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
code >r ( x -- ) ( R -- x )
        pop ebx
        lea ebp,[ebp-4]
        mov [ebp], ebx
        next
;
```

```
\ C
code >r ( x -- ) ( R -- x )
   *++rp=*sp--
;
```

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