33rd EuroForth Conference

September 8-10, 2017

College Garden Hotel
Bad Vöslau
Austria

Preface

EuroForth is an annual conference on the Forth programming language, stack machines, and related topics, and has been held since 1985. The 33rd EuroForth finds us in Bad Vöslau (near Vienna) for the first time. The two previous EuroForths were held in Bath, England (2015), and on Reichenau Island, Germany (2016). 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 all were accepted (100% acceptance rate). For more meaningful statistics, I include the numbers since 2006: 24 submissions, 17 accepts, 71% 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 all three submissions were from program committee members, and one of them from 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.

These online proceedings (`http://www.euroforth.org/ef17/papers/`) also contain papers and presentations that were too late to be included in the printed proceedings. In addition, you can find videos (also for presentations without paper or slides in the proceedings) on the proceedings website.

Workshops and social events complement the program. This year's EuroForth is organized by Gerald and Claudia Wodni

Anton Ertl

Program committee

Sergey N. Baranov, SPIIRAS, Russia
M. Anton Ertl, TU Wien (Chair)
Ulrich Hoffmann, FH Wedel University of Applied Sciences (Secondary Chair)
Phil Koopman, Carnegie Mellon University
Jaanus Pöial, Tallinn University of Technology
Bradford Rodriguez, T-Recursive Technology
Bill Stoddart
Reuben Thomas, SC3D Ltd.

Contents

A Formal Language Processor Implemented in Forth

Sergey N. Baranov

***Abstract*—**The structure of a Forth program is described which implements a language processor for an ALGOL-like programming language with its context-free component belonging to the class LL(1). It allows to check that a program in the given formal language is syntactically correct as well as to convert a correct program into a pseudo-code for a simple interpreter to interpret it and thus simulate the program behavior in a certain environment. The ultimate goal of this work is to build a tool for running experiments with programs in the Yard language which formally describes the behavior of multi-layer artificial neural networks on the principles of a machine with dynamic architecture (MDA) and due to that has a number of specific language constructs. The tool is assumed to run on a PC under MS Windows and is based on the system VFX Forth for Windows IA32 which implements the Forth standard Forth 200x of November 2014 (the so called Forth 2014).

***Keywords*—**formal languages, language processor, parser, regular expressions, Forth.

I. Introduction

AUTOMATED analysis of formal languages started in the 1960s. Various tools were developed for processing both context free and context dependent formal languages to be studied with computer machinery. To-day, data processing technologies are widely used in translation systems for a variety of computer devices with many applications to support them. E.g., the Flex/Bison parsers ([1], [2], [3]) are often used to quickly obtain a particular language processor from a formal definition of a language. The tool ANTLR [4] is the next step in developing language processors and is based on principles close to those of this paper. However, these mentioned tools constrain the type of the input formal grammar, and the employed algorithm often requires enormous memory for storing intermediate data. Moreover, their usage requires understanding their special language for representing a formal grammar of the considered programming language and code generation of the recognized program is out of scope of those useful tools.

This paper is aimed at developing a flexible and "pocket-like" inexpensive tool based on the current Forth 2014 standard [5] for experimenting with new formal languages. Usage of Forth provides the necessary flexibility and unlimited freedom [6] in designing the respective definitions while modern computing machinery eliminates a lot of memory and speed limitations of early 1960s. The tool should also allow for experiments with code generation for correct programs and simulation of their execution on some hardware platform (code generation and simulation will be the scope of future work based on the already obtained results).

Any ALGOL-like programming language may be considered as a set of chains composed from the lexical units of this language (the so called language lexems or terminals) according to the rules of the language grammar which splits into its context-free component and a number of context dependent rules or constraints [7]. To use the proposed technique, the context-free component of the language under consideration should be specified in the formalism of regular expressions [8], [9] built from the language terms (terminals, non-terminals, and expressions in them) using three classical operations: concatenation (sequencing), alternative choice (branching), and recursion (cycling). Initial terms are language lexems (terminals) directly recognizable in a program text, non-terminals denoting language constructs built from terminals, and an empty string denoting an absence of a lexem. Context dependencies are specified informally, they are checked through a mechanism of semantics – special procedures invoked by the language processor in the process of input text parsing and which exchange their data via a stack or global variables.

The text in the given programming language to be processed is contained in a simple text file considered as a sequence of symbols (characters) in the 8 bit ASCII coding. The following 4 classes of characters are distinguished:

1) 96 skip characters (space, tabulation and other "invisible" symbols);

2) 10 digits – characters which represent decimal digits , from "0" to "9";

3) 119 letters – letters of the Latin (52) and Russian (66) alphabets, both in the upper and lower case, as well as the underscore symbol "_";

4) 31 special characters – "!", "'", "#", "$", "%", "&", "'", "(", ")", "*", "+", ",", "-", ".", "/", ":", ";", "<", "=", ">", "?", "@", "[", "\", "]", "^", "`", "{", "|", "}", "~".

Thus a complete set of 96+10+119+31=256 8-bit character codes is obtained.

The text under processing may contain comments (see section IV) which are skipped and treated as a single space.

II. Sample Programming Language

Further narrative will be illustrated with examples in the OCC language – a subset of C for developing programs to run on a special kind of hardware [10].

There are a number of lexems (terminals) in any programming language, denoted in this paper with symbols in single quotes (apostrophes). Though their particular nomenclature may be different in different languages, one can always divide it in 3 groups:

S. N. Baranov for over 10 years was with Motorola ZAO, St. Petersburg, Russia. He is now with the St. Petersburg Institute for Informatics and Automation of the Russian Academy of Sciences, 14 linia 39, St. Petersburg, 199178, Russia (phone: 812-328-0887; fax: 812-328-4450; e-mail: SergeyBaranov@gmail.com).

1) lexems of the general mode (distinguished be angle brackets "<" and ">") in their denotation):

• '<finish>' – a special terminus lexem with no external representation;

• '<float>' – denotation of a floating point number in the decimal system in accorance with the ANSI/IEEE 754-1985 standard in the format: <d><d>*.<d>*[{E|e}[+|-]<d><d>*] – a number starts with a digit; there is always a period as a delimiter between the mandatory integer and a possible fraction in the significand; if present, the exponent begins with the Latin character "E" or "e", which may be followed by plus ("+") or minus ("-"), followed by at least one digit – examples are 3.14, 10.E-6, 123.456e3, 5.E+1;

• '<number>' – denotation of an integer in binary, octal, decimal (by default), or hexadecimal, the radix being denoted by letters "B" or "b" (binary), "C" or "c" (octal), "D" or "d" (decimal), and "H" or "h" (hexadecimal); in the latter case the first six letters of the Latin alphabet A..F in any case are used as digits after 0..9 (if the first significant digits is one of these letters, then it should be preceded by the digit 0) – examples are 101B, 111111B (binary), 777C, 100C (octal), 0, 125, 126D (decimal), 0ffH, 0AH, 0DH, 1000H (hexadecimal);

• '<operation>' – an operation sign composed by one or two special symbols: '-', '#', '*', '/', '/\', '/=', '\/', '^', '~', '+', '< ', '<=', '=', '=<', '=>', '>', '>='; each of these 17 operations (except for '~') is dyadic, two of them ('-' and '+') are both dyadic and monadic; and '~' is monadic; their order of execution in compound expressions is determined by their priorities (all monadic operations have the highest priority 10):

Operation	Name	Priority		Operation	Name	Priority
'~'	Negation	10		'/='	Not equal	5
'^'	Power	9		'<'	Less than	5
'=<'	Left shift	8		'<='	Less than or equal	5
'=>'	Right shift	8		'='	Equal	5
'*'	Multiply	7		'>'	Greater than	5
'/'	Divide	7		'/\'	And	4
'-'	Subtract	6		'#'	Exclusive Or	3
'+'	Add	6		'\/'	Or	3
'>='	Greater than or equal	5				

• '<start>' – a special initial lexem with no external representation;

• '<string>' – a denotation of a character string in double quotes ("), the string length – the number of contained characters – may be from 0 (empty string) up to 80 (maximal length); a double quote itself as a string character is denoted by two double adjacent double quotes; examples are "abc", "" (empty string), "a""bc" (a quote inside the string);

• '<tag>' – an identifier; i.e., a sequence of letters (including underscore) and digits beginning with a letter. the total number of characterd not exceeding 80; examples are abcd, x1, _great;

2) lexems – reserved words: 'auto', 'break', 'case', 'char', 'const', 'continue', 'default', 'do', 'double', 'else', 'enum', 'extern', 'float', 'for', 'goto', 'if', 'int', 'long', 'register', 'return', 'short', 'signed', 'static', 'struct', 'switch', 'typedef', 'union', 'unsigned', 'void', 'volatile', 'while' – these lexems correspond to words contained in their denotations;

3) indicants denoted by one two adjacent special characters: '(', ')', ',', '.', ':=', ';', '[', ']', '{', '}' – brackets of three kinds (round, square, and curly), assignation sign composed by a colon and an equal sign, comma, period, and semicolon.

Each lexem, except for '<start>' и '<finish>', enjoy an external representation in the source program text; some lexems may have several representations (e.g., the lexem '<=' may be represented with an indicant "<=" and the word "le"), and some may have an unlimited number of representations (like '<number>', '<string>' or '<tag>').

Three lexems of the general type are characterized with additional parameters. The lexem '<float>' (denotation of a floating point number) and '<number>' (denotation of an integer) have the respective value as their parameters, while '<operation>' (operation) has two additional parameters – operation priority and its name (add, multiply, etc.)

The following non-terminals are distinguished in the language description to denote particular language structures: abstr_decl, abstr_decl1, compound, declaration, declarator, declarator1, enumm, expression, formula, init, operand, param, paramlist, pointer, program, specif, statement, struct_decl. The non-terminal named program is usually the initial non-terminal which any syntactically correct text is generated from. The following semantics whose names begin with "$" occur in the grammar rules to take into account non-formal context dependencies: $array1, $array2, $arrow, $aster, $brk, $call1, $call2, $call3, $case1, $case2, $char, $comp1, $comp2, $cond, $cond1, $cond2, $cont, $decl, $default, $do1, $do2, $dot, $else, $expr, $field, $finish, $for1, $for2, $for3, $for4, $for5, $goto, $ident, $if, $incr1, $incr2, $init, $label, $mem, $null, $number, $op1, $op2, $opcode1, $opcode2, $operand, $qual, $ret1, $ret2, $start, $stmnt, $string, $sw1, $sw2, $sw3, $tag, $then, $type, $wl1, $wl2. Each semantic usually denotes a certain procedure which is executes if in the process of input text parsing the current recognized lexem follows this semantic the in a grammar rule, the semantic enjoying access to all lexem parameters.

Grammar rules are formulated in the following way:

```
Non-terminal : Regular_expression .
```

Non-terminal being one of the non-terminal denotations enlisted above, and being specified in accordance with the following syntax of the Naur-Backus formalism:

Regular expression ::= {	
Empty | Lexem | Non-terminal | Semantic |	Basic elements (1)
Regular_expression Regular_expression |	Concatenation (2)
Regular_expression ';' Regular_expression |	Alternatives (3)
Regular_expression '*' Regular_expression |	Recursion (4)
'(' Regular_expression ')' }.	Expression in brackets (5)

Alternatives are enumerated in curly brackets separated with a vertical bar. In the section marked (1) basic elements of a regular expression are enumerated: `Empty` (an empty expression denoting absence of anything), `Lexem`, `Non-terminal`, and `Semantic`. Section (2) represents a concatenation which has no particular denotation, section (3) is for alternative choice with a semicolon as the operation sign, section (4) stands for recursion with an asterisk as its operation sign, and the last section (5) allows to embrace a regular expression in round brackets and thus to consider it as one operand in operations of concatenation, alternative choice, and recursion.

