically happen several times). This causes problems in implementations that compile directly to flash memory.

In the following we focus on the first two problems.

`+field`

The first problem is exemplified by:

```
: +field ( u1 u "name" -- u2 )
   create over , +
does> ( addr1 -- addr2 )
   @ + ;

\ example use
1 cells 1 cells +field x ( addr1 -- addr2 )
: foo x @ ;
```

Using `set-does>`, `+field` is written as:

```
: +field ( u1 u "name" -- u2 )
  \ name execution: ( addr1 -- addr2 )
  create over , +
  [: @ + ;] set-does> ;
```

With the built-in `+field`, VFX compiles `foo` into `MOV EBX, [EBX+04]` (3 bytes). However, with the user-defined definition of `+field` above, this is not possible: the user could change the value in `x` later (e.g., with `0 ' x >body !`), and the behaviour of `foo` has to change accordingly. Therefore, VFX produces a an 8-byte two-instruction sequence instead.

With closures, we can write `+field` as follows:

```
: +field ( u1 u "name" -- u2 )
  \ name execution: ( addr1 -- addr2 )
  create over
  [{: u1 :}d drop u1 + ;] set-does>
  + ;
```

The `drop` is there to get rid of the body address of *name*, which the `set-does>` mechanism (like `does>`) pushes automatically.

In this variant, *u1* is transferred to *name* through the closure mechanism; its value does not change (there is no `to u1`), so the compiler can generate efficient code for `foo`. Currently there is no compiler that does that, but a compiler that inlines the closure when *name* is compiled and that is analytical about locals should not find it difficult.

A way to solve this problem without closures is to define the defining word based on `:` instead of `create`:

```
: +field ( u1 u "name" -- u2 )
  \ name execution: ( addr1 -- addr2 )
  over >r : r> ]] literal + ; [[ + ;
```

With this `+field`, VFX produces the same code for `foo` as with the builtin `+field`. This can be made slightly easier to read by using a local, and `postpone`ing it:

```
: +field ( u1 u "name" -- u2 )
  \ name execution: ( addr1 -- addr2 )
  {: u1 u :} : ]] u1 + ; [[ u1 u + ;
```

Another approach for dealing with the read-only problem is to declare the memory as not-going-to-change after initializing it (supported in iForth):

```
: +field ( u1 u "name" -- u2 )
  create over , +
  here cell- 1 cells const-data
does> ( addr1 -- addr2 )
  @ + ;
```

Yet another approach is to change the intelligent `compile,` to compile fields efficiently:

```
: +field ( u1 u "name" -- u2 )
  \ name execution: ( addr1 -- addr2 )
  create over , +
  [: @ + ;] set-does>
  [: >body @ ]] literal + [[ ;]
  set-optimizer ;
```

This works in Gforth (development version), and, with a different syntax, in VFX. `Set-optimizer` changes the last defined word (i.e., the one defined by `+field1`) so that `compile,`ing it calls the quotation; that first fetches the field offset (at compile time, not at run-time), compiles it as a literal and then compiles the `+`. A disadvantage of this approach is that the optimizer has to implement nearly all of the `does>` part again; and such redundancy can make errors hard to find (e.g., the word works fine when interpreted, but acts up when compiled).

We can use closures instead of the body to pass `u1`:

```
: +field ( u1 u "name" -- u2 )
   create
   over [{: u1 :}d drop u1 + ;] set-does>
   over [{: u1 :}d drop ]] u1 + [[ ;]
   set-optimizer
   + ;
```

This demonstrates the redundancy nicely. A disadvantage of this approach is that the redundancy now also costs memory, because two closures are stored in the dictionary.

Finally, there was a proposal for `const-does>` [Ert00], but it did not generate much interest. The code would look as follows:

```
: +field ( u1 u "name" -- u2 )
  over + swap ( u2 u1 )
1 0 const-does> ( addr1 -- addr2 )
  ( addr1 u1 ) + ;
```

The `1 0` tells `const-does>` to take one data stack item and 0 FP stack items from these stacks when `const-does>` is called, and push them on these stacks when the defined word is performed. The body address of the `create`d word is not pushed, `addr1` is passed by the caller of *name* (typically the base address of the structure containing the field), `u1` by `const-does>`.

`Interface-method`

The `+field` example is easy to understand, but the following, larger example is better for demonstrating the effects. It also demonstrates the second problem of passing several values across the `does>` boundary.

The following is a simplified variant of the word for defining interface method selectors in `objects.fs` [Ert97]:

```
\ fields: object-map selector-offset
\         selector-interface
\ structure (constant): selector

: interface-method ( n-sel n-iface -- )
  create here tuck selector allot
  selector-interface ! selector-offset !
does> ( ... object -- ... )
  2dup selector-interface @
  swap object-map @ + @
  swap selector-offset @ + @ execute ;
```

This example exhibits the read-only and the multiple-cells problem. The latter problem is attacked by organising these cells as a struct, storing into it in the `create` part, and reading from it in the `does>` part, but compared to the following closure-using variant, the code is still relatively complicated.

```
: interface-method ( n-sel n-iface -- )
  create [{: n-sel n-iface :}d
    drop dup object-map @ n-iface + @
    n-sel + @ execute ;] set-does> ;
```

This locals-using variant eliminates all the complications of storing the parameters in the `create` part. The `does>` part is also quite a bit simpler, as it avoids having to juggle the address of the `create`d word.

The `:`-using definition looks as follows:

```
: interface-method {: n-sel n-iface -- :}
  : ]] dup object-map @
  [[ n-iface ]] literal + @
  [[ n-sel   ]] literal + @
  execute ; [[ ;
```

The resulting code (produced by VFX 4.72) for a call to a word defined with `interface-method` is:

```
 does> version           : version
MOV  EDX, 0 [EBX]       MOV  EDX, 0 [EBX]
ADD  EDX, [080C0BB4]    MOV  EDX, [EDX+04]
MOV  ECX, [080C0BB0]    CALL [EDX+04]
ADD  ECX, 0 [EDX]
CALL 0 [ECX]
```

If we can postpone locals, or, in this case, use them inside `]]`...`[[`, this can be further shortened into:

```
: interface-method {: n-sel n-iface -- :}
  : ]] dup object-map @ n-iface + @
       n-sel + @ execute ; [[ ;
```

The code between `]]` and `[[` is almost the same as the code in the closure in the closure version.

4 Implementation

This section describes our implementation of the features described in this paper.[10] Other implementations are possible, but are not discussed here, with one exception: Gforth uses a locals stack, and we always mention the locals stack here; but adapting the implementation for a system where the return stack serves as locals stack is not difficult.

4.1 Closures and execution tokens

The execution token for a closure represents not just the code, but also the passed locals. Yet it has to fit into a single cell.

Our implementation deals with that by a variant of the trampolines used by gcc for the same purpose: A block of memory is allocated; the start of this block contains the header of an anonymous word, and the rest contains the values of the locals defined at the start of the closure. The closure is represented by the xt of the anonymous word.

When the closure is performed, it copies the values of the locals to the locals stack. This means that the closure locals can be treated like ordinary locals in the rest of the definition. After this copying, the user-defined code of the closure is performed.

[10] `http://git.savannah.gnu.org/cgit/gforth.git/tree/closures.fs`

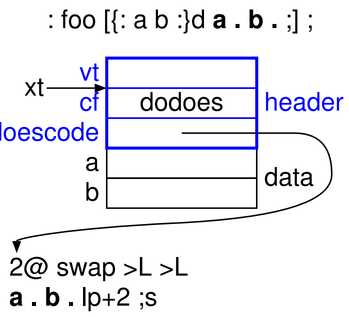


Figure 1: A definition containing a closure, and the memory representation of a closure created by invoking the definition; the part of the doescode before the bold part is generated by the compiler to copy the data of the closure to the locals stack.

In finer detail, a closure is an anonymous `create`...`does>` defined word (but it can reside not just in the dictionary, but alternatively on the locals stack or on the heap), where the code after the `does>` starts by copying the data from the body of the word to the locals stack, followed by the user-written code. Figure 1 shows an example.

The compile-time part of the closure implementation is deeply intertwined with the pre-existing implementations of locals and quotations in Gforth, and a detailed description will probably be of little benefit to implementors of other systems, but we still mention some interesting aspects: During closure construction, the locals stack pointer points to the memory for the closure (i.e., not always in the locals stack). The closure locals are arranged in the closure memory just as normal locals are arranged on the locals stack, they get the same offsets (using the same code as during ordinary locals definition), and after copying behave just as normal locals.

This part costs 78 source lines of code (SLOC, blank, and comment-only lines not counted).

4.2 Home locations

The home location syntax `<{:...;>` is based on the closure implementation: Creating a home location block differs from creating a closure by not producing a word header, and by letting the locals stack pointer point to the home location block until `;>`.

This part costs 6 SLOC.

4.3 Postpone locals

`Postpone`ing locals is implemented by special-casing locals in `postpone`.

This part costs 25 SLOC. It is so large because each of the nine locals definers needs a special case.

5 Lexical scoping and flat-closure conversion

The closure syntax presented above was originally designed for minimal implementation complexity, even at the expense of programmer inconvenience. However, looking at the examples, we now think that it is very appropriate for stack-based languages like Forth: For languages where the primary data location is the stack(s) rather than locals, it is appropriate to build closures from data on stacks rather than by copying existing locals.

However, this means that when we want to convert code from languages with lexical scoping to Forth, we have to perform flat-closure conversion manually. This section sketches how to do that, in a mechanical way. Alternatively, we can find a way to express the same purpose differently, as in the *testr* example (Section 3.4), but there is no mechanical way to do that, and no guarantee that there is such a way.

This also demonstrates that our closure syntax is as powerful as lexical scoping in Algol-family languages. There is no mechanical process for converting the automatic memory reclamation of, e.g., Scheme to manual memory reclamation. If we stick with mechanical conversion for that part, we either have to live with leaking memory, or use some kind of garbage collection for the closures.

We use the following contrived program with lexical scoping as a running example.

```
: foo ... {: a b :} ...
  [: ... {: c :} ... to a ... b ...
     [: ... to b ... c ... ;] ... ;] ... ;
```

This program only contains definitions and uses of locals, and quotations. Other operations can be inserted in the places marked with `...`, but do not play a role in flat-closure conversion.

The first step is to perform *assignment conversion* [Dyb87, Section 4.5]: For every local that is accessed with `to` and also accessed in a quotation where it is not defined, we convert it into a home location and access it with `@` and `!`:

```
: foo ... <{: w^ a w^ b :}d a b ;>
  drop {: a b :} ...
  [: ... {: c :} ... a ! ... b @ ...
     [: ... b ! ... c ... ;] ... ;] ... ;
```

In this example, we allocated the home locations in the dictionary, and then `drop`ped the address containing the home location block. Note that you need to use `w^` only when defining the home location; in the rest of the code, the addresses are passed around as values, so value-flavoured locals are fine there.

The next step is the actual flat-closure conversion: You have to mention all locals accessed inside the closure in the locals definition at the start of the closure, and pass them to the closure on the stack:

```
: foo ... <{: w^ a w^ b :}d a b ;>
  drop {: a b :} ...
  a b [{: a b :}d ... {: c :}
    ... a ! ... b @ ...
    b c [{: b c :}d ... b ! ... c ... ;]
    ... ;] ... ;
```

5.1 Alternative syntaxes and implementations

What if we tried to go for a syntax that supports lexical scoping directly instead of through manual flat-closure conversion? How much implementation complexity would that cost, and would the benefit be worth the cost? Are there intermediate approaches?

The first step towards lexical scoping is that closures get the values of the closure locals (those defined at the start of the closure) from same-named locals of the enclosing definition or closure, rather than from the stacks. This is relatively easy to implement, but it requires that the value is in a local. As the examples show, this requirement would often result in extra locals definitions, so implementing that is not necessarily an advantage.

The next step would be to completely hide the actual flat-closure conversion: The compiler would have to look at the whole code of the definition, and note which of the locals are used inside which quotation, and then convert the quotations into closures by itself. While that is not particularly hard, it requires looking at the whole definition at once, which would require a major rewrite for most Forth systems. The benefit would be that the code for `foo` shown after the assignment conversion step would work (with some adjustments for manual memory reclamation of closures).