For better visibility of an alternative choice between a regular expression `A` and an empty alternative: `( A ; )` is denoted as `A` in square brackets: `[ A ]` (pronounced as "possible `A`") , as well as for a recursions with an empty left or right operand `*A` and `A*` are denoted as `*( A )` and `( A )*` respectively, while a recursion with both non-empty operands `A*B` is denoted as `( A )*( B )`.

With the above denotations a grammar of the OCC language may be specified through the following regular expressions:

```
abstr_decl : pointer [ abstr_decl1 ] ;
             abstr_decl1 .
abstr_decl1 : ( '(' abstr_decl ')' )*(
              '[' [ formula ] ']' ;
                 '(' [ paramlist ] ')' )  .
compound : $comp1 '{' *( declaration $decl )
       *( statement $stmnt ) $comp2 '}' .
declaration : ( specif )*  ( declarator
      [ $init ':=' init ] )*( ',' ) ';' .
declarator : [ pointer ] declarator1 .
declarator1 : ( $tag '<tag>' ;
         $tag '<label>' ; '(' declarator
        ')' )*( ( '[' [ formula ] ']' )* ;
          '(' (  ( $tag '<tag>' )*( ',' ) ;
                 paramlist ;
               ) ')'          ) .
enumm : '<tag>' [ ':=' expression ] .
expression : ( formula )*( ',' ) .
formula : ( *( $opcode1 '<operation>' )
    operand $operand )*(
    $opcode2 ':=' ; $opcode2 '<operation>' ) .
init : formula ; '{' ( formula ; '.' '.' '.'
               )*( ',' ) '}' .
operand : ( ( $incr1 '<increment>'
       $tag ( '<tag>' ; '<label>' ) ;
       $tag '<tag>' ( $incr2 '<increment>' ;
       ( $array1 '[' formula $array2 ']' )* ;
       $call1 '(' [ expression $call2 ]
              $call3 ')' ;  $ident ) ;
         $op1 '(' expression $op2 ')' )
   [ ( $dot '.' ; $arrow '->' ) $field (
     '<tag>' ; '<label>' ) ] ;
    $tag '<label>' ;
    $number '<number>' ;  $char '<char>' ;
    $string '<string>'
   )*( $cond1 '?' expression $cond2 ':' ) .
param : specif ( declarator ;
        '[' [ formula ] ']' ) .
paramlist : ( param ; '.' '.' '.' )*( ',' ) .
pointer : ( $aster '<operation>'
         *( $qual 'const' ;
            $qual 'volatile' ) )* .
program : $start '<start>'
  ( ( specif )*( ';' ;
     declarator [ compound ]
       ( ';' ;
     ':=' init *( ',' declarator
          [ ':=' init ] ) ';' ;
       ',' ( declarator [ ':=' init ]
         )*( ',' ) ';'
        *( declaration ) compound
           ) ) ;
      declarator *( declaration ) compound
    )* $finish '<finish>' .
specif : ( $mem 'auto' ; $mem 'register' ;
         $mem 'static' ;
         $mem 'extern' ; $mem 'typedef' ;
         $type 'void' ; $type 'char' ;
         $type 'short' ;
         $type 'int' ; $type 'long' ;
         $type 'float' ;
         $type 'double' ; $type 'signed' ;
         $type 'unsigned' ;
    ( 'struct' ; 'union' ) [ '<tag>' ] [ '{' (
          struct_decl )*
     '}' ] ; 'enum' ( '<tag>' [ '{' ( enumm
       )*( ',' ) '}' ] ;
      '{' ( enumm )*( ',' ) '}' ) ;
      $qual 'const' ; $qual 'volatile'  )* .
statement : *( $label '<label>' ':' ;
      $case1 'case' expression
      $case2 ':' ; $default 'default' ':' )
      ( compound ;
        'if' $cond '(' expression $then ')'
             statement
         [ $else 'else' statement ] $if ;
        'switch' $sw1 '(' expression $sw2 ')'
            statement $sw3 ;
        'while' $cond '(' expression $wl1 ')'
                statement $wl2 ;
   $do1 'do' statement 'while' $cond '('
              expression ')' $do2 ;
        'for' $for1 '(' [ expression ]
              $for2 ';' [ expression ]
         $for3 ';' [ expression ] $for4 ')'
            statement $for5 ;( 'goto' $goto
'<tag>' ; $cont 'continue' ; $brk 'break' ;
  $ret1 'return' [ expression ] $ret2  ;
         expression $expr ; $null ) ';' ) .
struct_decl : specif ( declarator [ ':'
        expression ] )*( ',' ) ';' .
```

The initial non-terminal in this grammar is `program`.

This grammar representation is nothing more than a linear record of a syntactic graph of this language for recognizing

correctly constructed chains composed from its lexems, and it may be considered as a text in Forth (that's why its elements are separated with spaces) which in its turn may represent a Forth program if the respective word definitions are provided. Therefore, the development of a language processor which recognizes this programming language consists in development of these word definitions.

III. Language Processor Structure

The language processor works as follows.

1. An instrumental Forth system compliant with the Forth 2014 standard (e.g., VFX Forth for Windows IA32 [11] or gFORTH for Windows [12]) is launched on a working PC under MS Windows, and the respective Forth text with word definitions is compiled.
2. After successful compilation of this Forth text which establishes the necessary context, another Forth text with the language grammar is compiled which thus is transformed into a parser program. Successful compilation of the grammar is terminated with the message "`Lexical Analyzer successfully compiled!`" from the instrumental Forth system.
3. Upon successful completion of the step 2, the source text in the programming language is submitted as input to the parser for analysis and upon successful completion of parsing a pseudo-code of the submitted program is built in an output file for subsequent execution. If a syntax error in the input text were found or any exception occurred (like file system error, buffer overflow etc.) then a respective error message is produces and the parser terminates processing.

The language processor has two major components: the lexical analyzer (scanner) and the syntactic analyzer (parser), each of them may be considered as a separate software product.

In the current version, the scanner consists of 62 Forth definitions with the total size of 368 LOCs (source lines of code in Forth, excluding empty lines and lines with comments only) and the parser has 73 definitions of the total size 431 LOCs; thus the total size of the scanner and parser is 799 LOCs including the OCC grammar of 135 LOCs. The grammar graph of the OCC language in its internal representation occupies 1391 cells (machine words).

For scanner and parser testing, the respective test wrappers and test suites were developed. Exceptions which occur when running these programs are processed through the Forth interruption mechanism of `CATCH-THROW`, where the word `CATCH` receives an address of the message string formed by the scanner or parser when the exception is recognized and passed by `THROW`. Exceptions recognized while compiling these two components terminate compilation through the word `ABORT"` with a respective error message.

IV. Lexical Analyzer

The lexical analyzer (scanner) converts the input source text into a sequence of numeric values denoting lexems (terminals) of the programming language, recognized in the input text in the order of their occurrences in this text. The scanner is implemented as two co-programs: the low-level `GetChar` and the upper-level `GetLex`. The former sequentially consumes and filters characters from the input text skipping all irrelevant characters and returning the next significant character upon a request, while the latter forms the next lexem from characters obtained at the low level. Both co-programs work with "looking ahead" at 1 element – a character in case of `GetChar` and a lexem in case of `GetLex`. Results of each invocation of these co-programs are returned in a respective pair of variables: {`CurrChar`, `NextChar`} in case of `GetChar`, and {`CurrLex`, `NextLex`} in case of `GetLex`. The first variable in the pair contains the value (character or lexem) returned upon the given request to respective co-program, and the second variable in the pair contains the value to be returned upon the next request addressed to this co-program.

A list of lexems is specified by the defining word `LexClasses` created to build definitions of all lexems in the word list `Lexems`. A lexem is represented with its execution token which never executed during compilation of the lexical analyzer, while during parsing this code checks whether the current value of the variable `CurrLex` is the execution token of this lexem and if this is the case, the current lexem is "accepted" and the parser proceeds to considering the next lexem from its input stream; otherwise, this lexem signals the parser about its failure to accept the current lexem.

Along with lexems, the scanner recognizes two kinds of comments in the input: a) from two adjacent slashes ("`//`") up to the end of line; and b) from a slash and an asterisk ("`/*`") up to an asterisk and a slash ("`*/`"), as is common in many C realizations, unless these two character combinations marking the beginning of a comment are not inside a string denotation. At the formal level, a comment is treated as a space character.

The initial value of the variable `NextChar` is a space, and that of the variable `NextLex` is the lexem `'<start>'`. Upon reaching the end of the input file, the co-program `GetChar` returns a special character `<eof>` with no external representation and denoting the end-of-file. From that moment this character is returned by `GetChar` in response to all further requests for the next character. When consumed by the co-program `GetLex` this character transforms in a special lexem `'<finish>'` returned by `GetLex` in response to all further requests for the next lexem.

A test wrapper for the scanner provides three kinds of testing: getting text lines from the input file, getting characters from the input file, and getting lexems. They are specified by test words `TestGetLine"`, `TestGetChar"`, and `TestGetLex"` which get the name of the input file from the input stream and subsequently extract lines, characters, and lexems from it until the input file is exhausted. The word `Test"` is an extension of `TestGetLex"` – it establishes an interrupt handler through `CATCH` and executes `TestGetLex"` in this context.

```
: Test" ( "<chars>name<quote>"--)
 -1 ?Echo !
```

```
['] TestGetLex" CATCH ?DUP
IF
    CR ." Exception: " COUNT TYPE
    CR CloseInFile
THEN ;
```

Fig. 1 displays excerpts from a log of running `GetLex` in the test wrapper `LA_TestWrapper` which demonstrates the work of the scanner. Excerpts are separated by dotted lines.

```
include
c:\yard\LA_TestWrapper.fth
Including
c:\yard\LA_TestWrapper.fth
Including C:\yard\LA34.fth
ok
Test" c:\yard\words31.txt
Yard version:
C:\yard\LA34.fth
Test run on 30.06.2017 at
11:21:30
001 // Identifiers<eol>
002 abcd  x1 _great<eol>
'<START>'
'<TAG>'  repr=abcd
'<TAG>'  repr=x1
003 <eol>
004 // Integers<eol>
005 0 101B  111111B /*
Binary */<eol>
'<TAG>'  repr=_great
'<NUMBER>'  repr=0 value=0
'<NUMBER>'  repr=101B
value=5
006 777C 100C /* Octal
*/<eol>
'<NUMBER>'  repr=111111B
value=63
'<NUMBER>'  repr=777C
value=511
007 125 126D /* Decimal
*/<eol>
'<NUMBER>'  repr=100C
value=64
'<NUMBER>'  repr=125
value=125
008 0ffH 0AH 0DH 1000H /*
Hexadecimal */<eol>
'<NUMBER>'  repr=126D
value=126
.........................
.........................
014 static struct switch
typedef union<eol>
'SIGNED'  repr=signed
'STATIC'  repr=static
'STRUCT'  repr=struct
'SWITCH'  repr=switch
'TYPEDEF'  repr=typedef
015 unsigned void volatile
while <eol>
'UNION'  repr=union
'UNSIGNED'  repr=unsigned
.........................
020 ~ not ^ ** * / + - = < >
# xor /\ & \/ | or >= /= <=
=> =< shl shr le ge <eol>
'WHILE'  repr=while
'<OPERATION>'  repr=~ op=~
prio=10
.........................
025 <eol>
026 // Strings<eol>
027 "abc"<eol>
'--'  repr=-
028 "" /* Empty string
*/<eol>
'<STRING>'  repr=abc
length=3
029 """abc" "a""bc" "abc"""
/* Quote in various places
*/<eol>
'<STRING>'  repr= length=0
'<STRING>'  repr="abc
length=4
'<STRING>'  repr=a"bc
length=4
030 <eol><eof>
'<STRING>'  repr=abc"
length=4
'<FINISH>'
 ok
```

Fig. 1. A log of a scanner test run

Three digit numbers in the beginning of a line are the input text line numbers; the text lines are included in the log as they are read-in by the co-program `GetChar`. There are 30 such lines in this example. Each line terminates with the symbol `<eol>` which marks its end-of-line, while the symbol `<eof>` marks the end of the input file.

The log contains denotations of the recognized lexems followed be their external representation in the input file (after the key word "`repr=`") and additional parameters of this lexem if any with appropriate key words.

V. Syntactic Analyzer

The syntactic analyzer (parser) is built on-top of the lexical analyzer. Its main (starting) word is `OCC"` which obtains the name of the input file with the program text to be analyzed and checks whether this text complies with the grammar of the programming language OCC. As with the scanner, the test wrapper of the parser contains the word `Test"` which establishes an interrupt handler and initiates execution of the main parser word in this context.

The parser main word is created through the defining word `Grammar`. It starts grammar definition of the considered programming language in form of a series of generating rules for its non-terminals. The grammar ends with the closing word `EndGrammar` which identifies the initial non-terminal:

```
: Grammar    ( "<spaces>name"--123)
  CREATE  ALIGN HERE ( pfa)
    DUP GrammarGraph ! \ Grammar graph start
    ['] (CallNT) , 0 ,  HERE CELL+ CELL+ ,
    ['] (Success) , ['] (Fail) ,
.....................................
: EndGrammar ( "<spaces>name" addr 123 --)
  ( pfa) ' ( pfa xt-inital) >BODY @
  SWAP  CELL+ ! 0 ,
  CR ." Lexical Analyzer successfully
compiled!" ;
```

The Forth interpreter of the underlying Forth system ensures execution of the source grammar text as a text in Forth resulting in construction of a grammar graph for the given formal language which consists of elements of several kinds. The parser main word created by the defining word `Grammar` provides traversal of this graph controlled by the variable `Pnt`, pointing to its next element. The return stack `Return` with operations `Push` and `Pop` and the queue `SemanticsQueue` of semantics whose execution is delayed till accepting the current terminal, are used as auxiliary data structures. Executable codes are denoted with words in brackets. In total, 9 kinds of elements are provisioned for a grammar graph:

1. Jump at the address `addr` – 2 cells: `(Jump)|addr`
```
  : (Jump) Pnt @ @ Pnt ! ;
```
2. Call of a non-terminal at the address `addr` – 3 cells: `(CallNT)| addr |failaddr`
```
     : (CallNT) Pnt @ Return Push (Jump) ;
```
3. Starting a non-terminal `xt` – 2 cells: `(StartNT)|  xt`
```
  : (StartNT) CELL Pnt +! ;
```

if zero is specified instead of `xt` then this is an auxiliary non-terminal created automatically with no name.

4. Successful completion of a non-terminal – 1 cell: `(ExitNT)`
```
   : (ExitNT) Return Pop CELL+ CELL+ Pnt ! ;
```
5. Unsuccessful completion of a non-terminal – 1 cell: `(FailNT)`
```
   : (FailNT) Return Pop CELL+ Pnt ! (Jump) ;
```
6. Passing a semantic `xt` – 2 cells: `(Semantic)|  xt`
```
  : (Semantic) Pnt @
       SemanticsQueue Push  CELL Pnt +! ;
```
7. Passing a lexem `xt` – 3 cells: `(Lexem)|xt |failaddr`
```
  : (Lexem) CurrLex @ pnt @ @ =
   IF \ accept the current lexem:
     Pnt @ CELL+ CELL+ Pnt !
   ELSE \ reject the current lexem:
     CELL Pnt +!  (Jump) THEN ;
```
8. Successful completion of parsing – 1 cell: `(Success)`
```
 : (Success) ( --) CR ." Success!" 1 THROW ;
```
9. Unsuccessful completion of parsing – 1 cell: `(Fail)`
```
  : (Fail) ( --) CR ." Compilation Failed!"
 .......... \ Form a message in MessageBuf
             MessageBuf THROW ;
```

the scanner reports through `MessageBuf` which lexem was the current one and what other lexems were checked for it in form of the message: "`Lexem <name> is unexpected; possible options are: <list of lexem names>`".

The above list of 9 element kinds is complete for the following reasons. Elements `(Success)` and `(Fail)` are necessary because these are all possible outcomes of the parsing process (excluding its abnormal terminations through `ABORT` or exceptions). Elements `(CallNT)`, `(Semantic)`, and `(Lexem)` are inevitable as they match all grammar basic elements. Execution of a non-terminal may terminate either successfully or with a failure; therefore, two different exits `(ExitNT)` and `(FailNT)` should be provisioned as well. And finally, `(StartNT)` and `(Jump)` are needed to start a non-terminal body and to jump around it in a linear code of a grammar graph. Thus, totally 2+3+2+2=9 different kinds of elements are needed and this seems to be a sufficient minimum (as the number of Muses[1] is).

An initial value – the address of a five cell element "invoking the initial non-terminal"

```
(CallNT)| addr |failaddr|(Success)|(Fail)
```

of the given grammar is assigned to the variable `Pnt`, `addr` being the starting address of a series of elements for the initial non-terminal of the grammar, and `failaddr` being the address of the next cell but one which contains a reference to the code `(Fail)` while the previous cell contains a reference to the code `(Success)` (see items 8 and 9 above which occur in the grammar graph only once in this five cell element).

The parser provides an option to print-out the grammar graph with by the word `.DisplayCode` – see Fig. 2 below. Similar to Fig. 1, excerpts from the OCC grammar graph representing its beginning and end are separated by a dotted line. One can see that the whole graph occupies only 5564/4=1391 cells.

```
GrammarGraph @ .DisplayCode
0000 (CALLNT)  2848 0016
0012 (SUCCESS)
0016 (FAIL)
0020 (STARTNT)  ABSTR_DECL
0028 (CALLNT)  2700 0060
0040 (CALLNT)  0080 0052
0052 (JUMP)  0072
0060 (CALLNT)  0080 0076
0072 (EXITNT)
0076 (FAILNT)
0080 (STARTNT)  ABSTR_DECL1
0088 (JUMP)  0148
0096 (STARTNT)  Noname
0104 (LEXEM)  '('  0144
0116 (CALLNT)  0020 0144
0128 (LEXEM)  ')'  0144
0140 (EXITNT)
0144 (FAILNT)
0148 (CALLNT)  0096 0264
0160 (LEXEM)  '['  0204
0172 (CALLNT)  1216 0184
0184 (LEXEM)  ']'  0204
0196 (JUMP)  0240
0204 (LEXEM)  '('  0260
0216 (CALLNT)  2560 0228
0228 (LEXEM)  ')'  0260
0240 (CALLNT)  0096 0260
0252 (JUMP)  0160
0260 (EXITNT)
0264 (FAILNT)
0268 (STARTNT)  COMPOUND
0276 (SEMANTIC)  $COMP1
0284 (LEXEM)  '{'  0376
0296 (CALLNT)  0380 0324
0308 (SEMANTIC)  $DECL
0316 (JUMP)  0296
0324 (CALLNT)  4416 0352
0336 (SEMANTIC)  $STMNT
0344 (JUMP)  0324
0352 (SEMANTIC)  $COMP2
0360 (LEXEM)  '}'  0376
0372 (EXITNT)
0376 (FAILNT)
0380 (STARTNT)  DECLARATION
0388 (JUMP)  0424
0324 (CALLNT)  4416 0352
0336 (SEMANTIC)  $STMNT
0344 (JUMP)  0324
0352 (SEMANTIC)  $COMP2
0360 (LEXEM)  '}'  0376
0372 (EXITNT)
0376 (FAILNT)
0380 (STARTNT)  DECLARATION
0388 (JUMP)  0424
0396 (STARTNT)  Noname
0404 (CALLNT)  3440 0420
0416 (EXITNT)
0420 (FAILNT)
0424 (CALLNT)  0396 0584
0436 (CALLNT)  0396 0456
0448 (JUMP)  0436
0456 (JUMP)  0524
0464 (STARTNT)  Noname
0472 (CALLNT)  0588 0520
0484 (SEMANTIC)  $INIT
0492 (LEXEM)  ':='  0516
0504 (CALLNT)  1384 0516
0516 (EXITNT)
0520 (FAILNT)
0524 (CALLNT)  0464 0584
0536 (LEXEM)  ','  0568
0548 (CALLNT)  0464 0568
.........................
5424 (STARTNT)  STRUCT_DECL
5432 (CALLNT)  3440 5564
5444 (JUMP)  5504
5452 (STARTNT)  Noname
5460 (CALLNT)  0588 5500
5472 (LEXEM)  ':'  5496
5484 (CALLNT)  1120 5496
5496 (EXITNT)
5500 (FAILNT)
5504 (CALLNT)  5452 5564
5516 (LEXEM)  ','  5548
5528 (CALLNT)  5452 5548
5540 (JUMP)  5516
5548 (LEXEM)  ';'  5564
5560 (EXITNT)
5564 (FAILNT)   ok
```

Fig. 2. The beginning and the end of the OCC grammar graph

VI. Conclusion

The described implementation of a scanner and a parser in Forth turned out to be quite flexible and powerful. Its main part was borrowed from earlier author's development [10] and ported from Forth-83 to the Forth 2014 standard [5] with minor changes. However, it required reworking and redeveloping the grammar interpreter in order to avoid direct references to the internal structure of the definitions prohibited by Forth 2014. The tool runs on a PC under MS Windows and was developed using the system VFX Forth [11] which supports Forth 2014.

Another problem yet to be solved with the proposed analyzer is checking the input grammar for its correctness; i.e., that it really belongs to the class LL(1) and contains no undesirable recursions. This problem was successfully overcome in [13], so the tool may reuse the found solution.

Future work will consist in developing a pseudo-code generator and an its interpreter to simulate execution of programs in the considered programming language.

[1] 'The thrice three Muses mourning for the death
Of Learning late deceas'd in beggary'
That is some satire, keen and critical. (W. Shakespeare, "A Midsummer Night's Dream", Act 5, Scene 1, 52-54).

References

[1] M. E. Lesk, E. Schmidt, "Lex – A Lexical Analyzer Generator", web: http://dinosaur.compilertools.net/lex/ (2017).

[2] "Win flex-bison", web: http://sourceforge.net/projects/winflexbison/ (2017).

[3] "GNU Bison", web: http://www.gnu.org/software/bison/ (2017).

[4] Terence Parr, "ANTLR (ANother Tool for Language Recognition)", web: http://www.antlr.org/ (2017).

[5] "Forth 200x", web: http://www.forth200x.org/forth200x.html (2016)

[6] L. Brodie, *Thinking Forth*. Punchy Pub, 2004.

[7] A. Aho, R. Sethi, J. Ullman, *Compilers: Principles, Techniques and Tools*. Addison-Wesley (1986).

[8] Jeffrey E.F. Friedl, *Mastering regular expressions*. O'Reilly Media, Inc., 2002.

[9] B. K. Martynenko, "Regular Languages and CF Grammars". In *Computer Tools in Education. 1, pp.14–20*, 2012. (In Russian).

[10] S. Baranov, Ch. Lavarenne, *Open C Compiler in Forth*. In: *EuroForth'95, 27-29 Oct. Schloss Dagstuhl*, 1995.

[11] "VFX Forth for Windows. User manual. Manual revision 4.70, 19 August 2014". – Southampton: MPE Ltd, 2014. – 429 p., web: http://www.mpeforth.com/ (2014)

[12] "gForth", web: https://www.gnu.org/software/gforth/ (2017)

[13] S.N. Baranov, L.N. Fedorchenko, "Equivalent Transformations and Regularization in Context-Free Grammars" Cybernetics and Information Technologies. Bulgarian Academy of Sciences, Sofia. 2014. Volume 14, no 4, p.30-45. web: http://www.degruyter.com/view/j/cait.2014.14.issue-4/cait-2014-0003/cait-2014-0003.xml (2017)

Halting misconceived?