Similarly, the assignment conversion step can be split into two steps:

In the first step, the programmer marks some locals on definition as requiring assignment conversion (with special local definers, e.g., `w!`). The compiler would then allocate a home location for these locals automatically, pass the address around, and automatically convert read accesses to fetches from the address, and `to` accesses to stores to the address. A `to` access to a local that is not marked as requiring assignment conversion produces an error if the local occurs in a quotation where the local was not originally defined. This step would require some work, but no deep changes to the usual compilers. The benefit would be that the programmer would avoid nearly all of the assignment conversion work, and only needs to mark some locals as requiring assignment conversion.

In the second step, the compiler collects the information about the locals requiring assignment conversion by itself, relieving the programmer of that duty. Again, it requires looking at the whole definition at once, but otherwise would not be a lot of work.

6 Performance

This section presents performance results from microbenchmarks on the current implementation in Gforth. Note that microbenchmarks have their pitfalls and in application usage effects may dominate that are not reflected in these microbenchmarks. Moreover, the current implementation has seen only minimal performance work (costing 4 source lines), and some of these benchmarks might see substantial speedups by investing more work in performance.

We have two kinds of microbenchmarks: Creating a closure (or an alternative to a closure), and running a closure (or an alternative). The closure we use is:

```
[{: x :}l x + ;]
```

Running a closure with one or two cells as above profits from the little performance work we have applied, so for the *run closure* benchmark we also measure a three-cell variant that exercises the general case:

```
[{: x y z :}l x + ;]
```

We use the following variants:

closure For creation, we measure the three different allocation methods (locals stack, dictionary, heap), with the heap variant including the `free` overhead.

does Create an anonymous `create`d word with `x` in its body, with `[: @ + ;] set-does>`.

:noname create an anonymous `:noname` word which compiles `x` as literal in its body.

stack Use a quotation that uses `x` from the stack (without consuming it): `[: over + ;]`. Benchmarking its creation just means benchmarking pushing the xt.

We run the benchmark on a 4GHz Core i5-6600K (Skylake). We use 50,000,000 iterations for each microbenchmark, but report the cycles and instructions per iteration, subtracting the loop overhead. The results are:

cycles	inst.	per iteration
21.0	99.0	create closure local
62.9	183.5	create closure dictionary
113.6	459.0	create closure heap
735.1	2464.7	create does
5115.4	15159.5	create :noname
8.0	14.0	create stack
7.0	43.0	run closure 1 cell
21.3	85.0	run closure 3 cells
6.0	38.0	run does
6.2	27.0	run :noname
7.1	33.0	run stack

Note that results of 8 cycles or less in these microbenchmarks are usually dominated by dependency chains through instruction or stack pointers, and the relative performance may be different (maybe more like the relations of instructions counts) in applications.[11]

Still, there are some conclusions we can make:

Creating a local closure is relatively cheap, whereas creating heap and dictionary closures is quite a bit more expensive. Dynamically creating `create...does>` words instead is a lot more expensive, and the same with `:noname` is even more expensive. Pushing the xt of a quotation is cheap, as expected.

Running a closure with one cell is slightly more expensive than the other variants; the general case (3 cells) is quite a bit more expensive, but could be optimized, too; there will be very few cases where the number of runs/creation is so high that the does and :noname variants break even. The stack variant is cheap in both creation and run time.

7 Related work

Already Lisp [McC81] and Algol 60 allowed nested functions and accessing outer locals, but with limitations: Lisp initially used dynamic scoping; this was considered a bug by McCarthy (Lisp's creator) [McC81] (see Section 3.4), but that bug had entrenched itself as a feature in the meantime, and the Lisp family took a while to acquire lexical scoping (prominently in Scheme and Common Lisp). A reason for that is that Lisp allows returning functions, which in combination with lexical scoping creates the *upwards funarg* problem: local variables no longer always have lifetimes that allow to use a stack for memory management.

Algol 60 avoided the upwards funarg problem by not allowing to return functions. Still, lexical scoping (in combination with call-by-name) proved a challenge to implement, as can be seen by Knuth's man-or-boy test [Knu64] (see Section 3.3), which revealed that many Algol compilers failed to implement access to outer locals correctly.

The best-known ways to implement the access to outer locals are static link chains and the display [FL88]. They keep each local in only one place, and have relatively complex and sometimes slow ways to access them.

By contrast, in this paper we use the *flat-closure coversion* approach [Dyb87, Section 4.4] in combination with assignment conversion [Dyb87, Section 4.5], which replicates locals (or their addresses) in order to make the access cheap. Moreover, in typical Forth style, we only provide flat closures and home location support, and leave it to the programmer to perform assignment and flat-closure conversion manually. This makes the programmer responsible for optimizations in the conversion process [KHD12], and avoids the need to put values into locals in order to get them into closures.

Concerning memory management, most languages have chosen one of two approaches: 1) restrict function-passing or outer-locals access such that stack management is sufficient; or 2) don't have restrictions, and use garbage collection for the involved data structures when necessary.

After decades of growth in the functional programming community, using higher-order functions and passing functions to them has recently made the jump to mainstream languages like C++ (in C++11), Java (in Java 8), and C#. This feature is typically called *lambda*. The C++ variant[12] is extremely featureful, and, while too complex for Forth, inspires ideas on how such features can be implemented in close-to-the-metal languages.

Moving closer to Forth, Joy [vT01] is a stack-based functional language. It uses the term "quotation" for a nameless word that can be defined inside other words. Joy has no locals, so quotations in it cannot access outer locals.

Factor [PEG10] is a high-level general-purpose language with roots in Forth and Joy; it has quotations and locals, and allows access to outer locals.

Lynas and Stoddart [LS06] added lambda expressions with read-only lexical scoping to RVM-Forth. They implemented accesses to outer variables by compiling them as literals with placeholder values; when generating the xt, the code is copied, and the actual values of the outer variables are plugged into the code instead of the placeholder values. These code copies are not freed in forward execution.

Gerry Jackson implemented quotations with full lexical scoping and explicit deallocation of closures in Forth-94.[13] He managed to implement

[11] You may also wonder about the impossible apparent instructions per cycle (IPC) for some of the benchmarks, but note that you have to add the loop overhead (12 instructions in 6 cycles) to compute the actual IPC.

[12] `https://en.cppreference.com/w/cpp/language/lambda`

[13] news:<6b5eead4-f809-4dd4-81c6-16e1c2a9f613@q14g2000vbn.googlegroups.com>, `http://qlikz.org/forth/archive/lambda.zip`

all this functionality (but with some limitations) and workarounds for the limitations of Forth-94 in 312 SLOC (including an object-oriented package).

In contrast to these works, the present work abandons lexical scoping in favour of reducing the implementation effort, putting the onus of assignment and closure conversion on the programmer.

In 2017 the Forth200x committee has accepted a proposal[14] for quotations that does not standardize the access to outer locals, leaving it up to systems whether and how they implement accesses to outer locals.

Of course, in classical Forth fashion, some users explored the idea of what outer-locals accesses can be performed with minimal effort. In particular, Usenet user "humptydumpty" introduced rquotations[15], a simple quotation-like implementation that uses return-address manipulation. The Forth system does not know about these rquotations and therefore treats any locals accessed inside rquotations as if they were accessed outside. In the case of Gforth (as currently implemented) this works as long as the locals stack is not changed in the meantime; e.g., the higher-order word that calls the rquotation must not use locals.

There is no easy way to see whether this restriction has been met; this is also classical Forth style, but definitely not user-friendly. Static analysis could be used to find out in many cases whether the restriction has been met, but that would probably require more effort than implementing the approach presented in this paper, while not providing as much functionality.

8 Conclusion

Locals in standard Forth have a number of restrictions. In this paper we mainly looked at the restriction that, in a quotation, one can only access locals that have been defined in that quotation. But instead of adding the capability to access outer locals, we reduced it to the basic need to initialize locals of a quotation/closure from outside data, and presented syntax and an implementation of stack-initialized flat closures with explicit memory management. In addition, we present conveniences for defining home locations for writable locals, and for `postpone`ing (read-only) locals.

We presented a number of examples where these features allow additional, and sometimes shorter and easier-to-read ways to express the functionality. We also presented alternative code that does not use these features.

[14]`http://www.forth200x.org/quotations.txt`

[15]news:<f71bfb01-4b8e-49d6-abd5-12bda6dbfcd2@googlegroups.com>

In these examples, the features provide some benefits. The implementation of flat closures alone costs 78 source lines in Gforth, or 109 source lines for all the features combined. Whether the benefits are worth this implementation effort will have to be seen.

Acknowledgments

The anonymous referees, Marcel Hendrix, Gerry Jackson, and Bill Stoddart provided valuable feedback on earlier versions of this paper.

References

[Bel87] Johan G.F. Belinfante. S/K/ID: Combinators in Forth. *Journal of Forth Application and Research*, 4(4):555–580, 1987. 7

[Dyb87] R. Kent Dybvig. *Three Implementation Models for Scheme*. PhD thesis, University of North Carolina at Chapel Hill, April 1987. 5, 7

[Ert94] M. Anton Ertl. Automatic scoping of local variables. In *EuroForth '94 Conference Proceedings*, pages 31–37, Winchester, UK, 1994. 2.3

[Ert97] M. Anton Ertl. Yet another Forth objects package. *Forth Dimensions*, 19(2):37–43, 1997. 3.5

[Ert00] M. Anton Ertl. `CONST-DOES>`. In *EuroForth 2000 Conference Proceedings*, Prestbury, UK, 2000. 3.5

[Ert14] M. Anton Ertl. Region-based memory allocation in Forth. In *30th EuroForth Conference*, pages 45–49, 2014. 2.1

[FL88] Charles N. Fischer and Richard J. LeBlanc. *Crafting a Compiler*. Benjamin/Cummings, Menlo Park, CA, 1988. 7

[KHD12] Andrew W. Keep, Alex Hearn, and R. Kent Dybvig. Optimizing closures in O(0) time. In Olivier Danvy, editor, *Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming, Scheme 2012, Copenhagen, Denmark, September 9-15, 2012*, pages 30–35. ACM, 2012. 7

[Knu64] Donald Knuth. Man or boy? *Algol Bulletin*, page 7, July 1964. 3.3, 7

[LS06] Angel Robert Lynas and Bill Stoddart. Adding Lambda expressions to Forth. In *22nd EuroForth Conference*, pages 27–39, 2006. 7

[McC81] John McCarthy. History of LISP. In Richard L. Wexelblatt, editor, *History of Programming Languages*, pages 173–197. Academic Press, 1981. 3.4, 7

[PEG10] Sviatoslav Pestov, Daniel Ehrenberg, and Joe Groff. Factor: a dynamic stack-based programming language. In William D. Clinger, editor, *Proceedings of the 6th Symposium on Dynamic Languages, DLS 2010, October 18, 2010, Reno, Nevada, USA*, pages 43–58. ACM, 2010. 7

[vT01] Manfred von Thun. Joy: Forth's functional cousin. In *EuroForth 2001 Conference Proceedings*, 2001. 7

A Sudoku

This appendix shows another example. It demonstrates the use of an xt-passing style in a larger application. The shown code is complex, and we do not expect you to understand it completely. But you can try to follow the stack flow in Fig. 3 and 4 to get an impression of the benefits and drawbacks of these two approaches, and also skim Fig. 5 to get an impression of that alternative.

In 2006, I (Ertl) wrote a Sudoku program.[16] In Sudoku the same constraints apply to rows and columns, and squares have a related constraint, so I tried to find a good factoring.

At one point[17] I factored out horizontal and vertical walks (of the fields in a row/column, or of the columns/rows of the whole Sudoku) into higher-order words `map-row` and `map-col` (see Fig. 2). I passed the extra parameters to the words called by these words through the stack. You can see these higher-order words in action in Fig. 3.

However, I found it hard to track the stack contents, because the words are not called in the order in which they appear in the code. Therefore I also found it hard to write and maintain this code, even though I used locals to make it a little less opaque. Soon after I switched to a different approach.

But before we look into that approach, let's consider how things would look with closures: Fig. 4. The code is shorter, but, what's more, it is much easier to see the data flow: Instead of following how the data items flow through the higher-order words to the `execute`d xts, the xts (produced from closures) have simple stack effects such as `( var -- )`.

[16]https://github.com/AntonErtl/sudoku

[17]https://github.com/AntonErtl/sudoku/blob/da19285814c49a007dd8d954cf94a29f51fa51a1/sudoku3.fs

These xts do not use extra parameters;[18] instead, the data is passed through the closure mechanism. Note that there are two levels of closures, and accesses to data that originally came from one or two levels out.