Bill Stoddart

August 25, 2017

Abstract

The halting problem is considered to be an essential part of the theoretical background to computing. That halting is not in general computable has been "proved" in many text books and taught on many computer science courses, and is supposed to illustrate the limits of computation. However, there is a dissenting view that these proofs are misconceived. In this paper we look at what is perhaps the simplest such proof, based on a program that interrogates its own halting behaviour and then decides to thwart it. This leads to a contradiction that is generally held to show that a halting function cannot be implemented. The dissenting view agrees with the conclusion that the halting function, as described, cannot be implemented, but suggests that this is because its specification is inconsistent. Our paper uses Forth to illustrate complex abstract arguments.

Keywords: Forth, halting problem, proof

1 Introduction

In his invited paper [2] at The First International Conference on Unifying Theories of Programming, Eric Hehner dedicates a section to the proof of the halting problem, claiming that it entails an unstated assumption. He agrees that the halt test program cannot exist, but concludes that this is due to an inconsistency in its specification. Hehner has republished his arguments using less formal notation in [3].

The halting problem is considered to be an essential part of the theoretical background to computing. That halting is not in general computable has

been "proved" in many text books and taught on many computer science courses to illustrate the limits of computation. Hehner's claim is therefore extraordinary. Nevertheless he is an eminent computer scientist[1] whose opinions reward careful attention. In judging Hehner's thesis we will take the view that to illustrate the limits of computation we need a program which can be consistently specified, but not implemented.

In this paper our aims are to examine Hehner's arguments by expressing them in Forth. A secondary aim is to show the suitability of Forth for performing such an analysis.

The halting problem is typically stated as follows. Given a Turing machine equivalent (TME) language there is no halt test program $H(P, X)$ which will tell us, for arbitrary program P and data X, whether P will halt when applied to X.

Hehner simplifies this, saying there is no need to consider a program applied to data, as data passed to a program could always be incorporated within the program. So his version is that there is no halt test $H(P)$ which tells us, for an arbitrary program P, whether execution of P will halt.

To express the halting proof in Forth, we assume we have implemented a program H with stack effect (xt $--$ f) which, for any token xt, reports whether execution of xt will halt.[2] We write $'P$ to represent the token for program P.

Were H to exist, we could use it as follows:

```
: Skip ;   : Loop BEGIN AGAIN ;
'Skip H . ↲ -1 ok
'Loop H . ↲ 0 ok
```

[1] In the area of programming semantics, Hehner was the first to propose "programs as predicates", an approach later adopted in Hoare and He's work on unifying theories. He was the first person to express the semantics of selection and iteration in terms of two simple semantic primitives, choice and guard, an approach now generally adopted and used, for example, in Abrial's B-Method. He has proposed a reformulation of set theory that supports unpacked collections and gives semantic meaning to the contents of a set, which is referred to as a bunch, and has properties perfect for representing non-determinism. His other contributions have been in areas as diverse as quantum computing and the semantics of OO languages. When the book "Beauty is our business" was conceived as a tribute to the work of E W Dijkstra, Hehner contributed a chapter discussing Gödel's incompleteness theorem. His book *A Practical Theory of Programming* [1] is available online in updated form.

[2] Our tokens are abstract analogies of Forth execution tokens, freed from any finiteness constraints.

When a Forth program is executed from the keyboard it either comes back with an "ok" response, or exhibits some pathological behaviour such as reporting an error, not responding because it is in an infinite loop, or crashing the whole system. We classify the "ok" response as what we mean by "halting".

The proof that H cannot be implemented goes as follows. Under the assumption that we have implemented H, we ask whether the following program will halt:[3]

```
: S 'S H IF Loop THEN ;
```

Now within S, H must halt and leave either a true or false judgement for the halting of S. If it leaves a true flag (judging that S will halt) then S will enter a non-terminating Loop. If it leaves a false flag (judging that S will not halt), then S will immediately halt.

Since H cannot pass a correct judgement for S, we must withdraw our assumption that there is an implementation of H. Thus halting behaviour cannot, in general, be computed. □

Hehner asks us to look in more detail at the specification of H. Since it must report on the halting behaviour of any program, it must assume the objective existence of such a behaviour. But S contains a "twisted self reference" and the halting behaviour of S is altered by passing judgement on it. H does not have a consistent specification. It cannot be implemented, but this has little significance, as it is due to the inconsistency of its specification. When someone claims a universal halt test is uncomputable, and you reply, "What do you mean by a universal halt test?" you won't receive a mathematically consistent answer.

From a programming perspective, we can add that S looks as if it will NOT terminate, because when `'S H` is executed, it will be faced with again commencing execution of `'S H`, and with no additional information to help it. S will not terminate, but this is because the halt test invoked within it cannot terminate.

The paper is structured as follows. In section 2 we verify Hehner's simplification of the halting problem. In section 3 we make some general remarks on halting, finite memory computations, and connections between halt tests and mathematical proofs. In section 4 we present a tiny language consisting of

[3]The program accesses its own token. Later, in some code experiments, we show how this is achieved.

three Forth programs with access to a halt test. We find we can still use the same proof, that a halt test cannot be implemented. We examine the specification of the halt test for this minimal scenario in detail, and we produce a Forth implementation of an amended halt test that is allowed to report non-halting either by the return of a stack argument or by an error message. In section 5 we perform a semantic analysis of S, taking its definition as a recursive equation, and conclude that its defining equation has no solution. S does not exist as a conceptual object, and neither does H.

The halting problem is generally attributed to Turing's paper on Computable Numbers [6], but this attribution is misleading. In an appendix we briefly describe Turing's paper and how the halting problem emerged from it. We also give an example of uncomputability which has a consistent specification.

The original aspects of this paper are: a careful examination of Hehner's arguments by re-expressing them in Forth; a translation of the halting problem proof to a minimal language where exactly the same argument can be made; an examination of the consistency of the halt test specification for our minimal language with an extension of this argument to the general case; an implementation of a less strict halt test for the minimal language which allows a result to be computed except in the self referential case, where non-halting is reported as an error; a semantic analysis of S and H as a conceptual objects; and a critique of the response to Hehner's 2006 paper [2] presented in *Halting still standing* [5].

2 Hehner's simplification of the halting problem

Normally the halting problem is discussed in terms of a halt test taking data D and program P and reporting whether P halts when applied to D.

Hehner's simplified halt test takes a program P and reports whether it halts.

We refer to the first of these halt tests as H_2, since it takes two arguments, and the second as H.

To verify Hehner's simplification of the halting problem we show that any test that can be performed by H_2 can also be performed by H, and any test that can be performed by H can also be performed by H_2.

Proof Given P_0 ($--$?), P_1 (x $--$?) and D ($--$ x), where P_0, P_1 are

arbitrary Forth definitions with the given signatures and D is arbitrary Forth code that returns a single stack value, and assuming tests H (xt $--$ f) and H_2 (x xt $--$ f) where $'P_0$ H reports whether P_0 halts, and D $'P_1$ H_2 reports whether D P_1 halts, then:

The test D $'P_1$ H_2 can be performed by H with the aid of the definition
: T D P_1 ; as $'T$ H.

The test $'P_0$ H can be performed by H_2 with the aid of the definition
: U $DROP$ P_0 ; as D $'U$ H_2. □

3 Some notes on halting analysis

Fermat's last theorem states that for any integer $n > 2$ there are no integers a, b, c such that:

$$a^n + b^n = c^n$$

Fermat died leaving a note in a copy of Diophantus's *Arithmetica* saying he had found a truly marvellous proof of his theorem, but it was too long to write in the margin. No proof was never found. All subsequent attempts failed until 1995, when Andrew Wiles produced a proof 150 pages long.

However, given a program *FERMAT* which searches exhaustively for a counter example and halts when it finds one, and a halt test H we could have proved the theorem by the execution:

'FERMAT H . ↲ 0 ok

This would tell us the program *FERMAT* does not halt, implying that the search for a counter example will continue forever, in other words that no counter example exists and the theorem is therefore true.

In the same way we could explore many mathematical conjectures by writing a program to search exhaustively over the variables of the conjecture for a counter example. Then use H to determine whether the program fails to halt, in which case there is no counter example, and the conjecture is proved.

3.1 Known, bounded, and unbounded memory requirements

If a program has a known memory requirement of n bits its state transitions can take it to at most 2^n different states. We can solve the halting problem by running it in a memory space of $2n$ bits and using the additional n bits as a counter. When we have counted 2^n state transitions and the program has not halted, we know it will never halt because it must have, at some point, revisited a previous state.

The postulated *FERMAT* program above has an *unbounded* memory requirement, since as it performs its exhaustive search for a counter example it will need to work with larger and larger integers.

Now consider the Goldbach conjecture, which states that every even integer can be expressed as the sum of two primes (we include 1 in the prime numbers). This is an unproved conjecture, but so far no counter example has been found, although it has been checked for all numbers up to and somewhat beyond 10^{18}.

Now suppose we have a program *GOLDBACH* which performs an exhaustive search for a counter example to Goldbach's conjecture and halts when it finds one. If Goldbach's conjecture is true, this program has unbounded memory requirements. If the conjecture is false, it has bounded memory requirements, but the bound is unknown.

When Turing formulated his Turing machines he gave them an unbounded memory resource in the form of infinite tapes. This allows a Turing machine to be formulated which will perform an unbounded calculation, such as calculating the value of π. Although we cannot ever complete the calculation, we can complete it to any required degree of accuracy, and the existence of an effective procedure for calculating π gives us a finite representation of its value.

A Turing machine consists of a finite state machine (FSM) plus an infinite tape. To be TME a language needs to be powerful enough to program a FSM, and needs to be idealised to the extent of having an infinite memory resource corresponding to the tape of the Turing machine. Providing Forth with a pair of infinite stacks is sufficient to simulate an infinite tape.

Unbounded memory resources are important for the discussion of halting in this section, but they do not play a part in our discussion of the halting proof.

4 Halting in a trivial language

The conventional view of the halting problem proof is that it shows a universal halt test is impossible in a TME language. We have also seen that failure to halt can be detected in programs with known memory requirements, because after a known number of transitions such programs are bound to have revisited a previous state, which tells us they will never terminate.

It is rather strange, therefore, that we can apply the halting program proof to a minimal language whose only programs, in semantic terms, are one that terminates and one that does not.

Consider a language $\mathcal{L}_0$ consisting of two words, *Skip* and *Loop*.

This is a stateless language for which we can specify and implement a halt test H_0. The specification is consistent because it has a model:

$$\{\ 'Skip \ \mapsto \ true\ ,\ \ 'Loop \ \mapsto \ false\ \}$$

Now we become ambitious and wish to consider a more complex language $\mathcal{L}_1$ which consists of three words, *Skip*, *Loop*, and S, with a halt test H.

Our definition of S is still:

$$:\ S\ \ 'S\ \ H\ IF\ \ Loop\ \ THEN\ \ ;$$

and note that, were S to exist it will either behave like *Skip* or *Loop*,

and our specification for H is:

H (xt $--$ f) Where xt is the execution token of *Skip*, *Loop*, *or* S, return a flag that is true if and only if execution of xt halts.

But what is the model for H ?

$$\{\ 'Skip \ \mapsto \ true\ ,\ \ 'Loop \ \mapsto \ false,\ \ 'S \ \mapsto ?\ \ \}$$

Our model must map $'S$ to either true or false, but whichever is chosen will be wrong. We have no model for H, so it cannot have a consistent specification.

We have reduced the halting scenario to a minimal language so we can write out the model of halting, but exactly the same argument applies to halting in a TME language.