The approach I actually switched to was quite different, though: Following advice from Andrew Haley, I created macros `do-row loop-row do-col loop-col` for performing the walks, and wrote `gen-row-constraints gen-col-constraints` and other words using these macros (Fig. 5). The result[19] feels more Forth-like and has seven lines less than the stack-using one (once we eliminate two now-unused words), but increases the dictionary size (including threaded code, excluding native code) on 64-bit Gforth 0.7.9_20180830 from 9168/9248 bytes (for xt-passing/closures) to 11544 bytes (the macros generate quite a bit of code each time they are used).

[18]The `var` in `( var -- )` is produced by the higher-order words that call the xt.

[19]https://github.com/AntonErtl/sudoku/blob/dc0f80bbbed8a7c488af7aecb5de0b7d5c5662ac/sudoku3.fs

```
\ gen-valconstraint ( var container xt -- )
\ check ( -- )
\ map-row ( ... row xt -- ... ) apply xt ( ... var -- ... ) to all variables of a row
\ map-col ( ... col xt -- ... ) apply xt ( ... var -- ... ) to all variables of a col
\ row-constraint ( var row -- )
\ col-constraint ( var col -- )
```

Figure 2: Helper words for Sudoku

```
: gen-valconstraint1 { xt container var -- xt container }
    var container xt gen-valconstraint
    xt container
    check ;
: gen-contconstraint { xt-map xt-constraint container -- xt-map xt-constraint }
    xt-map xt-constraint container dup ['] gen-valconstraint1 xt-map execute drop ;
: gen-row-constraints ( -- )
    check ['] map-row ['] row-constraint grid @ ['] gen-contconstraint map-col 2drop ;
: gen-col-constraints ( -- )
    check ['] map-col ['] col-constraint grid @ ['] gen-contconstraint map-row 2drop ;
```

Figure 3: Part of Sudoku program with higher-order words using the stack

```
: gen-contconstraint1 ( xt-map xt-constraint -- xt-contconstraint )
  [{: xt: map xt-constraint :}d ( container -- )
    xt-constraint over [{: xt-constraint container :}l ( var -- )
      container xt-constraint gen-valconstraint check ;] map ;] ;
: gen-row-constraints ( -- )
  check grid @ ['] map-row ['] row-constraint gen-contconstraint1 map-col  ;
: gen-col-constraints ( -- )
  check grid @ ['] map-col ['] col-constraint gen-contconstraint1 map-row ;
```

Figure 4: Part of Sudoku program with closures

```
\ replace MAP-ROW and MAP-COL with
\ do-row ( compilation: -- do-sys; run-time: row -- row-elem R: row-elem )
\ loop-row ( compilation: -- do-sys; run-time: R: row-elem -- )
\ do-col ( compilation: -- do-sys; run-time: col -- col-elem R: col-elem )
\ loop-col ( compilation: -- do-sys; run-time: R: col-elem -- )
: gen-row-constraints ( -- )
    check grid @ do-col
        dup do-row
            over ['] row-constraint gen-valconstraint check loop-row
        drop loop-col ;
: gen-col-constraints ( -- )
    check grid @ do-row
        dup do-col
            over ['] col-constraint gen-valconstraint check loop-col
        drop loop-row ;
```

Figure 5: Part of Sudoku program with macros

Method dispatch in Oforth

M. Franck Bensusan
http://www.oforth.com

Abstract

Oforth is a Forth dialect that implements Object Oriented Programming as a built-in mechanism. For methods, it provides a full dynamic binding : two classes that are unrelated (ie Object is their common parent) can implement methods with the same name and the method to execute is resolved at runtime. Furthermore, classes are never "closed" and it is possible to extend a class with new methods at any moment.
As many core words are implemented as methods, method dispatch must be as fast as possible, while, if possible, limiting the memory used.
This paper discusses the implementation of method dispatch in Oforth : classic virtual tables are used to cache code addresses but they are allocated and constructed at runtime, while methods are executed. This is done without suffering much performance penalties.

1 Introduction

Oforth is a Forth dialect that implements a full OOP model. Many core word, like #+, #-, ... are implemented as methods so method dispatch must be as fast as possible. There are two more constraints to be addressed : dynamic binding and non-closed classes. "Dynamic binding" means that all classes can implement all methods, whatever their position in the hierarchy and the selection of the method to run will occur at runtime, according to the top of stack. "Non-closed" classes means that we can always add a new methods to an existing class. For instance, we can create the Integer class, then create the Float class, then add the “>float” method to the Integer class.

With theses constraints, it is not possible to create, at compile time, a definitive virtual table for each class with a pointer to this VT stored in each object. We have to adjust the virtual tables at runtime.

In this paper, we look at the syntax of messages, class definition and method definitions (2), the dispatch message mechanism implemented (3), optimizations that occur at compile time (4), some discussion about the performances and memory cost (5), and some discussions for future work (6).

There have been many works on method dispatch in the general programming language literature ([DUC11] for instance) and some work in the Forth community ([RP96], [ERT12]). This paper is not intended to expose new ideas on this subject : its objective is to expose the dispatch method used in Oforth and what choices have led to this implementation.

2 Messages, classes and methods

Messages are represented by words created in the dictionary. They can be "ticked", executed, ... as classic words. You will almost never create a new message without its first method, but, if necessary (forward definition for instance), you can do it using :

```
message: foo
```

A class is also a word in the dictionary. It is created by sending the #new: message to the Class class (a meta-class) :

```
Object Class new: A
```

This creates a new word, A, in the dictionary with Object as its parent. Oforth only supports single-inheritance. Using A word will push the class on the stack.

Once a class is created, methods can be added :

```
A Class new: A1

A1 method: foo
   "Foo for A1 :" . self . ;

Object Class new: B

B method: foo
   "Foo for B :" . self . ;

A method: bar
   "Bar for A :" . self . ;

A virtual: foo2
   "to be redefined" abort ;

A1 method: foo2
    "Redefined: " . self . ;

#bar .s
[1] (Message) #bar
```

If messages (here words foo, bar and foo2) were not created yet, they are created when the first method corresponding to the message is created.

All methods call (whether they are virtual or not) have dynamic binding, according to object on top of stack. Calling a method is just like calling a word, but the object that will receive the message have to be pushed on the stack first. One important rule is that this TOS is removed from the stack when calling the method, and stored on the return stack. In order to push this TOS (called the method receiver) on the stack in the method's body, the self word can be used. For instance, this is how the previous words are called on objects :

```
A1 new foo
Foo for A1 : aA1 ok

B new foo
Foo for A2 : aB ok

A new dup bar foo2
Bar for A : aA [console:1] #Exception : to
be redefined

A1 new dup bar foo2
Bar for A : aA1 Redefined:  aA1 ok
```

Methods can't be redefined into subclasses unless they are declared as virtual (here foo2, for instance). Non virtual methods correspond to final methods in Java : they can't be redefined in subclasses. Declaring a method as virtual can have impact on optimizations during compilation (see chapter 4).

Ticking a word is done using the # word and not ' (' is dedicated to characters). No space is needed between # and the name. So #bar will push the word bar (here a message) on the stack, or compile a literal into the current definition when compiling.

There is no word such as "end-class". The #bar method is added to A after A1 and B are declared. This allows to extend a class whenever we want, but this also adds constraints on the dispatch mechanism as the list of messages a particular class can respond to is never fixed once for all.

Furthermore, many core words are implemented as methods. The number of messages a class may respond to can be very important and this also adds constraints to the dispatch performances.

3 Dispatch mechanism

3.1 Object's tag field

Instead of associating an index with each message, Oforth uses an "orthogonal" mechanism : an index is associated with each class. In the first slot of each object, a tag is stored, which includes its class index (attributes are stored after this field). On 32bits systems, the class index is present in the 12 least-significant bits of the tag field :

```
0xnnnnnIII
```

Here, the class index value is III. By the way, this means that, on a 32bits Oforth systems, we can't declare more than 4095 classes (the Object class index is 1).

Other information in the tag field is not used for the dispatch mechanism and is not discussed here. Figure 1 shows the tag field stored into each object.

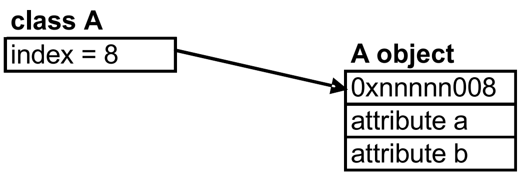


Figure 1 : tag field

3.2 One virtual table by message : the MVT.

As indexes are associated with each class, messages hold the virtual tables : the Message Virtual Table (MVT). Each slot of the MVT contains the address of the method's code to execute for the class corresponding to the index slot. The first slot of a MVT (index 0) holds its size.

When a message is created, it points to an empty virtual table (a static slot with value 0).

Figure 2 shows a MVT for message foo. At index 8, we find the code address of the method to be executed for objects of class A.

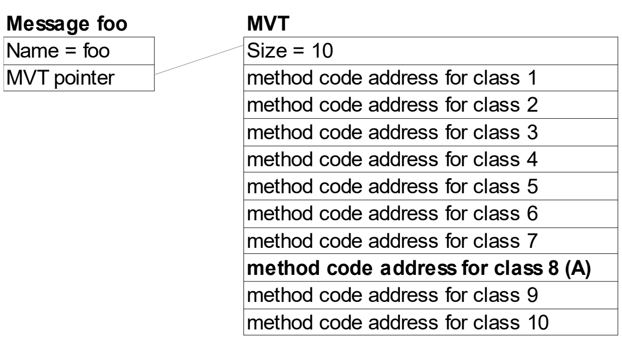

Message foo
Name = foo
MVT pointer
MVT
Size = 10
method code address for class 1
method code address for class 2
method code address for class 3
method code address for class 4
method code address for class 5
method code address for class 6
method code address for class 7
method code address for class 8 (A)
method code address for class 9
method code address for class 10

Figure 2 : a Message Virtual Table

3.3 Sending a message

As it is not possible to populate the MVT when classes are declared, everything must be handled at runtime, when messages are sent.

Listing below is the assembler code (x86 32bits) that is executed to send a message (in register r1). The method to execute is retrieved according to the Top Of Stack (TOS) class :

```
func(runMessage)
   test $1, TOS                (1)
   jne LcallMethodInteger      (1)
   test TOS, TOS               (1)
   je  LcallMethodNull         (1)

   movl (TOS), r0              (2)
   andl 0x00000FFF, r0         (2)

   cmpl IDClass, r0            (3)
   je LcallMethodClass         (3)

   movl virtualTable(r1), r2   (4)

   cmp r0, *r2                 (5)
   jl reallocMVT               (5)

   movl (r2, r0, 4), r3        (6)
   jmp *r3                     (6)
```

Registers used are macros to map CPU registers. For x86 CPU, register allocation is :

```
#define r0          %eax
#define TOS         %ebx
#define r1          %ebp
#define r2          %ecx
#define r3          %edx
```

`virtualTable(r1) is the field offset of the MVT pointer in the message objects.`

Steps that occur during the runMessage function are :

(1) If TOS is a primitive integer or null, TOS is the value itself (and not a pointer) and we can't retrieve the tag field value from those objects. Class index (r0) is set manually before going to (step 4)

(2) Otherwise, we retrieve the class index (in r0) from TOS tag field.

(3) If it is the index of Class meta-class, we have to search for a class method to execute and the dispatch is done using another mechanism (see 3.6).

(4) The MVT associated with the message is retrieved (in r2)

(5) The MVT size is checked. If the size is smaller than the TOS class index, the MVT is reallocated (see 3.4).

(6) An indirect jump to the MVT slot value corresponding to the class index is performed.

3.4 MVT dynamic setting and reallocation

When a message is created, it points to an empty static MVT of size 0 (one static slot with 0 value). So no memory is consumed until the message is actually performed.

When the message is performed, if the MVT size is smaller than TOS class index (this will always be the case if the MVT is the static empty MVT), a new MVT of greater size is allocated and all its slots are populated with the address of a function named "polymorphic", then we go back to the dispatch mechanism. At this point, the MVT is larger enough and we can retrieve the value of the slot corresponding to the class index and run the "polymorphic" function. The purpose of this function is to retrieve the code address to be executed for class r0 and to adjust the slot value with thus value. This is done only once and, the next time, the slot will hold this calculated address code of the method to run and will jump directly to this address.

Figure 3 shows the MVT for foo message just after its first execution for class A. If it is executed again for class A, the method code is now performed. If it is performed for another class (index 5 for instance), the "polymorphic" function will update the slot 5 with the address of method code to run for class “5”.

The same code (step 6 in code 3.4) is used to adjust the MVT slots values (when their value is "polymorphic") and to run the method code (when their value is the method code address).

Also, as the MVT pointer is always accessed when a polymorphic call is performed for a message (step 4), we can extend classes by adding new methods without needing to adjust objects already created.

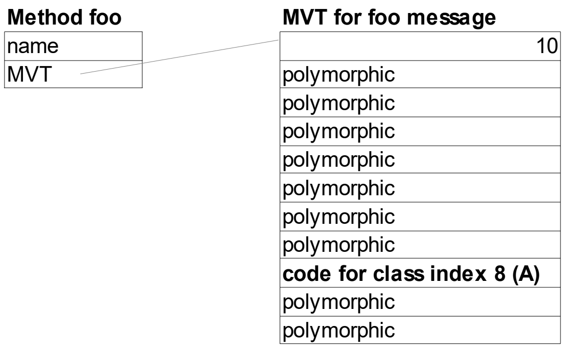


Figure 3 : MVT after executing foo for class A

3.5 The "polymorphic" function

The "polymorphic" function job is to retrieve the code address to execute for the message and TOS, and store its value in the message's MVT.

Each message in the dictionary has a linked list of all the methods declared and each method has two attributes : the class and the code address. A message is a word (with a name), but a method is not a word.

The "polymorphic" function starts with the TOS class and tries to retrieve into the linked list a method for this class. If not found, it retrieves the superclass of this class and searches again until it finds a method or it reaches null (null is the superclass of Object).
The algorithm is actually a little more complex, as it takes into account Properties (at each level, the search is done for the class and its properties).

If a method is found, the MVT slot is updated with its code address. Otherwise (ie the superclass null is reached), the virtual #doesNotUnderstand message is executed for TOS (default behavior, at Object level, is to raise a "does not understand" exception, but it can be redefined for a particular class).

3.6 Dispatch for class methods

For class methods, the dispatch mechanism is different. The correct implementation is also retrieved at runtime but without MVT : each time, the search is done through the hierarchy to retrieve the correct code to run : each class is searched one by one in the order of the inheritance until a method is found. This (slow) dispatch search has been chosen as it will not be used a lot because of optimizations that occur during compilation (see next chapter) : class methods call will mostly be optimized. This will save memory, as no MVT is allocated for class methods.

4 Optimizations during compilation

Those optimizations are handled by the #compile method implemented for messages words. If an optimization is possible, the polymorphic call is reduced to a procedure call.

4.1 When TOS is self

When the last word compiled is self, the receiver will be pushed on the stack. For instance :

```
A method: foo
   self bar ;
```

In this case, we know the type of the TOS object when #bar is performed (here A or one of its subclasses). So, at compile time, we search for a method to run. If we find a non-virtual method, it is the one that will be performed at runtime, so we can optimize by compiling a direct call to this method's code.

If the method found is virtual, no optimization is possible.

4.2 When TOS is a literal

When the last instruction is a push of a literal on the stack (Integer, Float, String, Word, ...), we also know the type of TOS at compile time and we can optimize by compiling a direct call.

A literal can be word, including a class. So this optimization will often apply when performing a class method :

```
: test
   120 Array newSize ;
```

Here the call to message #newSize will be optimized as we know that TOS will be the Array class. That is why there is no MVT allocated for class methods (see 3.6 dispatch for class methods).

4.3 When the message is declared for Object

If the message to compile is declared at the Object level and is not virtual, a direct call is compiled. It is the case for messages like #apply, #detect, #include?, ...

4.4 Otherwise...

If the type of TOS can't be detected at compile time and no optimization is possible, a polymorphic call is generated by calling the runMessage described before. Using the message word (its "name token"), the code generated in the current definition is :

```
   movl message, r1
   call runMessage
```

5 Performances and memory used

5.1 Performances considerations

When a polymorphic call is performed, 4 memory accesses and one indirect jump are executed :
- Access to the tag field of TOS (step 2).
- Access to the message's MVT (step 4).
- Access to the MVT's size (step 5).
- Access to the code to run and jump (step 6)

Memory access for step 2 and 6 are mandatory : we need to access the object to retrieve an information about its type, and we need to retrieve the MVT slot value. Furthermore, on modern processors, access to the field tag probably cache the object's attributes that can be accessed in the method.

Memory access for steps 4 and 5 are not strictly necessary but needed if we want to re-allocate the MVT at runtime and have extendable classes. On modern processors, (5) may cache the access to method's code address (6).

Two other mechanisms are used to optimize performances :

1) A static MVT of size 0 is associated with the newly created message. So there is no need to check if the MVT is null : we directly check if the MVT size is greater than the index (step 5).

2) Polymorphism is calculated by a function ("polymorphic") whose code address initializes the MVT slots. So the same jump to the slot address (step 6) allows to calculate the method to run (the first time) and to directly run the method code (the next times). There is no need to check if the slot value is empty or not.

5.2 Memory used

Main objective for the dispatch mechanism is performances, but it allows to save some memory compared to a "n classes x m messages" matrix :

1) Newly created messages don't allocate virtual table. A MVT is created only if the message is sent at runtime. This is important as many core words are messages and not all declared messages are used at runtime.

2) MVT size are calculated at runtime and won't be greater than necessary (the max index of the class that receives the message).

3) MVT are "by message" and not "by class", so there are many “small” MVT instead of few big virtual tables (one by class).

4) There is no MVT for class methods so no memory is allocated.

Nevertheless, there may still have lot of unused slots in a MVT. It could be interesting to implement other mechanisms (hold also a minimum index or hash MVT, ...). Those mechanisms have not been implemented yet.

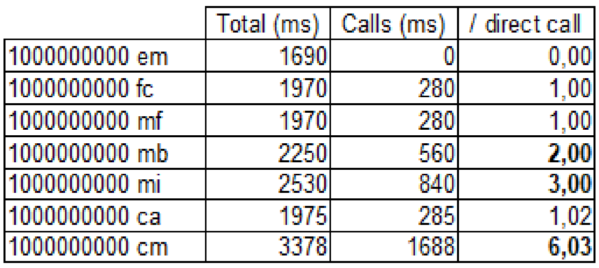
5.3 Benchmarks

The following (simplistic) benchmark tests the various cases. On modern processors, everything will be in cache (particularly the message and its MVT) and branch prediction will apply.

Tests have been run on a core i7-4720 HQ 2,6Ghz on Windows 10.

```
: f ( -- )  ;

Object virtual: m   ;
Float method: m ;

Object Class new: A
A method:      m   ;
A classMethod: m   ;

A Class new: B

: em | i | #[ loop: i [ ] ]     bench . ;

: fc | i | #[ loop: i [ f ] ]   bench . ;

: mf | i | #[ loop: i [ 5.0 m ] ] bench . ;

: mb | i b |
   B new ->b
   #[ loop: i [ b m ] ] bench . ;

: mi | i j |
   10 ->j
   #[ loop: i [ j m ] ] bench . ;

: ca | i | #[ loop: i [ A m ] ] bench . ;

: cm | i cl |
   A ->cl
   #[ loop: i [ cl m ] ] bench . ;
```

Results are :



	Total (ms)	Calls (ms)	/ direct call
1000000000 em	1690	0	0,00
1000000000 fc	1970	280	1,00
1000000000 mf	1970	280	1,00
1000000000 mb	2250	560	**2,00**
1000000000 mi	2530	840	**3,00**
1000000000 ca	1975	285	1,02
1000000000 cm	3378	1688	**6,03**

The first column is the total time in milliseconds. The second column is the total time for calls (ie subtracting the time spent for the empty loop). The third column is the cost of a the polymorphic call compared to a direct call.

#em is the benchmark for an empty loop.

#fc calls an empty function (f) n times. Of course, here, a direct call is compiled.

#mf test calls a message that is optimized during compile time into a direct call (because TOS is a literal). It runs in the same time as #fc

#mb calls a message that will not be optimized during compile time. It uses the dispatch code described in (3.3). This test shows that a polymorphic call takes twice the time compared to a direct call.

#mi calls a message that will not be optimized during compile time. It uses the dispatch code described in (3.3), but as the receiver is an integer, it uses the special case for integers/null (step 1). In this case, the polymorphic call takes three times the time for a direct call. This is probably the result of explicit jumps that breaks CPU optimizations.

#ca calls a class method on class A. It is optimized (because TOS is a literal) and runs in the same time than a direct call.

#cm calls a message on class A that will not be optimized. The code to run is searched each time in the hierarchy, without VTM. Those calls are 3 times slower than MVT dispatch for methods and 6 times slower that a direct function call.

6 Future work

The main purpose of this dispatch implementation is to keep high performances while allowing extended classes.

Nevertheless, future work may be interesting on alternative mechanisms for MVT storage.

In a typical application :
- Some messages will be implemented only for one class and a big MVT will be allocated for only one pertinent slot (this is the worst pattern).
- Some messages will be implemented only at Object level and the MVT will be very small.
- Some messages will be in-between (#read, #+, #size , #<<, #log, ...) and the MVT will be partially filled.

In order to handle the first pattern, a possibility would be to have 2 sizes for each MVT : the minimum class index and the maximum class index. This would complexify a little the rumMessage function with a performance penalty, but save a lot of space when a message is implemented only for a few classes.

Other mechanisms have also been discussed in the literature ([ERTL11]) and could be implemented and benchmarked in a future work.

7 Conclusion

Oforth implements a full dynamic binding for method dispatch for methods, associated with extendable classes.

During compile time, some optimizations occur to reduce, when possible, messages call to direct call.

Virtual tables are "by message" and not "by class". They are not defined at compile time but calculated and reallocated at runtime, while messages are performed. This allows to save some memory (unused messages don't use MVT) and to extend classes.

This is done without suffering much performances penalties as the same code is used to manage MVT and to call methods : everything is done by calling to the addresses stored in the MVT slots.

8 References

[RP96] Bradford J. Rodriguez and W. F. S. Poehlman. A survey of object-oriented Forths. SIGPLAN Notices, pages 39–42, April 1996.

[DUC11] Roland Ducournau. Implementing statically typed object-oriented programming languages. ACM Computing Surveys, 43(3):Article 18, April 2011.

[ERT12] M. Anton Ertl. Methods in objects2: Duck Typing and Performance. 28th EuroForth Conference 2012.

A List Toolkit

Dr. Peter Knaggs

September 15, 2018

The ordered list is an essential construct in programming. Indeed it is fundamental to Forth in that the dictionary is a list of word definitions, the search order, accessed via the ungainly words `GET-CURRENT` and `SET-CURRENT` is a simple list of word lists. Many systems access this as a stack, providing words such as `+ORDER` and `-ORDER`.

While it is probably easer to implement a list in Forth than in many other languages of its generation. Many modern languages (e.g., Perl, PHP, JavaScript, Java, C#, etc.) include an ordered list as a first-class variable type. Indeed the ordered list is fundamental to the functional language paradigm (languages such as SML and Haskall). This paper is an attempt to provide a tool kit for manipulating a list of generic elements.

Used correctly an ordered list is able to represent many different data structures:

Stack	Add to the end of a list and remove from the end the list.
Queue	Add to the end of a list and remove from the start of the list.
Array	Index into a list
Sequence	
Set	Check for membership before adding element.

The philosophy of this tool kit is to provide a set of words to manipulate a list of nodes, each with a single cell element. While this means the list is capable of holding basic data types (character, integer) is is also able to hold a reference to another data structure such as a data node or object. This allows the list to be generic. The word names used in the proposed tool kit have been chosen because they do not exist in current systems.

1 Implementation

The words have been defined in a way that does not require any particular implementation, thus allowing the developer to use the most appropriate implementation method for there system. For example:

Array	When a list is constructed a *capacity* parameter is given which provides the implementer with a minimum list capacity. Thus it is possible to implement this word set with a simple array. Although not so useful. A cyclic array would be a better fit, allowing manipulation of both the start and and end of the list.

Linked List	A single (or doubly) linked list is the more traditional implementation technique used by most Forth programmers. If the list constructor ignores the *capacity* parameter, a linked list implementation is more than viable.
Array List	An array of *capacity* elements held in user memory. When an element is added to the list it will check if the list has space for the element. If not then it will allocate a new array of *capacity* $\times$ *ratio* elements, moving the exiting array into the new array, and returning the old array back into the user memory.
Bucket List	A linked list of buckets (an array of *capacity* elements) held in user memory. When an element is added to the list, it will check the list has space for the element. If not then it will allocation another bucket which is linked to the exiting list.