In this minimal scenario of a state free language we can make the same "proof" that halting is uncomputable that we used for TME languages in the introduction. Yet we have seen that for programs with known memory requirements halting can be verified by monitoring execution of the program until it terminates or has performed enough steps for us to know that it will not halt. Of course the question being answered by the proof, on the one hand, and the monitoring of execution, on the other, are not the same. Monitoring execution does not require a "twisted self reference". There is a separation between the monitor, as observer, and the executing program, as the thing observed.

4.1 Experiments with code

We have already noted in the introduction that S looks as if it will NOT terminate, because when $'S\ H$ is executed, it will be faced with again commencing execution of $'S\ H$ with no additional information to help it. S will not terminate, but this is because the halt test invoked within it cannot terminate.

There is no reason, however, why a halt test cannot terminate in other situations, or why failure to halt cannot be reported via an error message when the halt test itself cannot halt.

Here is a specification of a slightly different halting test.

H_1 (xt $--$ f), Return a true flag if execution of xt halts. If execution of xt does not halt return a false flag, unless that failure to halt is due to non-termination within H_1, in which case report an error.

We define : $S_1\ 'S_1\ H_1\ IF\ Loop\ THEN$;

Here is the error report when S_1 is invoked.

S_1 ↲
Error at S_1
Cannot terminate
reported at H_1 *in file ...*

And here is the interaction when halt tests are invoked directly from the keyboard.

$'Loop\ H_1$. ↲ 0 *ok*
$'Skip\ H_1$. ↲ -1 *ok*
$'S_1\ H_1$. ↲ 0 *ok*

Implementation requires H_1 to know when it is being invoked within S_1. This information is present in the run time system, and we obtain it from the word S_1X "S_1 executing" which, when used in H_1, will return true if and only if H_1 has been invoked by S_1.

```
0 VALUE 'Skip  0 VALUE 'Loop  0 VALUE 'S1  0 VALUE 'H1

: S1X ( -- f, true if S1 is executing, implementation specific code )
  R> R@ SWAP >R 'S1 - 16 = ;

: Skip ;   : Loop BEGIN AGAIN ;

: H1 ( xt -- f, , If H1 has been invoked within xt and cannot terminate
without compromising the termination behaviour of xt, report a
Cannot terminate error. Otherwise return the halting behaviour of xt)
  CASE
   'Skip OF TRUE ENDOF
   'Loop OF FALSE ENDOF
   'S1 OF S1X ABORT" Cannot terminate" FALSE ENDOF
    DROP
  ENDCASE ;

: S1 ( -- ) 'S1 H1 IF Loop THEN ;

' Skip to 'Skip  ' Loop to 'Loop  ' S1 to 'S1  ' H1 to 'H1
```

This illustrates that the problem is not that halting of S_1 cannot be computed, but that the result cannot always be communicated in the specified way. Requiring H (or in this case H_1) to halt in all cases is too strong, as it may be the halt test itself that cannot halt. We may, however, require that the halt test should always halt when not invoked recursively within S_1.

5 Proof and paradox

In [2] the halting problem is compared to the Barber's paradox. "The barber, who is a man, shaves all and only the men in the village who do not shave themselves. Who shaves the barber?" If we assume he shaves himself, we see we must be wrong, because the barber shaves only men who do not shave themselves. If we assume he does not shave himself, we again see we must be wrong, because the barber shaves all men who do not shave themselves. The statement of the paradox seems to tell us something about the village, but it does not, since conceptually no such village can exist.

In a similar way, the program S which we have used in the halting problem proof, does not exist as a conceptual object[4] so what we say about it can be paradoxical.

To prove this we need a rule for the termination of the form $g\ IF\ T\ THEN$ under the assumption that computation of g terminates. To formulate the rule we need a mixture of Forth notation and formal logic notation: where the Forth program P has stack effect $--\ x$ we use $\lceil P \rceil$ to represent the value of x in our formal logic. We use $trm(T)$ for the predicate which is true if and only if T will terminate.

Now we can state our rule as:[5]

$$trm(g)\ \Rightarrow\ (\ trm(g\ IF\ T\ THEN)\ \Leftrightarrow\ (\neg \lceil g \rceil\ \vee\ (\lceil g \rceil\ \Rightarrow\ trm(T))\ )\ (1)$$

And we can state the specification of $'P\ H$ as:

$$\lceil 'P\ H \rceil\ \Leftrightarrow\ trm(P)$$

Bearing in mind that $trm(H)$ is true by the specification of H, we argue:

$trm(S)\ \Leftrightarrow$ by definition of S
$trm(\ 'S\ H\ IF\ Loop\ THEN\ )\ \Leftrightarrow$ by rule (1) above
$\neg \lceil 'S\ H \rceil\ \vee\ (\ \lceil 'S\ H \rceil\ \Rightarrow\ trm(Loop)\ \Leftrightarrow$ property of $Loop$
$\neg \lceil 'S\ H \rceil\ \vee\ (\ \lceil 'S\ H \rceil\ \Rightarrow\ false\ )\ \Leftrightarrow$ logic
$\neg \lceil 'S\ H \rceil\ \vee\ \neg \lceil 'S\ H \rceil\ \Leftrightarrow$ logic
$\neg \lceil 'S\ H \rceil\ \Leftrightarrow$ specification of H
$\neg\ trm(S)$

So we have proved that $trm(S)\ \Leftrightarrow\ \neg\ trm(S)$. This tells us that S does not exist as a conceptual object, let alone as a program. We have seen in the previous section that by relaxing the specification of H we can implement the same textual definition of S, so the non existence of S proved here can only be due to the specification of H being inconsistent.

The proof of the halting problem assumes a universal halt test exists and then provides S as an example of a program that the test cannot handle. But S is not a program at all. It is not even a conceptual object, and this

[4]Examples of conceptual objects include numbers, sets, predicates, and programs. Suppose we have P, which is supposed to be a predicate, but is claimed to have the property $P\ \Leftrightarrow\ \neg\ P$. No such predicate exists: P is not a conceptual object. $P\ \Leftrightarrow\ \neg\ P$ reduces to false, from which we can prove anything, including paradoxical properties.

[5]A referee queried whether we need to refer to a formal semantics of recursion. Such a semantics would suggest that S might not exist as a conceptual object [5]. However, since the only semantic question concerns termination, we can show it does not exist with a simple direct approach.

is due to inconsistencies in the specification of the halting function. H also doesn't exist as a conceptual object, and we have already seen this from a previous argument where we show it has no model.

A response to Hehner's Unifying Theories paper was given by by Verhoeff *et al* [5]. This paper, like [2], frames its arguments in the specialist notation of [4]. They note that Hehner's proof that the specification of S does not define a conceptual object is based on an analysis of the definition $S = \neg\ ok' \lhd\ H(S)\ \rhd\ ok'$. This just says S halts if H says it doesn't and *vice versa*. But Hehner defines $S =$ "$\neg\ ok' \lhd H(S) \rhd ok'$ ", i.e. S is a string, and this is what is passed to H. This can be fixed with a notational adjustment. Their second point is that whilst specifications, which are just mathematics, may not define conceptual objects (not all equations have solutions) the same is not true of programs. Code always defines a semantic object. However we find that, under the assumption that H has been implemented, we don't have the right mathematical conditions for the implementation of S to have a solution, and this is enough to establish the contradiction. This point caused Hehner to change his rhetoric slightly – his point is not that the proof does not confirm the non-existence of a universal halt test, but rather that a universal halt test does not exist conceptually, so we can't expect to implement it.

Our notion of an uncomputable specification requires the specification to have a model, but no implementation. An example is provided by Turing's uncomputable sequence β, discussed briefly in the appendix.

6 Conclusions

The halting problem is universally used in university courses on Computer Science to illustrate the limits of computation. Hehner claims the halting problem is misconceived. Presented with a claim that a universal halt test cannot be implemented we might ask – what is the specification of this test that cannot be implemented? The informal answer, that there is no program H which can be used to test the halting behaviour of an arbitrary program, cannot be formalised as a consistent specification.

The program S, used as example of a program whose halting cannot be analysed, observes its own halting behaviour and does the opposite. Hehner calls this a "twisted self reference". It violates the key scientific principle of, where possible, keeping what we are observing free from the effects of the

observation.

To better understand Hehner's thesis we have re-expressed his argument using Forth as our programming language. We have verified Hehner's simplification of the problem, and proposed a minimal language of three programs and a halt test, to which exactly the same proof can be applied.

Our programming intuition tells us that S will not terminate because when $'S\ \ H$ is invoked within S, H will not terminate. However, we cannot require H to return a value to report this, because that would require it to terminate! We provide a programming example based on a minimal language where we resolve this by allowing the option for a halt test to report via an error message when it finds itself in this situation. However, we can require that the halt test should always halt other situations. The problem is not the uncomputability of halting!

We have also performed semantic analysis using Forth. This analysis confirms that the halt test and S *do not exist as conceptual objects.*

We have found nothing to make us disagree with Hehner's analysis. Defenders of the status quo might say – so the halt test can't even be conceived, so it doesn't exist. What's the difference? Hehner says that uncomputability requires a *consistent* specification that cannot be implemented. Turing's uncomputable sequence β can provide such an example. A computation that inputs n and outputs $\beta(n)$ has a model, since β is mathematically well defined, but if we could compute it for arbitrary n, then β would be a computable sequence. The uncomputability of β is proved in the appendix.

Forth has been invaluable in this work in providing a concise notation, and in helping us combine programming intuition with abstract arguments. We have used it to transfer the argument to the scenario of a minimal language, where the proof still holds, and to play with a variation of the halt test that demonstrates that he problem in the scenarios we examine is not the uncomputabilty of halting.

Acknowledgements.

Thanks to Ric Hehner for extensive electronic conversations; Steve Dunne for extended discussions; participants at EuroForth 2016 for the stimulating comments and questions in response to my talk "The halting problem in Forth"; to the referees and Ric Hehner for their corrections of, and interesting comments on, a draft paper; also to Campbell Ritchie for proof reading the final version.

References

[1] E C R Hehner. *A Practical Theory of Programming.* Springer Verlag, 1993. Latest version available on-line.

[2] E C R Hehner. Retrospective and Prospective for Unifying Theories of Programming. In S E Dunne and W Stoddart, editors, *UTP2006 The First International Symposium on Unifying Theories of Programming*, number 4010 in Lecture Notes in Computer Science, 2006.

[3] E C R Hehner. Problems with the halting problem. *Advances in Computer Science and Engineering*, 10(1):31–60, 2013.

[4] C A R Hoare and He Jifeng. *Unifying Theories of Programming.* Prentice Hall, 1998.

[5] Cornelis Huizing, Ruurd Kuiper, and Tom Verhoeff. *Halting Still Standing – Programs versus Specifications*, pages 226–233. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[6] Alan M Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.

Appendix: Turing's 1936 paper and the halting problem

On computable numbers, with a contribution to the Entscheidungsproblem, Turing's paper from 1936 [6] is cited as the source of the halting problem, but it does not mention halting. The paper captures Hilbert's notion of an "effective procedure" by defining "computing machines", consisting of finite state machines with an infinite tape, which are similar to what we now call Turing machines but with significant differences. He uses such machines to define all numbers with a finite representation as "computable numbers", with the fractional part of such a number being represented by a machine that computes an infinite binary sequence. The description of these machines is finite, so numbers such as π, which are computable to any desired accuracy, can have a finite representation in terms of the machines that compute them.

Turing's idea of a computer calculating π would perhaps have been of a human being at a desk, performing the calculation, and now and then writing down another significant figure. His "computing machines" are supposed to continue generating the bits of their computable sequence indefinitely,

but faulty machines may fail to do so, and these are not associated with computable sequences.

The computing machines that generate the computable sequences can be arranged in order. Turing orders them by an encoding method which yields a different number for each computing machine, but we can just as well think of them being lexigraphically ordered by their textual descriptions.