2 Indexing

When indexing into the list an index parameter (n) is used. This allows indexing into the list from the start of the list (when n is positive) or from the end of the list (when n is negative). Thus $n = -1$ will provide access to the last element of the list, while $n = 0$ will provide access to the first element in the list.

When the index is beyond the range of the list, an exception is thrown. For example, if a list has 10 elements, an index of 10 (or -11) will be beyond the range of the list. As the standard does not provide an "out of memory", "out of range" or "bad index" exception, the "result out of range" (-11) exception has been abused to indicate the index is out of range.

3 Iterator

Those languages that provide for a list also provide a mechanism to iterate over a list of elements in the general form:

for (*variable* **in** *list*) { *body* }

where the *body* of the loop is executed once for each element of the *list* and *variable* is given the current element of the list on each iteration.

A Forth variant of this idea is proposed, where the word `FOREACH` takes a list and creates an implementation dependent data structure *iter* that is used to control the iteration loop. The word `I` (from the iterator word set) places the current list element on the data stack, while the `NEXT` word (from iterator word set) ends the iteration loop.

Therefore we could implement a loop to sum the contents of a list of n as:

```
: SUM ( list -- n )
  0 SWAP FOREACH I + NEXT
;
```

If *iter* where to contain an *xt* of a word that determines the next element of the iteration, `NEXT` could simply call that *xt*. Replacing the iteration constructor word (`FOREACH`) with a word more appropriate to the iteration type that builds the corresponding *iter* (complete with relevant *xt*). This would provide the ability to iterate over any type of data structure, such as a word list, the characters in a string, the lines in a file, rows in a database and so on.

4 List toolkit

`CREATE-LIST` (*n* -- *list*) TOOLS

Create a new list capable of holding a minimum of *n* single cell elements, returning a list identifier *list*.

Comments:
This is a basic list constructor that builds a new anonymous list. n is the capacity *parameter. It also introduces the opaque type list. Depending on the implementation this could be an xt or a variable pointing to the first element in this list.*

Formally: $list = \langle\ \rangle$

`LIST:` (*n* "$\langle ccc\rangle$" --) "list-colon" TOOLS

Crate a new list $\langle ccc\rangle$ capable of holding a minimum of *n* elements.

Implementation:

```
: LIST: ( n "<ccc>" -- ) CREATE-LIST CONSTANT ;
```

Comments:
This is the named list constructor.

Separating out the named list constructor and the anonymous list constructor allows the developer to combine the anonymous constructor with a system specific method of creating a definition, thus allowing developers to build a list factory.

`LIST+` (*x list* --) "list plus" TOOLS

Add the element *x* to the end of the *list*.

If there is insufficient space in the list, the system may extend the list to allow the new element or throw a -11 (out of range) exception.

Comments:
This is frequently known as `push`*,* `add` *or* `append` *in other languages.*

Formally: $list' = list \frown x$

`LIST-` (*list* −− *x*) "list minus" TOOLS

Remove the last element the *list* placing on the stack (*x*).

If there are no elements in the list a -11 (out of range) exception is thrown.

Comments:
This is frequently known as `pop` *or* `remove` *in other languages.*

Formally: $list = list' \frown x$

`+LIST` (*x list* −−) "plus list" TOOLS

Add the element *x* to the start of the *list*.

If there is insufficient space in the list, the system may extend the list to allow the new element or throw a -11 (out of range) exception.

Comments:
This is frequently known as `insert` *or* `unshift` *in other languages.*

Formally: $list' = x \frown list$

`-LIST` (*list* −− *x*) "minus list" TOOLS

Remove the first element of the *list* placing it on the stack (*x*).

If there are no elements in the list a -11 (out of range) exception is thrown.

Comments:
This is frequently known as `shift` *or* `remove` *in other languages.*

Formally: $list = x \frown list'$

`CONCAT` ($list_1$ $list_2$ −−) TOOLS

Append the content of $list_1$ to $list_2$ such that $list_2$ contains all the elements in $list_2$ followed by all of the elements in $list_1$.

If there is insufficient space in $list_2$, the system may extend the list to allow for the new elements or throw a -11 (out of range) exception.

Comments:
This is frequently known as `concatenate`, `append` *or* `join` *in other languages.*

Formally: $list_2' = list_2 \frown list_1$

`>LIST` (*x n list* ––) “to-list” TOOLS

Insert the element *x* into the *list* at position *n*, relative to the start of the list. If *n* is negative, the position is relative to the end of the list. If *n* is beyond the end of the list a -11 (out of range) exception is thrown.

When *n* is 0, this is the same as `+LIST`. When *n* is -1, this is the same as `LIST+`.

Comments:
This is frequently known as `insert` *or* `insertAt` *in other languages.*

Formally: $list' = list_{0..(n-1)} \frown x \frown list_{n..\#list}$

`LIST>` (*n list* –– *x*) “list-from” TOOLS

Remove element *n* from the *list* returning the removed element (*x*). *n* is relative to the start of the list. If *n* is negative, the position is relative to the end of the list. If *n* is beyond the end of the list a -11 (out of range) exceptionis thrown.

When *n* is 0, this is the same as `-LIST`. When *n* is -1, this is the same as `LIST-`.

Comments:
This is frequently known as `remove` *or* `removeAt` *in other languages.*

Formally: $list' = list_{0..(n-1)} \frown x \frown list_{(n+1)..\#list}$

`/LIST` (*list* –– *u*) “slash list” TOOLS

Return the number of elements (*u*) in *list*.

Comments:
This is known as `count` *or* `length` *or simply* `#` *in other languages.*

Formally: $u = \#list$

`#LIST` (*x list* –– *u*) “number list” TOOLS

Return the number of times (*u*) the element *x* appears in the *list*. If *x* does not appear in *list*, *u* will be 0.

Comments:
This is known as `count` *in other languages.*

Formally: $u = \#\{\forall e \in list \mid e = x\}$

`?LIST` (*x n list* −− *u* | *-1*) “query list” TOOLS

Return the position (*u*) of the first occurrence of *x* in the *list*, starting the search at position *n*. *n* is relative to the start of the list if positive and relative to the end of the list if negative. Return -1 if *x* is not found or if the start position (*n*) is beyond the range of the *list*. The first element in the list is at position 0 and the last element is at position -1. The result (*u*) is always given relative to the start of the list.

Rationale:

One can check for membership of a list by comparing the result to -1.

```
: in ( x list -- flag )
    0 SWAP ?LIST 0< 0=
;
```

Comments:

This is known as `indexOf` *in other languages.*

Formally: $(list_u = x \wedge x \notin list_{n..u-1}) \vee (u = -1 \wedge x \notin list)$

`LIST@` (*n list* −− *x*) “list fetch” TOOLS

Return the element (*x*) at position *n* of *list*. *n* is relative to the start of the list if positive and relative to the end of the list if negative. The first element in the list is at position 0 and the last element is at position -1. If *n* is beyond the range of the *list* a -11 (out of range) exception is thrown.

Comments:

Other languages use the index operator `[]` *to index into a list.*

Formally: $x = list_n$

`LIST!` (*x n list* −−) “list store” TOOLS

Set the element at position *n* of *list* to be the value *x*. *n* is relative to the start of the list if positive and relative to the end of the list if negative. The first element in the list is at position 0 and the last element is at position -1. If *n* is beyond the range of the *list* a -11 (out of range) exception is thrown.

Comments:

Other languages use the index operator `[]` *to index into a list.*

Formally: $x = list'_n$

`TRAVERSE-LIST` (*list xt(x * i x – x * j)* ––) TOOLS

The *xt* is executed once for each element in the *list*, with each element being presented to the *xt* on the top of the stack (*x*).

The *xt* may change the data stack during execution.

Rationale:

For example, it is possible to sum a list of *n* using the following:

```
: SUM ( list -- n )
    0 SWAP ['] + TRAVERSE-LIST
;
```

Comments:

This is known as `map` *or* `each` *in other languages.*

Formally: $\forall x \in list \bullet xt(x)$

`FOREACH` TOOLS

Interpretation:

The interpretation semantics are undefined.

Compilation: (C: –– *dest*)

Put the next location for a transfer of control (*dest*) onto the control flow stack. Append the run-time semantics given below to the current definition.

Run-time: (*list* ––) (R: –– *iter*)

Initialise an iteration over the *list*, placing the iteration control (*iter*) onto the return stack.

Rationale:

For example, it is possible to sum a list of *n* using the following:

```
: SUM ( list -- n )
    0 SWAP FOREACH I + NEXT
;
```

Comments:

The opaque type iter *will vary depending on the implementation, but will probably contain at least the list identifier and the current position within the list. This will alter the search order such that the iteration version of the* `I` *and* `NEXT` *words are found before the Core versions.*

`I` (–– *x*)(R: *iter* –– *iter*) TOOLS

Use the iteration control (*iter*) to return the current value (*x*) in the list iteration started by `FOREACH`.

Comments:

iter *is used to obtain the current value in the iteration, but is not altered.*

Interpretation:

The interpretation semantics are undefined.

Compilation: (C: *dest* --)

Append the run-time semantics given below to the current definition, resolve the backward reference *dest*.

Run-time: (R: *iter* --)

Use the iteration control (*iter*) to determine the next element in the iteration. If there is another element in the iteration, update the iteration control and continute execution at the location specified by *dest*. If there are no more entries in the list, remove the iteration control (*iter*) from the return stack and continue execution with the next instruction.

Comments:

This should remove the iteration word list from the search order.

RFC Ulrich Hoffmann / Andrew Read
EuroForth 2018

Forth: A New Synthesis - Progress Report
Growing Forth with seedForth

1 Introduction

The "new synthesis" of Forth is an ongoing effort in spirit of the Forth Modification Laboratory workshops. Its aim is to identify the essentials of Forth and to combine them in a new way to build systems that can scale-down as Forth always did and can scale-up to large applications and development projects.

The new synthesis is guided by the two principles biological analogy and disaggregation.

We scrutinise many aspects of traditional and modern Forth implementations trying to separate techniques that are normally deeply intertwined. After isolating the techniques we thrive to combine them in new ways.

Our findings so far can be summarized:

- high level inner interpreter (EuroForth 2016, [1])
  We showed that a traditional Forth indirect threaded code virtual machine can implemented in high level Forth bringing threaded code manipulation tricks to any Forth implementations.
- stacks for structured data (Forth Tagung (convention) 2017, german. [2])
  Stores and handles structured items (strings, queues, lists, stacks) on stack and return stack. No memory required. Shows how terminal input and number output can work without random accessible memory.
- handler based outer interpreter (EuroForth 2017, [3])
  This demonstrates a very simple modular architecture for the Forth text interpreter separating interpretation and compilation actions for parsed tokens by handlers that possible consume and process a token text or pass it on unprocessed.
- preForth, simpleForth, Forth (Forth Tagung (convention) 2018, german, [4])
  Presents preForth, a minimalistic non-interactive Forth kernel that can bootstrap itself, simpleForth, still non-interactive, which adds memory and control structures and Forth a simple interactive Forth bootstrapped from preForth/simpleForth. See below for details.
- String Descriptors (EuroForth 2018, [5])
  We revise different Forth string manipulation facilities and present string descriptors, an intermediate string representation balancing utility and ease of implementation.
- Regex (part of string descriptors paper, EuroForth 2018, [5])
  Presents a simple implementation of regular expressions extended for Forth's demand to detect space separated tokens and intended to be used in the token detection part of handler based outer interpreters.

We try to use Forth wherever possible in order to minimize semantic and formalism mismatches. Everything should be readily available - no hidden secrets.

Of course many of the subjects we are looking at have been used by others in the Forth community and outside - we are dwarfs standing on the shoulders of giants - however we believe our new synthesis to be original.

2 preForth (simpleForth and Forth)

preForth is a minimalistic non-interactive Forth kernel that can bootstrap itself and can be used as an easy-to-port basis for a full Forth implementation.

preForth feels like Forth - it is mainly a sublanguage of ANS-Forth - but is significantly reduced in its capabilities.

Features:

minimal control structures, no immediate words, strings on stack, few primitives

just

- stack
- return stack
- only ?EXIT and recursion as control structures
- colon definitions
- optional tail call optimization
- IO via KEY/EMIT
- signed single cell decimal numbers (0-9)+
- character constants via 'c'-notation
- output single cell decimal numbers

and

- no immediate words, i.e.
- no control structures IF ELSE THEN BEGIN WHILE REPEAT UNTIL
- no defining words
- no DOES>
- no memory @ ! CMOVE ALLOT ,
- no pictured numeric output
- no input stream
- no state
- no base
- no dictionary, no EXECUTE, not EVALUATE
- no CATCH and THROW
- no error handling

preForth is based on just 13 primitives: emit key dup swap drop 0< ?exit >r r> - nest unnest lit which are defined in the host language. Implementations in i386 assembler and ANSI-C exist. Executable code is generated by a host language translator (compiler, assembler).