The computable sequences define binary fractions that can be computed. Turing's contribution to the Entscheidungsproblem is in *defining* a binary sequence β that *cannot be computed*. Let $M(n)$ be the *nth* computable sequence, and define the sequence:

$\beta(n) = \quad \textbf{if } M(n)(n) = 1 \textbf{ then } 0 \textbf{ else } 1 \textbf{ end}$.

By a diagonalisation argument β is not one of the computable sequences: it is definable but not computable. The link with halting comes from asking why it cannot be computed, the reason being that although we can talk about the sequence of computing machines that generate infinite binary sequences of 0's and 1's we cannot distinguish these from machines which have the correct syntactic properties but which do not generate infinite sequences. So we cannot compute which of the computable sequences is the nth computable sequence because we cannot distinguish good and bad computing machines.

The first reference to the "halting problem" I have been able to find comes in Martin Davis's book *Computability and Unsolvability*, from 1958. By then Turing machines had taken their current form and were required to halt before the output was read from their tape. He credits Turing's 1936 paper as the source of the problem's formulation.

A proof using a computing mechanism which enquires about its own halting behaviour and then does the opposite appears in Marvin Minsky, *Computation. Finite and infinite machines*, from 1967.

SIMD and Vectors

M. Anton Ertl*
TU Wien

Abstract

Many programs have parts with significant data parallelism, and many CPUs provide SIMD instructions for processing data-parallel parts faster. The weak link in this chain is the programming language. We propose a vector wordset so that Forth programmers can make use of SIMD instructions to speed up the data-parallel parts of their applications. The vector wordset uses a separate vector stack containing opaque vectors with run-time determined length. Preliminary results using one benchmark show a factor 8 speedup of a simple vector implementation over scalar Gforth code, a smaller (factor 1.8) speedup over scalar VFX code; another factor of 3 is possible on this benchmark with a more sophisticated implementation. However, vectors have an overhead; this overhead is amortized in this benchmark at vector lengths between 3 and 250 (depending on which variants we compare).

1 Introduction

Current computer hardware offers several ways to perform operations in parallel:

Superscalar execution Independent instructions are executed in parallel, if enough functional units and other resources are available. This requires little programmer intervention: out-of-order processors find independent instructions by themselves.

SIMD instructions perform the same operation on multiple data in parallel. The programmer or compiler has to use these instructions explicitly.

Multi-core CPUs Programs have to be split into multiple threads or processes to make use of this feature.

As a close-to-the-metal language, Forth should provide ways to make use of these hardware features. At least SwiftForth and Gforth already contain features to make use of shared-memory multicores, by extending the classical Forth multi-tasking wordset. Superscalar execution is exploited by the hardware and/or the compiler [SKAH91] without programmer intervention.

In this paper I present the basic concepts and an initial version of a vector wordset (Section 3) that can make good use of SIMD instructions. The main concept and contribution is a vector stack that contains opaque vectors of dynamically determined length. The programmer can use this vector stack to express vector operations in a way that does not introduce additional dependencies, and is therefore the key to allowing very efficient implementations with relatively little compiler complexity, as well as enabling simpler implementations (with less performance). We present different ways to implement this wordset in Section 4, and Section 5 presents preliminary performance results of using this wordset. We discuss related work in Sections 2 and 6.

Terminology: In this paper, *vector* refers to application-level one-dimensional arrays with an arbitrary number of elements, while SIMD refers to what machine instructions offer: arrays with limited (and often fixed) number of elements.

2 Background

In many applications one has to perform the same operations on a lot of data, mostly independently, sometimes combining the results. This is known as *data parallelism*.

Data parallelism is obvious for many scientific applications, but can also be found in other applications, e.g., in the Traveling Salesman Problem[1]. So introducing a wordset for expressing data parallelism may be useful in more applications than one might think at first.

Computer architects provide SIMD (single instruction multiple data) instructions that allow to express some of this data parallelism to the hardware. The Cray-1 was an early machine with SIMD instructions, but starting in the 1990s, microprocessor manufacturers for general-purpose CPUs incorporated SIMD instructions in their architectures. E.g., Intel/AMD incorportated MMX,

*Correspondence Address: Institut für Computersprachen, Technische Universität Wien, Argentinierstraße 8, A-1040 Wien, Austria; anton@mips.complang.tuwien.ac.at

[1] <news:b2aed821-2b7e-456d-9a6d-c2ea1fdedd55@googlegroups.com>

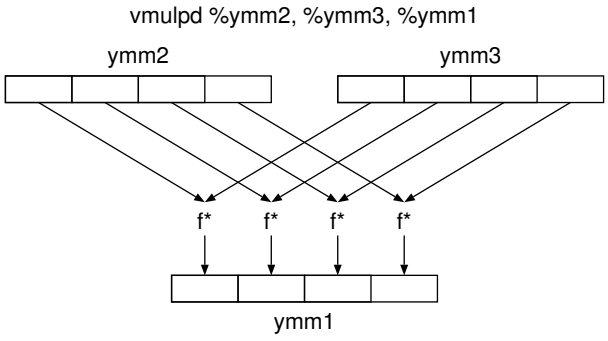


Figure 1: A SIMD instruction: `vmulpd` (AVX)

3DNow, SSE, AVX etc. and ARM incorporated Neon.

These instruction set extensions typically provide registers with a given number of bits (e.g., 128 bits for the XMM registers of SSE and AVX128, and 256 bits for the YMM registers of AVX256), and pack as many items of a basic data type in there as fit; e.g., you can pack 16 16-bit integers or 4 64-bit FP values in a YMM register. A SIMD instruction typically performs the same operation on all the items in a SIMD register. E.g., the AVX instruction `vmulpd %ymm2, %ymm3, %ymm1`[2] multiplies each of the elements of ymm2 with the corresponding element in ymm3, and puts the result in the corresponding place in ymm1 (Fig. 1).

On the application side, these instructions are usually used for implementing vector operations, such as the inner product.

Making use of these instructions in programs has been a major challenge. The following methods have been used, and Fig. 2 shows examples.

Assembly language allows specifying a specific SIMD instruction directly.

Intrinsics tell the compiler to use specific SIMD instructions; e.g., the intrinsic `_mm256_mul_pd` tells the Intel C compiler to use the `vmulpd` instruction. These intrinsics are just as architecture-dependent as assembly language, but at least they play nicely with the rest of the C code. Other compilers (e.g., gcc) typically support the same intrinsics as the Intel compiler.

Vectors as language feature APL and its modern descendent J have arrays as first-class data type, and many operations that work on arrays or generate arrays. The array sizes are determined at run-time.

[2]In this paper, we use the AT&T syntax for the AMD64 architecture; in contrast to Intel syntax, the destination of an instruction is the rightmost operand in AT&T syntax.

```
;Assembly language
vmulpd ymm1, ymm2, ymm3

/* C with Intel Intrinsics */
__m256d a,b,c;
c = _mm256_mm_mul_pd(a, b);

NB. J
a =: 3 5 7 9
b =: 2 4 6 8
c =: a*b

!Fortran Array language
REAL, DIMENSION(4) :: a,b,c
c = a*b;

/* GNU C Vector Extensions */
typedef double v4d
   __attribute__ ((vector_size (32)));
v4d a,b,c;
c = a*b

/* C with auto-vectorization */
double a[4], b[4], c[4];
for (i=0; i<4; i++)
  c[i] = a[i] * b[i];
```

Figure 2: Using SIMD instructions in programs

Modern Fortran contains an array sublanguage that allows the programmer to express various operations on whole arrays and sub-arrays directly instead of through scalar[3] operations in loops; the example in Fig. 2 shows an example that can be directly translated to `vmulpd`, but vectors of any length (including dynamically determined lengths) are supported, as well as higher-dimensional arrays, and parts of arrays.

GNU C contains a simple vector extension[4], usable only with fixed-size vectors with 2^n elements, ideally the size of the SIMD registers (gcc generates relatively bad code for larger vectors). So it mostly is useful as an architecture-independent way to specify SIMD operations, and the programmer should compose the code for longer vectors himself; for run-time determined vector sizes, this is the only option.

Auto-vectorization Ever since the Cray-1 there has been the hope of auto-vectorization: Programmers would write scalar code oblivious of SIMD instructions, and the compiler would

[3]In the context of programming with vectors, *scalar* refers to a single value, i.e., a non-vector.

[4]`https://gcc.gnu.org/onlinedocs/gcc/Vector-Extensions.html`

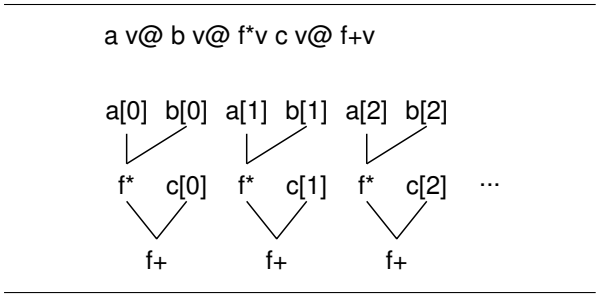


Figure 3: Dependences in a sequence of vector operations. Note that there are no dependences between computations of different elements.

find out by itself how to make use of these instructions for that code.

While auto-vectorization occasionally succeeds in vectorizing a piece of code (especially benchmarks), this is an unreliable method; there are often obstacles, that make it hard or impossible for the compiler to vectorize the code, e.g., the possibility of overlap between memory accesses in the loop; and if you ask the programmer to change his program to remove these obstacles, why stick with scalar code? If the programmer thinks in terms of vectorizing the program, the way to go is to directly express vector operations rather than expressing them through scalar operations and then hoping that the compiler will auto-vectorize them. Compiling language-level vector operations to SIMD code also requires much less complexity than auto-vectorization.

Therefore, in this paper I propose an approach to express vector operations in Forth.

3 Forth Vector Wordset

3.1 Vectors

A vector contains a dynamically determined number of bytes; different vectors can contain different numbers of bytes, but many operations require that all operands have the same length (as in APL, J, and Fortran).

Vectors are opaque: They do not reside in memory that an application program is allowed to access. Programmers can easily comply with this restriction: Only access vectors with vector words; moreover, when using this wordset, if you use the wrong words, the error will typically reveal itself quickly.

Vectors have value semantics, like cells or floats, and unlike strings in memory: When you copy a vector, the copy has an identity of its own, and it is unaffected by operations on the original (and vice versa).

The main benefit of these properties is that it gives a lot of freedom to the implementation of vector operations: Every part of every vector is independent of every other part of the same vector, other vectors, and main memory (see Fig. 3), so vectors can be processed in any order: front-to-back, back-to-front, in parallel, in some combination of these methods, or in some other order.

This restriction also gives the implementation a lot of freedom when storing the vector data: It can be stored in main memory with as much alignment and padding as is useful for an efficient implementation; or it can be (partially or fully) stored in SIMD registers, or in, e.g., graphics card memory; the implementation may also store the vector data in a way convenient for calling high-performance computing libraries in other languages (as long as the data used by these libraries are vectors or subvectors).

An alternative approach would represent vectors by an address/length pair, as is done in the standard for strings [For14, Section 3.1.4.2]; this is somewhat similar to Fortran's array language, that also works on the existing arrays in memory. The disadvantage of this approach is that efficient implementation is hard, and in some cases impossible: vectors would not necessarily be aligned for efficient memory access, the size may not be a multiple of the vector register size, and the memory areas of vectors specified in this way may overlap, so the order of element operations of a vector operation would be restricted, it would require significant compiler and/or run-time sophistication to achieve correct operation while using SIMD instructions, and efficiency would suffer as well.

Some have suggested combining the addr/len approach with restrictions on the program to avoid the compiler/run-time complexity and performance disadvantages, but this would add conceptual complexity to the usage of the wordset and very likely lead to non-portable programs, because such restrictions are hard to comply with in every instance: It is hard to find out by testing that you have not complied, unless every restriction is always exploited by the implementation you use. Also, when choosing between portability and performance, programmers will often choose performance (especially when dealing with a performance-enhancing feature).

3.2 Vector Stack

There is a separate vector stack that is used by vector words.

Why have a separate stack instead of just storing single-cell vector tokens on the data stack? Vector tokens on the data stack would be error-prone: it would be natural to use, e.g., `dup` to copy a vector token, or `drop` to get rid of it; this would be incompatible with otherwise attractive implementa-

tion options for the value semantics of vectors (see Section 4.1), and, worse, there is no implementation that would be compatible with it that does not use garbage collection.

By contrast, with a separate vector stack, the user has to use `vdup` and `vdrop` to deal with vectors, and these words (and all others dealing with vectore) can perform the bookkeeping necessary for preserving value semantics in the vector implementation.

The usual stack manipulation words get vector equivalents: `vdup vover vswap vrot vdrop vpick vroll`.

Once we have such a vector stack, we can also use it for other data, such as strings, bignums and matrixes, but that is outside the scope of the present work.

3.3 Data types

Forth uses only a few on-stack data type sizes: single-cell, double-cell and float. It uses additional in-memory sizes for communications with other software or hardware, and for making efficient use of memory.

For vectors, we usually also want to use the smallest element data types that are big enough for the application. This allows us to have shorter vectors and faster vector operations. So while on a 64-bit system we have only, e.g., `+` for adding 8-bit, 16-bit, 32-bit and 64-bit integers, for vectors we want `b+v w+v l+v x+v`, because, for long vectors, `b+v` will be 8 times faster than `x+v`.

The vector wordset uses the following prefixes for types: `b ub w uw l ul x ux sf df`.

In this paper, vector stack elements are denoted with `v` for general vectors, or *type*`v` for specific types, e.g. `uwv` for a vector of unsigned 16-bit (`uw`) values, and `sfv` for a vector of 32-bit floats.[5] Vector items are always on the vector stack, so this paper does not use a `V:` notation or somesuch for indicating that an item is on the vector stack.

Should we have vector types with cell-sized elements (`v uv`), and vectors with float-sized elements (`fv`)? Vectors with cell-sized elements would be fine, but are not implemented in the current wordset; the vector words dealing with them would be aliases of words dealing with `xv uxv` or `lv ulv` vectors.

Vectors with float-sized elements have the problem, that there are relevant Forth systems where the default float type uses the 80-bit 387 format, and is stored in memory in 10-byte (VFX) or 16-byte (iForth) units. There are no SIMD instructions for dealing with this format, so vector words for them would be slow. So, programmers should use `dfv` or `sfv` words for portability (including performance portability), and implementing `fv` words does not make much sense.

[5] The standard has `r` for on-stack FP numbers and `f` for flags, but uses the prefixes `f`, `sf`, `df` for dealing with FP numbers of various lengths in memory; should we use `sr` for 32-bit floats?

3.4 Vector operation patterns

There are a number of operation patterns when working with vectors:

Parallel vector/vector E.g., adding each element of the first vector to the corresponding element of the second vector, resulting in a third vector. This pattern works only with vectors of the same length. Words implementing this pattern have a `v` suffix.

Parallel vector/scalar E.g., adding a scalar to each element of a vector. Words implementing this pattern have a `vs` or `sv` suffix (`sv` only for non-commutative operations).

Reduce E.g., for the sum or the maximum of all elements of a vector, producing a scalar. Words implementing this pattern have suffix `r`. Currently the vector wordset does not support this pattern.

Generate a vector of a certain length, with all elements containing the same scalar (possibly unnecessary if we have `vs` instructions). Other generating operations are NGSPICE's vector(n) that produces a vector containing 0,1,2..9. Matlab's `linspace(x1,x2,n)` generates n points; the spacing between the points is $(x_2 - x_1)/(n-1)$ (always includes the endpoints). Currently the vector wordset does not support this pattern.

Reorder/shuffle elements of the vector, e.g., for use in a FFT. Certain shuffle operations are supported by SIMD instructions, but they are rather limited. For now, the vector wordset will not support this pattern.

Compress Given a vector of data and a vector of flags, pick only those data corresponding to true flags, and put them in a new (possibly shorter) vector. While this pattern is commonly used in APL, it is not well-supported in SIMD instructions, and the vector wordset will not support it for now.

Scan The APL operator \ produces the intermediate results of reducing a vector. E.g. `+\1 2 3` produces `1 3 6`. The vector wordset will not support this for now.

Index Sometimes the index of the first element satisfying a condition, or the index of the maximum or minimum element is needed. The vector wordset will not support this pattern for now.

3.5 Words

The vector wordset provides `v` (vector/vector parallel) versions of the arithmetic, logic and comparison operations `+ - * / mod negate and or xor invert lshift rshift`[6] `mux`[7] `abs max min < = > <= >= <>`, `vs` (parallel vector/scalar, with the first argument being the vector) versions of `+ - * / mod and or xor lshift rshift arshift max min < = > <= >= <>` (not `negate invert abs`, because they are unary operations, and not `mux`, because it is ternary and scalar operands do not make sense for it), and `sv` (parallel scalar/vector, with the first argument being scalar) versions of the non-commutative words `- / mod lshift rshift arshift < > <= >=`[8]. Signed integer `/ mod` may be symmetric or floored (and not necessarily the same as the scalar / and mod). The result of comparison operations is a vector with elements of the same size as the operand elements (e.g, 8 bits per element for `b<v`); all bits of the result element are 1 if the comparison result is true or 0 otherwise. For bitwise operations (`and or xor invert mux`), no type prefix is used.

For reducing words, one would use the associative binary operations: `+ * and or xor max min`.[9]

The combination of types, operations, and patterns produces a large number of words: In the current implementation, 137 `v` words, 123 `vs` words, and 76 `sv` words. We see the same thing in the SIMD extensions of computer architectures: They introduce a large number of instructions thanks to the combinations of types and operations (and possibly register widths).

3.6 Memory

Having vectors only on the vector stack with no way to move data to/from ordinary memory would be too restrictive for general usage. So the vector wordset provides two ways to deal with memory:

- Have concrete data on the memory side, i.e., a memory range where you can access individual elements with address arithmetics; the alignment of the memory range is not necessarily SIMD-friendly, nor is it padded to a multiple of the SIMD width; the memory after the last element may be inaccessible, so the last element requires special (expensive) treatment even on loading.

- Have opaque vectors in memory, i.e., the same representation as on the vector stack. Opaque vectors make the use of SIMD instructions easy by storing the vectors appropriately, but the implementation of these words has to manage the memory for the vectors with `allocate` or some other dynamic memory management method in order to allow proper padding and alignment, plus management information such as the size of the vector. Our wordset uses single-cell vector tokens.

The words for these memory accesses are:

`b!v ( v c-addr u -- )` store a vector into concrete memory; if the vector length is different from u, an error is thrown.

`b@v ( c-addr u -- v )` load a vector from concrete memory.

`v! ( v v-addr -- )` Store a vector into memory as opaque vector, storing the single-cell vector token into v-addr.

`v@ ( v-addr -- v )` Load a vector from an opaque vector in memory, accessed through the single-cell vector token v-addr.

`v@' ( v-addr -- v )` Fetch v, then clear v-addr. The advantage of this operation over `v@` is that the implementation can just use the existing reference to the opaque vector without incurring the implementation cost of copying the rest of the vector (Section 4.1). Moreover, a later `v!` to `v-addr` does not incur the implementation cost of deleting the vector that `v@` leaves at `v-addr`.

One danger of storing single-cell vector tokens in memory is that this provides a hole for subverting the implementation of value semantics: these tokens can be copied with `@ ! move` etc. If this proves to be a problem, a debugging mode can make these tokens location-dependent by xoring them with v-addr on `v!` and `v@`. This should quickly unveil accidential copying of this kind; there is, of course, no protection against intentional subversion of the implementation in Forth.

[6] Both signed and unsigned shifts right are supported.

[7] `Mux ( x1 x2 x3 -- x )` is a bitwise operation, that selects a bit from x1 if the corresponding bit from x3 is 1, otherwise it selects the corresponding bit from x2.

[8] E.g., `0 l-vs` would have no effect, while `0 l-sv` would be equivalent to `lnegatev`.

[9] `F+` and `f*` do not satisfy the associative law, but `df+r` is useful anyway, because it delivers an approximation to the rational/real value of the computation, just like `f+` and `f*` itself.

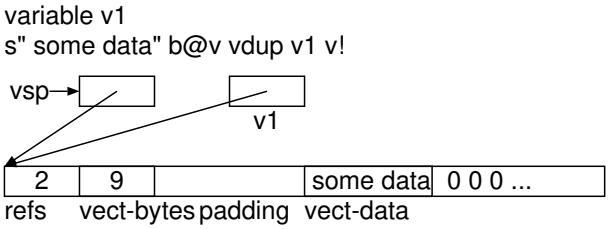


Figure 4: A vector in our implementation with refcounts (`refs`) after performing the shown code

```
vec vect-bytes @ vect-data 0 vect-data ?do
  vec1 i + df@ vec2 i + df@ f+ vec i + df!
[ 1 dfloats ] literal +loop
```

Figure 5: The vector-processing loop of the *trivial* implementation of `f+v`. `vec`, `vec1` and `vec2` are locals containing vector addresses, `vect-bytes` and `vect-data` are the fields shown in Fig. 4.

4 Implementation

This section describes various implementation approaches, as well as the current implementation.

4.1 Vectors

A vector is stored in `allocate`d memory and is aligned and padded to the SIMD granularity (e.g., 32 bytes for AVX256). In front of the actual vector data, there is bookkeeping information: The number of bytes in the vector, and possibly a reference count (see below); in addition, there may be some padding to align the actual vector data, too (see Fig. 4). The address of the start of the `allocate`d memory is the vector token.

There are two ways to implement the value semantics:

linear Every vector has exactly one reference. Copying (`v@ vdup vover`) creates (`allocate`s) a new copy of the complete vector, and consuming or overwriting a token (`vdrop v!` and many other operations) `free`s the vector.[10]

refcount Every vector can have several tokens referring to it; the number of tokens is stored with the vector in a reference count. When copying the token, the reference count is increased, when consuming or overwriting a token, it is decreased; if the reference count reaches 0, the vector itself is `free`d. This reduces the number of `allocate`s and `free`s, at the cost of some additional complexity.

I normally avoid reference counting, because it does not handle cyclic data structures well, but vectors don't contain any pointers to other data at all, and therefore cannot form cycles, so reference counting is ok for this purpose.

In many operations (e.g., `df+v`), one vector (or more) is consumed, and another of the same length is created. Then one can use the memory of a consumed vector for the created one, avoiding the overhead of `free` and `allocate`, unless (in the refcount variant) the consumed vector still has references left.

The current implementation supports both approaches (based on a compile-time flag), so the user can determine the efficiency difference himself.

Another alternative that comes to mind is to copy just the references, but use garbage collection for managing the memory. A disadvantage is that words that consume and produce a vector (e.g., `f+v`) cannot reuse the vector memory directly, but would always have to allocate new vector memory. Allocation is about as expensive as with explicit `allocate`/`free` unless you use a copying garbage collector, and a copying garbage collector has relatively high collection overhead for big data structures (such as potentially our vectors, when they are used for sounds or pictures). Overall, this alternative does not look attractive.

Region-based memory allocation [Ert14] may be useful when dealing with longer-term storage of vectors, but is probably too cumbersome to be used for the memory management of intermediate vector results; also, it is not clear how to make region-based memory allocation work properly with reference counting.

Finally, a compiler that is analytical about the vector operations can avoid the overhead of vector allocation and freeing in many cases.