As an example the definition of the primitive >r in preForth (i386 resp. C) looks like:

```
\ i386                                  \ C
code >r ( x -- ) ( R -- x )             code >r ( x -- ) ( R -- x )
        pop ebx                            *++rp=*sp--
        lea ebp,[ebp-4]                 ;
        mov [ebp], ebx
        next
;
```

Using the defined primitives as building blocks preForth allows for colon definitions to define new words. As there are no control structures all definitions have to be (tail) recursive and use the primitive ?EXIT for conditional branches. In essence ?EXIT is a conditional branch to the end of a definition, recursion an unconditional jump to the beginning of a word. Here is the definition of the word SHOW that displays a string represented characterwise on the data stack (First character deepest, count on top of stack):

```
\ preForth: display topmost string
: show ( S -- )
   ?dup 0= ?exit  swap >r 1- show r> emit ;
```

Tail calls can be tagged with the TAIL prefix and preForth then converts the call to a branch:

```
\ preForth: read and append non-control characters to the given string.
\ Return resulting string and the delimiting character.
: scan ( S1 -- S2 bl )
    key dup bl > 0= ?exit swap 1+  tail scan ;
```

PreForth bootstraps itself. Using the i386 version:

```
$ cat preForth-i386-rts.pre preForth-rts.pre \
      preForth-i386-backend.pre preForth.pre ./preForth >preForth.asm
$ assemble preForth.asm ./preForth
```

The initial bootstrap can be done with gforth or swiftForth.

preForth is modularized into platform specific and plattform independent parts that concatenated build the complete preForth system. The overall source code (i386) is 820 LOCs:

```
$ wc preForth-i386*.pre preforth.pre preForth-rts.pre
     166     760    3729 preForth-i386-backend.pre
     175     403    2553 preForth-i386-rts.pre
     328    1908   10045 preforth.pre
     151     695    2981 preForth-rts.pre
     820    3766   19308 total
```

2.1 simpleForth

simpleForth is an extension to preForth built using preForth. It is still non-interactive but adds

- control structures IF ELSE THEN BEGIN WHILE REPEAT UNTIL
- definitions with and without headers in generated code
- memory: @ ! c@ c! allot c, ,
- variable, constants
- ['] execute
- immediate definitions

Enough convenient words to formulate an interactive Forth:

2.2 Forth

Forth is a simple interactive Forth system built using simpleForth. Forth is open ended and still has an incomplete set of features. Work in progress. Here is a system start:

```
Forth 1.2.0

last * warm cold empty patch minor major banner quit restart REPEAT WHILE AGAIN
UNTIL BEGIN THEN ELSE IF ; : constant variable header cmove compile, , allot here
dp +! clearstack interpret parse-name \ .( ( parse (interpreters ?word (compilers ,word
immediate !flags @flags or and #immediate ] [ interpreters compilers handlers ,'x' ?'x'
,# ?# scan skip source /string >in query #tib tib accept min words .name l>interp l>name
l>flags type count cell+ cells find-name .s prefix? compare 2dup 2drop rot off on ?dup +
space bl cr . u. negate > 1- nip = 0= pick 1+ < over depth execute c! ! c@ @ ?branch
branch lit exit unnest - r> >r ?exit 0< drop swap dup key emit bye

Inspect sources and generated files.

Have fun. May the Forth be with you.

>
```

3 seedForth

Defining and using preForth was (still is) quite satisfying but we were quite unhappy with simpleForth and Forth as several of their aspects tend to be defined twice - such as the structure of headers or defining words - in the overall setup.

This seems to be an issue from which also (all?) target and cross compilers suffer: The cross compiler needs to define a structure of the target system in order to be able to generate it. The generated running system might also generate the very same structure and so needs a description of its own...

seedForth tries to eliminate this issue by further simplifying the system structure.

seedForth is a very small interactive Forth system that can be extended to a full Forth implementation.

seedForth is really very small (460 LOC) but already has a dictionary and is extensible by colon definitions. Currently an i386 and an AMD64 implementation exist.

In order to eliminate parsing, number conversion and string headers, seedForth accepts source code in byte tokenized form that is generated by a very simple tokenizer written in gforth that has roughly 100 lines only. It is assumed that also a capable text editor could perform the transformation from text source to byte tokenized source code.

Instead of names, words just have consecutive indices to identify them.

3.1 seedForth virtual machine

The seedForth virtual machine is defined in preForth. It has the following components:

data stack
as usual for operands

return stack
as usual for return addresses and intermediate values

dictionary
addressable memory for code and colon definitions (not initially for headers)
a dictionary pointer dp register points to the next free location.

headers
an array that maps word indices to their starting address in the dictionary
a head pointer hp points to the next free header entry.

A dictionary lookup is a very simple indexed access to the headers array. When defining a new word, the current address in the dictionary is recorded in the headers array and the new definition builds up at the current dictionary location.

Here is a list of seedForth predefined words:

```
  $00 #FUN: bye       $01 #FUN: emit          $02 #FUN: key       $03 #FUN: dup
  $04 #FUN: swap      $05 #FUN: drop          $06 #FUN: 0<        $07 #FUN: ?exit
  $08 #FUN: >r        $09 #FUN: r>            $0A #FUN: -         $0B #FUN: unnest
  $0C #FUN: lit        $0D #FUN: @            $0E #FUN: c@        $0F #FUN: !
  $10 #FUN: c!        $11 #FUN: execute       $12 #FUN: branch    $13 #FUN: ?branch
  $14 #FUN: negate    $15 #FUN: +             $16 #FUN: 0=        $17 #FUN: ?dup
  $18 #FUN: cells     $19 #FUN: +!            $1A #FUN: h@        $1B #FUN: h,
  $1C #FUN: here      $1D #FUN: allot         $1E #FUN: ,         $1F #FUN: c,
  $20 #FUN: fun       $21 #FUN: interpreter   $22 #FUN: compiler  $23 #FUN: create
  $24 #FUN: does>     $25 #FUN: cold          $26 #FUN: depth     $27 #FUN: compile,
  $28 #FUN: new       $29 #FUN: couple        $2A #FUN: and       $2B #FUN: or
  $2C #FUN: catch     $2D #FUN: throw         $2E #FUN: sp@       $2F #FUN: sp!
  $30 #FUN: rp@       $31 #FUN: rp!           $32 #FUN: $lit
```

It has the usual suspects but also seedForth specific definitions. Most of them are already defined in high level code.

h@ (i -- addr)
: does the indexed access to the headers array. Given a function index it returns its start address in dictionary.

h, (x --)
: makes a new headers entry that points to the address x.

here and , deal with the dictionary pointer.

key and **emit** communicate byte values.

fun starts a new colon definition (compiles entry code, assigns the next function index and records the start address for this function index in the headers array, starts compilation).

3.2 seedForth's interpreter and compiler

INTERPRETER is accepting one token of byte tokenized source code after the other and executes it.

```
: interpreter ( -- )
   key execute   tail interpreter ;
```

So - how to push operands on the seedForth stack (literals)? The tokenized source contains the sequence key n where n is the number to push.

COMPILER is accepting tokenized source code and compiles it.

```
: compiler ( -- )
   key ?dup 0= ?exit compile, tail compiler ;
```

The token 0 (bye) ends the compiler loop. To compile the literal n the sequence

```
      lit 0 key n , compiler
```

is required. The tokenizer uses #, to do this.

COLD displays the boot message "seed", initializes the headers array for the predefined words and starts the interpreter.

3.3 seedForth example program

Here is a small seedForth human readable source program:

```
     program demo.seed

     'H' # emit  'e' # emit  'l' # dup emit emit  'o' # emit  10 # emit

     ': 1+ ( x1 -- x2 ) 1 #, + ;'

     'A' #  1+  emit   \ outputs B

     end
```

The tokenizer transforms this to the byte tokenized source:

```
00000000  02 48 01 02 65 01 02 6c  03 01 01 02 6f 01 02 0a  |.H..e..l....o...|
00000010  01 20 0c 00 02 01 1e 22  15 0b 00 02 41 33 01 00  |. ....."....A3..|
```

Running seedForth on this input gives:

```
cat demo.seed | bin/seedForth
seed
Hello
B
```

3.4 seedForth capabilities

As seedForth already has C@ @ and C! ! you have full access to the dictionary and can define also not-yet-present structures, such as

- headers with dictionary search and NDCS support
- dynamic memory allocation with allocate, resize and free
- text interpreter and compiler that work on non tokenized source
- compiling words
- a Forth assembler for the target platform and additional primitives, defining words including DOES> afterwards
- multitasking
- OOP
- file and operating system interface
- access to hardware
- the tokenizer and preForth can eventually also be expressed in seedForth and so it will be self contained.

seedForth will bootstrap to a full Forth entirely by extension. We'll never leave it. No need to define structures twice.

Since its invention seedForth continues to evolve. It now supports defining words, string literals, an integrated testing facility and dynamic memory management. Support for regular expressions using string descriptors and a handler based outer interpreter is in the works.

4 Summary

We presented the status quo of our project "Forth - A new Synthesis".

We gave an overview of the topics that we already have addressed and we introduced preForth and seedForth, two very small Forth systems that make use of our ideas, one non-interactive that can bootstrap itself, one potentially interactive that can be extended seamlessly.

By disaggregating Forth and recombining its isolated constituents in new ways, we can construct very simple yet very flexible systems that have the potential to scale up coherently to larger systems and applications. We did not achieve this yet.

We are on our way.

Forth: words, stacks, blocks

5 References

[1] Implementing the Forth Inner Interpreter in High Level Forth,
Ulrich Hoffmann, EuroForth Conference 2016, Reichenau, 2016

[2] Stack of Stacks,
Ulrich Hoffmann, Forth Tagung 2017, Karlkar, 2017

[3] A Recognizer Influenced Handler Based Outer Interpreter Structure,
EuroForth 2017, Bad Vslau, 2017

[4] Bootstrapping Forth,
Forth Tagung 2018, Linux Hotel,
Essen, 2018

[5] A descriptor based approach to Forth strings,
Andrew Read and Ulrich Hoffmann, EuroForth conference, Edinburgh, 2018

[6] String descriptors on GitHub
https://github.com/Anding/descriptor-based-strings

[7] preForth and seedForth on GitHub
https://github.com/uho/preForth

net2o: MINΩΣ2 GUI, $quid “crypto”

($quid = ethical micropayment with efficient BlockChain)

Bernd Paysan

EuroForth 2018, Edinburgh

Motivation

5 Years after Snowden

What changed?

Politics

- EU parliament wants upload filters 🤦‍♀️
- EU parliament taxes the link (instead: “right”) 🤦‍♀️🤦‍♀️
- EU parliament wants filtering “terrorist contents” 🤦‍♀️🤦‍♀️🤦‍♀️
- Germany wants a Cyberadministration like CAC (Medienstaatsvertrag) 🤦‍♀️🤦‍♀️🤦‍♀️🤦‍♀️
- Backdoors still wanted (“reasonable encryption”) 🤦‍♀️🤦‍♀️🤦‍♀️🤦‍♀️🤦‍♀️

Competition

- Cambridge Analytica scandal (Facebook)
- Security fuckups: Passwords pawned, chat log saved unprotected in the cloud, etc.

Progress

- The ECHR ruled that GCHQ’s dragnet surveillances violates your rights
- net2o becomes more and more usable

5 Years after Snowden

What changed?

Politics

- Germany wants a Cyberadministration like CAC (Medienstaatsvertrag) 🤦‍♀️🤦‍♀️🤦‍♀️🤦‍♀️
- Backdoors still wanted (“reasonable encryption”) 🤦‍♀️🤦‍♀️🤦‍♀️🤦‍♀️🤦‍♀️
- Legalize it (dragnet surveillance) 🤦‍♀️🤦‍♀️🤦‍♀️🤦‍♀️🤦‍♀️🤦‍♀️
- You can't reasonably expect privacy on your own PC 🤦‍♀️🤦‍♀️🤦‍♀️🤦‍♀️🤦‍♀️🤦‍♀️🤦‍♀️
- “Crypto” now means BitCoin 🤦‍♀️🤦‍♀️🤦‍♀️🤦‍♀️🤦‍♀️🤦‍♀️🤦‍♀️🤦‍♀️

Competition

- Cambridge Analytica scandal (Facebook)
- Security fuckups: Passwords pawned, chat log saved unprotected in the cloud, etc.

Progress

- The ECHR ruled that GCHQ’s dragnet surveillances violates your rights
- net2o becomes more and more usable

MINΩΣ2 Widgets

Design principle is a Lego–style combination of many extremely simple objects

actor	base class that reacts on all actions (clicks, touchs, keys)
animation	action for animations
widget	base class for all visible objects
glue	base class for flexible objects
tile	colored rectangle
frame	colored rectangle with border
icon	icon from an icon texture
image	larger image
edit	editable text: 中秋节快乐！ Happy autumn festival! 🌙🌕
text	text element/Emoji/中文/... 😀🤭😁😂😇😈🙈🙉🙊❤️💔💕💖💗💘🍺🍻🎉🎻🎺
part-text	pseudo–element for paragraph breaking
canvas	vector graphics (TBD)
video	video player (TBD)

MINΩΣ2 Widgets

Design principle is a Lego–style combination of many extremely simple objects

actor	base class that reacts on all actions (clicks, touchs, keys)
animation	action for animations
widget	base class for all visible objects
glue	base class for flexible objects
tile	colored rectangle
frame	colored rectangle with border
icon	icon from an icon texture
image	larger image
edit	editable text: 中秋节快乐！ Happy autumn festival! 🌙🌕
text	text element/Emoji/中文/... 😀🤭😁😂😇😈🙈🙉🙊❤️💔💕💖💗💘🍺🍻🎉🎻🎺
part-text	pseudo–element for paragraph breaking
canvas	vector graphics (TBD)
video	video player (TBD)

MINΩΣ2 Displays

Render into different kinds of displays

viewport	Into a texture, used as viewport
display	To the actual display (no class, just the default)

Minimize Draw Calls

OpenGL wants as few draw–calls per frame, so different contexts are drawn in stacks with a draw–call each

init	Initialization round
bg	background round
text	text round (same draw call as bg round, just different code)
image	draw images with one draw–call per image

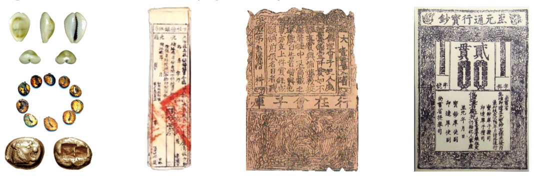
$quid & SwapDragonChain

Content:

Money	What’s that all about?
BitCoin	Shortcomings of a first proof of concept
Wealth	Ethical implication in deflationary systems
Proof of	Trust instead Work
BlockChain	What’s the actual point?
Scale	How to scale a BlockChain?
Contracts	Smart oder dumb?
$quid	Ethical ways to create money

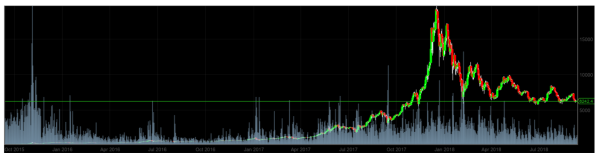
What’s Money?

Commodity ~:	Objects with inherent value
Promissory note:	Bank created paper for commodity
Representative ~:	Promise to exchange with “standard object” (e.g. gold)
Fiat ~:	No inherent value; promise, if any, as legal tender
Legal tender:	Medium of payment by law



BitCoins — early “Crypto” shortcoming

- Proof of work: wasteful and yet only marginally secure
- Inflation is money’s cancer, deflation its infarct
- Consequences: unstable exange rate, high transaction fees
- Ponzi scheme–style bubble
- (Instead of getting Viagra spam I now get BitCoin spam)
- Can’t even do the exchange transaction on–chain

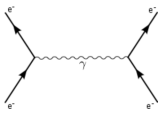


Dumb Contracts

- Smart Contracts: Token–Forth subset (BitCoin), JavaScript (Ethereum)
- For Smart Contracts you need a lawyer, a programmer, *and* a pentester
- Keep it simple: A contract must have a balanced balance
- Select sources (S), select their assets (A), debit them (±)
- select destinations (D), set assets&credit them
- Shortcut: balance an asset (B)
- Obligations for debt and futures (O)
- Sign the target account with new content+hash of the contrat

Examples:

Transfer	SA–SBDD
Cheque	SA–D, cash: SA–DSBD
Exchange/Purchase	SA+A–DSBBD



Wealth & Ethics

- Huge first mover advantage
- Already worse wealth distribution than neoliberal economy
- Huge inequality drives society into servitude, not into freedom
- No concept of a credit
- Lightning network also binds assets (will have fees as consequence)

EXCUSE ME, BUT DO YOU HAVE A MOMENT TO TALK ABOUT OUR LORD AND SAVIOR THE FREE MARKET?

$quid: Ethical mining

- Concept of mining: Provide difficult and rare work
- Suggesting: Provide vouchers for free software development sponsorships
- These vouchers are tradeable on their own
- Free software is public infrastructure for the information age
- That way, we can encourage people to sponsor out of self–interest
- They get a useful and valueable token back
- Or they develop FOSS themselves to earn (fiat) money

Decentral bank?

- Central bank grants big banks credits, which then are gambled in the stock market
- The decentral bank gives credits to small business
- Credit assessment more like croudfunding

Proof of What?!

Challenge	Avoid double–spending
State of the art:	Proof of work
Problem:	Proof of work burns energy and GPUs
Suggestion 1:	Proof of stake (money buys influence)
Problem:	Money corrupts, and corrupt entities misbehave
Suggestion 2:	Proof of well–behaving (trust, trustworthyness)
How?	Having signed many blocks in the chain gains points
Multiple signers	Not only have one signer, but many
Suspicion	Don't accept transactions in low confidence blocks
Idea	Repeated prisoner’s dilemma rewards cooperation

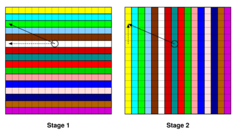
BTW: The attack for double spending also requires a MITM–attack

Literatur & Links

Bernd Paysan	*net2o fossil repository*
	`https://net2o.de/`
Bernd Paysan	*$quid cryptocurrency & SwapDragonChain*
	`https://squid.cash/`

SwapDragon BlockChain

- Banks distrust each others, too (i. e. GNU Taler is not a solution)
- Problem size: WeChat Pay peaks at 0.5MTPS (BTC at 5TPS)
- Lightning Network doesn’t stand an overrun–the–arbiter attack
- Therefore, the BlockChain itself needs to scale
- Introduce double entry booking into the distributed ledger
- Partitionate the ledgers by coin pubkey
- Use n–dimensional ledger space to route transactions

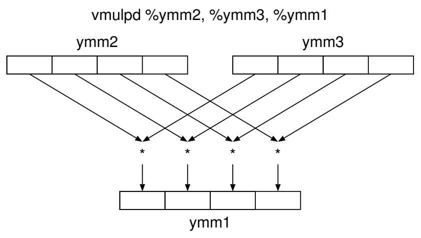


Software Vector Chaining

M. Anton Ertl
TU Wien

Data Parallelism and SIMD instructions

- Data parallelism in programming problems
- Hardware provides SIMD instructions
  Cray-1 vector instructions, Intel/AMD SSE/AVX, ARM Neon/SVE



- Little programming language support

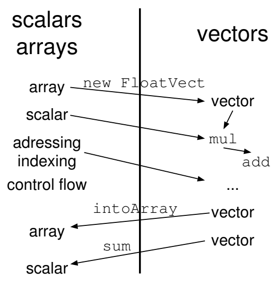
Programming language support: How?

- Manual Vectorization
- Application vector length
- Opaque, immutable vectors with value semantics
- Vector stack

```
: vcomp ( va vb -- vc )
  vdup sf*v vswap vdup sf*v sf+v sfnegatev ;
```

Properties, benefits and drawbacks

- Vectors are immutable (value semantics)
- – Explicit conversion from/to memory
- + gives control to programmer,
  who can make conversions infrequent
- + Padding to SIMD granularity
- + Aligning to SIMD granularity
- + No aliasing problems
- + Results do not overlap input operands
- + only explicit dependences
- + vectors are a separate world
- + Compiler can arrange computations



Implementation

simple sf+v

```
simple:
 vmovaps (%rdi,%r10,1),%ymm0
 vaddps  (%rsi,%r10,1),%ymm0,%ymm0
 vmovaps %ymm0,(%rdx,%r10,1)
 add     $0x20,%r10
 cmp     %r10,%rcx
 ja      simple
```

fused vcomp

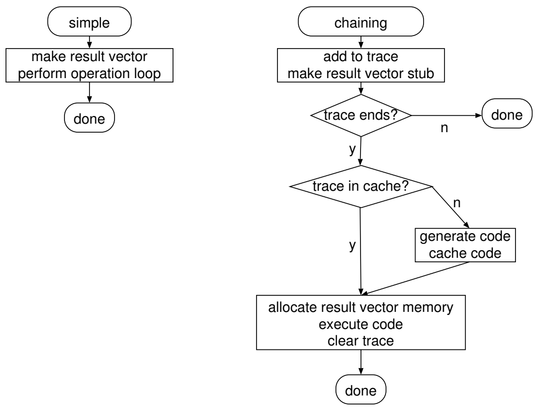
```
fused:
 vmovaps (%rdi,%r10,1),%ymm0
 vmulps  %ymm0,%ymm0,%ymm1
 vmovaps (%rsi,%r10,1),%ymm2
 vmulps  %ymm2,%ymm2,%ymm3
 vaddps  %ymm1,%ymm3,%ymm1
 vxorps  %ymm1,%ymm4,%ymm1
 vmovaps %ymm0,(%rdx,%r10,1)
 add     $0x20,%r10
 cmp     %r10,%r9
 ja      fused
```

... but how?

Who performs vector loop fusion?

Compiler	Run-time Library
+ Low run-time overhead	– High run-time overhead
– High implementation effort?	+ Low implementation effort
– Control-flow may limit fusion	+ Fuses across control flow
– Aliasing plays a role	+ Dependencies resolved
	Software Vector Chaining

Implementing a vector operation

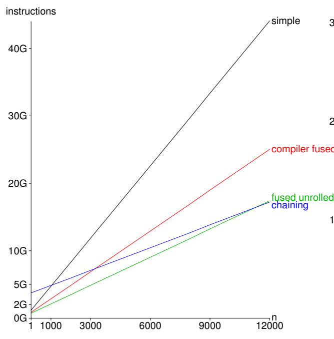


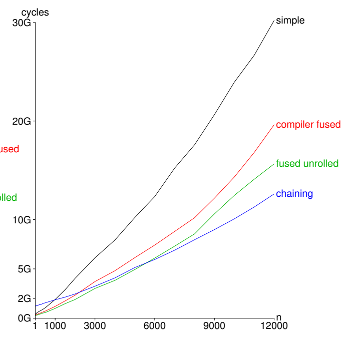
Generate code

```
vdup sf*v vswap vdup sf*v sf+v sfnegatev

                  $24147C0 refs= 0 bytes=16 $24147A0  :14
                  $2414B10 refs= 0 bytes=16 $2415150  :15
sftimesv_  15 15 temporary                            :16
sftimesv_  14 14 temporary                            :17
sfplusv_   16 17 temporary                            :18
sfnegatev_ 18  0 $2415030 refs= 0 bytes=16 $2417300   :19

fused:
 vmovaps (%rdi,%r10,1),%ymm0
 vmulps  %ymm0,%ymm0,%ymm1
 vmovaps (%rsi,%r10,1),%ymm2
 vmulps  %ymm2,%ymm2,%ymm3
 vaddps  %ymm1,%ymm3,%ymm1
 vxorps  %ymm1,%ymm4,%ymm1
 vmovaps %ymm0,(%rdx,%r10,1)
 add     $0x20,%r10
 cmp     %r10,%r9
 ja      fused
```

Evaluation

Multiply 50 $\times$ 50 with 50 $\times$ n Double matrix for varying n, 500 times on Core i5 6600K (Skylake)





Conclusion

- How to use SIMD instructions for data parallelism?
- Manual vectorization, application vector size, opaque vectors
  gives freedom to the compiler/library writer
- Software vector chaining
  Build trace at run-time
  Compile if not cached

+ Can be implemented as library
  315 source lines of code

+ For long vectors $>$ 2$\times$ as fast as *simple*

– High per-operation overhead
  Useful only for long vectors
  Select between *simple* and *chaining* per operation

- `github.com/AntonErtl/vectors`
  Paper at ManLang 2018
  `https://www.complang.tuwien.ac.at/papers/ertl18.pdf`

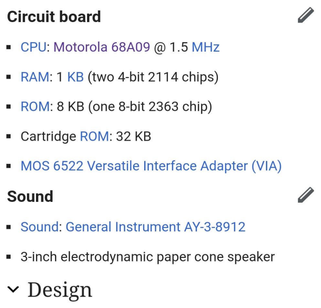
Programming In Forth on the Vectrex – Phillip Eaton 2018

Define Goals

- Get Forth running on Vectrex with interactive terminal
- No Vectrex hardware modification allowed (can't swap out the BIOS)
- Must provide Forth API to the BIOS
- Must be comparatively fast compared with assembler and C, not a toy
- Must be accessible to potential new developers

What is a Vectrex?

https://youtu.be/k8GiErP6Nfc



Circuit board

- CPU: Motorola 68A09 @ 1.5 MHz
- RAM: 1 KB (two 4-bit 2114 chips)
- ROM: 8 KB (one 8-bit 2363 chip)
- Cartridge ROM: 32 KB
- MOS 6522 Versatile Interface Adapter (VIA)

Sound

- Sound: General Instrument AY-3-8912
- 3-inch electrodynamic paper cone speaker

⌄ Design

European release Vectrex playing the built-in game Minestorm, without overlay

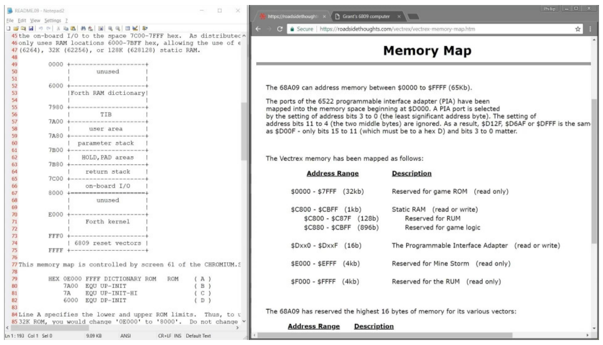
Step 1

- Configure CamelForth For Vectrex and cross compile
- No DOSBox – convert cross compiler from F83 to…. Gforth
- No block source files, need to tweak parser – took a lot of thinking about!

My Background

- Spent 90s programming Z80 SBCs with MPE Forth for SCADA applications
- Collected a lot of classic video arcade games: Space Invaders, Asteroids, Defender
- Spent 2000's in London and Zurich on financial systems
- 2 years ago, acquired a dead Vectrex and fixed it

Setting up camel forth memory map



```
the on-board I/O to the space 7C00-7FFF hex.  As distributec
only uses RAM locations 6000-7BFF hex, allowing the use of e
(6264), 32K (62256), or 128K (628128) static RAM.

        0000 +--------------------+
             |       unused       |
             |                    |
        6000 +--------------------+
             |Forth RAM dictionary|
             |                    |
        7980 +--------------------+
             |        TIB         |
        7A00 +--------------------+
             |     user area      |
        7A80 +--------------------+
             |  parameter stack   |
        7B00 +--------------------+
             |   HOLD,PAD areas   |
        7B80 +--------------------+
             |    return stack    |
        7C00 +--------------------+
             |    on-board I/O    |
        8000 +--------------------+
             |       unused       |
             |                    |
        E000 +--------------------+
             |    Forth kernel    |
             |                    |
        FFF0 +--------------------+
             | 6809 reset vectors |
        FFFF +--------------------+

This memory map is controlled by screen 61 of the CHROMIUM.S

        HEX 0E000 FFFF DICTIONARY ROM   ROM     ( A )
            7A00  EQU UP-INIT                   ( B )
            7A    EQU UP-INIT-HI                ( C )
            6000  EQU DP-INIT                   ( D )

Line A specifies the lower and upper ROM limits.  Thus, to u
32K ROM, you would change '0E000' to '8000'.  Do not change
```

https://roadsidethoughts.com/vectrex/vectrex-memory-map.htm

Memory Map

The 68A09 can address memory between $0000 to $FFFF (65Kb).

The ports of the 6522 programmable interface adapter (PIA) have been mapped into the memory space beginning at $D000. A PIA port is selected by the setting of address bits 3 to 0 (the least significant address byte). The setting of address bits 11 to 4 (the two middle bytes) are ignored. As a result, $D12F, $D6AF or $DFFF is the sam as $D00F - only bits 15 to 11 (which must be to a hex D) and bits 3 to 0 matter.

The Vectrex memory has been mapped as follows:

Address Range	Description
$0000 - $7FFF (32kb)	Reserved for game ROM (read only)
$C800 - $CBFF (1kb)	Static RAM (read or write)
$C800 - $C87F (128b)	Reserved for RUM
$C880 - $CBFF (896b)	Reserved for game logic
$Dxx0 - $DxxF (16b)	The Programmable Interface Adapter (read or write)
$E000 - $EFFF (4kb)	Reserved for Mine Storm (read only)
$F000 - $FFFF (4kb)	Reserved for the RUM (read only)

The 68A09 has reserved the highest 16 bytes of memory for its various vectors:

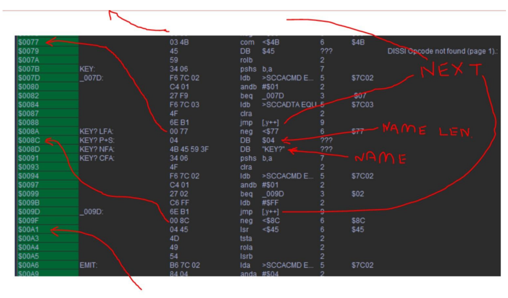
Address Range **Description**

What can I do with it?

- Vibrant home brew community, some amazing programs, hardware hacking
- Memory map and cartridge port simple and open
- I could put CamelForth onto the bare metal 😀
- Challenges: no serial port, don't know 6809 assembler, don't know Vectrex BIOS, don't know low-level Forth

Step 2

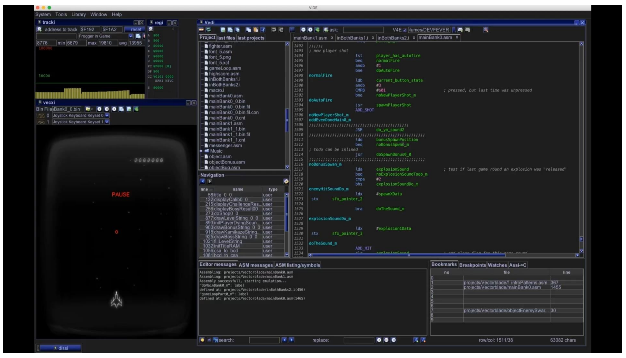
- Debug in VIDE emulator
  - Create label file for debugger
  - Use Starting Forth to learn how code is compiled
  - Will it clash with BIOS?
  - Hack COLD to write to display via BIOS



```
$0077                03 4B        com   <$4B          6    $4B
$0079                45           DB    $45           ???       DISSI Opcode not found (page 1).:
$007A                59           rolb                2
$007B  KEY:          34 06        pshs  b,a           7
$007D  _007D:        F6 7C 02     ldb   >SCCACMD E... 5    $7C02
$0080                C4 01        andb  #$01          2
$0082                27 F9        beq   _007D         3    $07
$0084                F6 7C 03     ldb   >SCCADTA EQU  5    $7C03
$0087                4F           clra                2
$0088                6E B1        jmp   [,y++]        9
$008A  KEY? LFA:     00 77        neg   <$77          6    $77
$008C  KEY? P+S:     04           DB    $04           ???
$008D  KEY? NFA:     4B 45 59 3F  DB    "KEY?"        ???
$0091  KEY? CFA:     34 06        pshs  b,a           7
$0093                4F           clra                2
$0094                F6 7C 02     ldb   >SCCACMD E... 5    $7C02
$0097                C4 01        andb  #$01          2
$0099                27 02        beq   _009D         3    $02
$009B                C6 FF        ldb   #$FF          2
$009D  _009D:        6E B1        jmp   [,y++]        9
$009F                00 8C        neg   <$8C          6    $8C
$00A1                04 45        lsr   <$45          6    $45
$00A3                4D           tsta                2
$00A4                49           rola                2
$00A5                54           lsrb                2
$00A6  EMIT:         B6 7C 02     lda   >SCCACMD E... 5    $7C02
$00A9                84 04        anda  #$04          2
```

https://youtu.be/t4lwoWBXPhA

Vectrex IDE



Other little videos

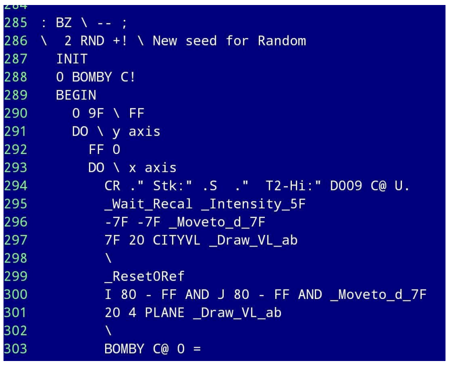
- City Bomber – the basics of a game
  https://youtu.be/wbV4a56reNA

- Interactive test to discover what BIOS Wait_Recal function does
  https://youtu.be/yWUVZyadA0w

Step 3

- No serial port. Time to get hands dirty now…enter VecFever
- Rewrote EMIT, KEY?, KEY for soft UART
- Unhack COLD
- Try it out…

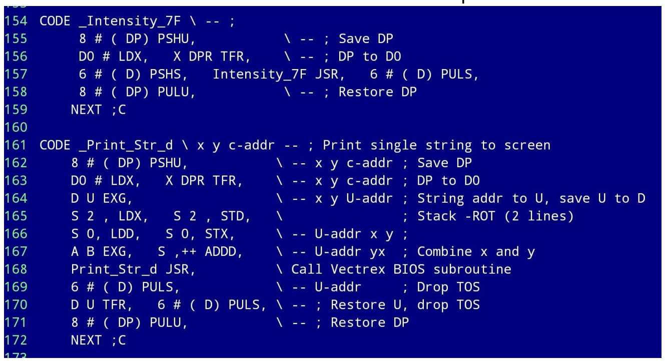
https://youtu.be/FhHfR9zPggg



```
: BZ \ -- ;
\  2 RND +! \ New seed for Random
  INIT
  0 BOMBY C!
  BEGIN
    0 9F \ FF
    DO \ y axis
      FF 0
      DO \ x axis
        CR ." Stk:" .S  ."  T2-Hi:" D009 C@ U.
        _Wait_Recal _Intensity_5F
        -7F -7F _Moveto_d_7F
        7F 20 CITYVL _Draw_VL_ab
        \
        _ResetORef
        I 80 - FF AND J 80 - FF AND _Moveto_d_7F
        20 4 PLANE _Draw_VL_ab
        \
        BOMBY C@ 0 =
```

Game main loop – not optimized or factored!

Forth interface to Vectrex BIOS – no optimization!



```
CODE _Intensity_7F \ -- ;
     8 # ( DP) PSHU,            \ -- ; Save DP
     D0 # LDX,   X DPR TFR,     \ -- ; DP to D0
     6 # ( D) PSHS,   Intensity_7F JSR,   6 # ( D) PULS,
     8 # ( DP) PULU,            \ -- ; Restore DP
    NEXT ;C

CODE _Print_Str_d \ x y c-addr -- ; Print single string to screen
    8 # ( DP) PSHU,            \ -- x y c-addr ; Save DP
    D0 # LDX,   X DPR TFR,     \ -- x y c-addr ; DP to D0
    D U EXG,                   \ -- x y U-addr ; String addr to U, save U to D
    S 2 , LDX,   S 2 , STD,    \               ; Stack -ROT (2 lines)
    S 0, LDD,   S 0, STX,      \ -- U-addr x y ;
    A B EXG,   S ,++ ADDD,     \ -- U-addr yx  ; Combine x and y
    Print_Str_d JSR,           \ Call Vectrex BIOS subroutine
    6 # ( D) PULS,             \ -- U-addr     ; Drop TOS
    D U TFR,   6 # ( D) PULS, \ -- ; Restore U, drop TOS
    8 # ( DP) PULU,            \ -- ; Restore DP
    NEXT ;C
```