4.2 Vector stack

Once vectors have been implemented, the vector stack is trivial: It is just a stack of vector tokens. However, unlike in a normal stack of cells, the copying or consuming words have to perform the appropriate copying or deleting of the referenced vectors and/or reference-count bookkeeping.

4.3 Computations

Trivial implementation

The vector words can trivially be implemented in standard Forth with loops containing scalar computation words (see Fig. 5).

While this implementation realizes none of the SIMD speedup that the vector wordset is designed for, it provides a fallback option for users who want

[10] The name `linear` for this approach is inspired by Henry Baker [Bak94].

```
simple:
 vmovapd (%rdi,%rax,1),%ymm0
 vaddpd  (%rsi,%rax,1),%ymm0,%ymm0
 vmovapd %ymm0,(%rdx,%rax,1)
 add     $0x20,%rax
 cmp     %rax,%rcx
 ja      simple
```

Figure 6: The vector-processing loop of the *simple* implementation of `df+v`.

```
sophisticated:
 vmulpd  (%rdi,%rax,1),%ymm0,%ymm1
 vaddpd  (%rsi,%rax,1),%ymm1,%ymm1
 vmovapd %ymm1,(%rdx,%rax,1)
 add     $0x20,%rax
 cmp     %rax,%rcx
 ja      sophisticated
```

Figure 7: The vector-processing loop of the *sophisticated* implementation of the sequence `df*vs df+v`.

to write portable code: They can write code using the vector wordset, and still run it (albeit slowly) on Forth systems that do not have a SIMD implementation of the vector wordset. It also provides a gradual approach for SIMD implementations: the implementor can implement the most important operations using SIMD instructions first, still falling back to the trivial implementation for the words he has not implemented yet.

Simple implementation

A simple implementation implements every vector word separately, but uses SIMD instructions for that.

For the vector-parallel `v`, `vs`, and `sv` words, the meat of the word is the *vector loop*: in each iteration, it loads the operand(s) from the `vect-data` memory into SIMD registers, uses a SIMD instruction to perform the operation n times in parallel, and then stores the result back into the memory for a vector (see Fig. 6). For running that on a CPU with in-order execution, you want to software-pipeline [Cha81] this loop for good performance; on out-of-order execution hardware, the hardware reorders the instruction execution by itself, achieving the same result.

For the `r` (reducing) words, the implementation is more involved: Thanks to associativity, there are many different ways to evaluate the result: A very parallel implementation of `+r` divides the vector into pairs of numbers, computes the sum of the pairs, resulting in an $n/2$-sized vector; repeat that until you have only one number left, the result. A very sequential implementation of `+r` would add up all the vector elements, one after the other, incurring $n-1$ times the latency of $+$.

One way to use SIMD instructions for reduction would be to add up SIMD-register-wide parts of the vector in a SIMD register; then each element of the vector occurs exactly once in the computation of exactly one of the components of the SIMD register; finally, the components of the SIMD register are reduced to form the final result. This may not fully utilize the resources of the CPU, especially for FP operations, which have a latency of more than one cycle. More parallelism can be exploited by adding up the elements of the vector in 4 or 8 SIMD registers in parallel (in an unrolled loop), then adding these registers together with SIMD instructions, and finally the components of the resulting SIMD register.

Sophisticated implementation

If we have several vector-parallel words in a Forth-level basic block[11], the simple implementation would produce several loops, with the data stored in memory between the loops, incurring loop overhead, and load and store overhead, and possibly overhead for allocating and freeing vectors for the intermediate results. Instead, several vector-parallel words can be combined into a single loop[12], with the intermediate results only in SIMD registers (i.e., not as full vectors, see Fig. 7).[13]

We can also let reducing vector words participate in this scheme, with some caveats: The result of the reduction must not be used in the same sequence of vector words, so the combining ends with the first word that uses the result of the reduction. And the unrolling that you may want for the reduction would complicate the rest of the code generation; on the other hand, a smaller unrolling factor (even 1) may be sufficient to achieve good performance given that the loop performs not just one reduction, but more.

Letting concrete-memory stores (e.g., `b!v`) participate in the combining also has caveats: If the memory of such a store overlaps the memory of concrete loads or other concrete stores, the result of a naïve combination of vector operations can produce an incorrect result. As a simple example,

```
a 1024 b@v a 64 + 1024 b!v
```

logically has to copy the whole 1024 bytes to the vector stack before it starts storing, but a naïve combining implementation might overwrite `a 64 +`

[11] A basic block is a straight-line code segment.

[12] This is a special case of the general optimization *loop fusion*.

[13] This is similar to *vector chaining* used in hardware-pipelined vector processors such as the Cray-1.

```
simple2:
 vmovapd (%rdi,%rax,1),%ymm0
 vaddpd  (%rsi,%rax,1),%ymm0,%ymm0
 vmovapd %ymm0,(%rdx,%rax,1)
 vmovapd 0x20(%rdi,%rax,1),%ymm0
 vaddpd  0x20(%rsi,%rax,1),%ymm0,%ymm0
 vmovapd %ymm0,0x20(%rdx,%rax,1)
 add     $0x40,%rax
 cmp     %rax,%rcx
 ja      simple2
```

Figure 8: The loop of Fig. 6 unrolled by a factor of 2

before loading this memory location, producing a different result (like the difference between `move` and `cmove`).

One solution to this problem is to check the memory ranges for overlaps before the loop; if there is an overlap, let the loop write to temporary, non-overlapping memory regions, and copy these to the target addresses in the right order after the loop.

Unrolling

By unrolling the vector loop (see Fig. 8), the loop overhead can be reduced for long vectors. It turns out that this does not improve performance significantly on the Core i5-6600K, but it may help on other CPUs.

Unrolling normally has to deal with left-over iterations. In the case of vectors we can avoid that by making the vector data long enough for our preferred unrolling factor (e.g., with 32-byte SIMD instructions and unrolling factor 2, always have multiples of 64 bytes as vector data).

Beyond basic blocks

Extending the combining of vector words beyond basic blocks is possible, but significantly more complex: the vector stack has to be analysed beyond basic blocks, and there are some issues to consider.

For `if`s, one implementation is to pull the `if` inside the loop implementing the combined vector operation; loop unrolling can reduce the number of dynamically executed `if`s (see Fig. 9).

Another way to deal with `if` is if-conversion [MLC+92]: Both branches are computed, and the result is selected with a `muxv` operation. However, in cases where if-conversion is beneficial, I expect programmers to perform it at the source level, so I would not perform this at the compiler level. Also, this is not possible for every operation, in particular not for stores.

If the vector operations are contained in a loop, we can extend the combining by unrolling this loop

```
df+v x 0< if
  a v@ df+v then
0.5e df*vs

# x in %r10, a in %r11, 0.5 in %ymm2
vector_loop:
  vmovapd (%rsi,%rax,1), %ymm0
  vmovapd 0x20(%rsi,%rax,1), %ymm1
  vaddpd  (%rdi,%rax,1),%ymm0,%ymm0
  vaddpd  0x20(%rdi,%rax,1),%ymm1,%ymm1
  test    %r10, %r10
  jns then
  vaddpd  (%r11,%rax,1),%ymm0,%ymm0
  vaddpd  0x20(%r11,%rax,1),%ymm1,%ymm1
then:
  vmulpd  %ymm2,%ymm0,%ymm0
  vmulpd  %ymm2,%ymm1,%ymm1
  vmovapd %ymm0, (%r12,%rax,1)
  vmovapd %ymm0, 0x20(%r12,%rax,1)
  add     $0x40,%rax
  cmp     %rax,%rcx
  ja      vector_loop
```

Figure 9: A vector code fragment containing an if, and a possible way to compile it. The `if` moves inside the vector loop, and loop unrolling (factor 2) is used to reduce its overhead.

```
n2 0 ?do
  b1 i th v@
  a j n2 * i + dfloats + df@
  f*vs f+v
loop

# %ymm3=a[j,i]
# %ymm2=a[j,i+1]
# %rbx=%r9=vtos vect-data
# %r11=b[i]   vect-data
# %r10=b[i+1] vect-data
sophisticated_unrolled:
  vmulpd (%rcx,%r11,1),%ymm3,%ymm0
  vaddpd (%rcx,%rbx,1),%ymm0,%ymm0
  vmulpd (%rcx,%r10,1),%ymm2,%ymm1
  vaddpd %ymm1,%ymm0,%ymm0
  vmovapd %ymm0,(%rcx,%r9,1)
  add    $0x20,%rcx
  cmp    %rcx,%r8
  ja     sophisticated_unrolled
```

Figure 10: A Forth loop containing vector words, and the assembly language for the vector loop (the (outer) do loop is not shown) for two iterations of the do loop (not the vector loop); i.e., the result of unrolling the do loop by a factor of 2.

```
typedef double
  vdf __attribute__ ((vector_size (32)));

static void dfplusv_(vdf*v1,
  vdf*v2, vdf*v, size_t bytes)
{
  size_t i;
  ...
  for (i=0; i<bytes; ) {
    *v = *v1+*v2;
    i+=SIMD_SIZE, v1++, v2++, v++;
  }
}
```

Figure 11: The C-level implementation of the vector loop of `df+v` in Gforth. The resulting assembly code is shown in Fig. 6.

```
genv-binary-c dfplusv_ vdf *v1+*v2
genv-binary df+v dfplusv_ df-type f+
```

Figure 12: Generating words: The first line generates the C function `dfplusv_` shown in Fig. 11 (note how the C expression from this line appears in the function), the second line generates the Forth vector word (including memory management) `df+v`, calling `dfplusv_` if available, otherwise generating a *trivial* implementation that uses `f+`.

(instead of or in addition to unrolling the vector loop). Figure 10 shows the inner do loop of the vector version of matrix multiplication, as well as the vector loop generated from an unrolled (factor 2) do loop body. In addition to reducing the vector loop overhead, the unrolling reduces the number of vtos accesses (only one load and one store vs. two each for the two iterations without unrolling). However, this kind of unrolling requires dealing with left-over iterations.

4.4 Current implementation

The current implementation of the vector wordset supports different implementation options: It supports choosing between *linear* and *refcount* options (independent of the other options), it includes a *trivial* implementation (for all systems that don't have anything better yet), and it has a *simple* implementation for Gforth that is based on GNU C's vector extensions (see Fig. 11). There are further configuration options for this variant: You can define the SIMD size (default 16 bytes), and the vector-loop unroll factor (default 1).

Given the large number of vector words, all following a few patterns, plus these configuration options, the words are not hand-coded, but are instead generated. Figure 12 shows the lines generating `df+v`. The `dfplusv_` function and Forth word is generated only on Gforth. Another system could provide its own (e.g. `code`) version of `dfplusv_` (the core vector loop), and this would then be called by `df+v`, upgrading this word from a trivial implementation to a simple implementation.

```
C_scalar:
 movsd (%rdi),%xmm1
 add   %rsi,%rdi
 mulsd %xmm0,%xmm1
 addsd (%rdx),%xmm1
 movsd %xmm1,(%rdx)
 add   %rcx,%rdx
 sub   $0x1,%r8
 jne   C_scalar
```

Figure 13: The inner loop of the *C scalar* implementation; it allows different strides for the involved vectors and therefore has separate increments for the addresses.

5 Results

This section gives some idea of the speedups achievable by using various vector word implementation approaches. We enhance the existing matrix multiplication code[14] with variants that use the vector wordset.

Note that while the vector wordset shows nice speedups over scalar code on large-matrix multiplication, calling a specialized matrix multiplication library probably shows even better performance, so this is not the ideal application area for vectors; but there are areas where no specialized libraries are available, and the vector wordset can be useful there. Here I use matrix multiplication for benchmarking, because it is vectorizable, because we already have a matrix multiplication benchmark, and because it allows scaling for arbitrary vector lengths.

We compared the following vector implementations and matrix multiplication variants:

trivial Vector words are implemented using scalar Forth words without loop unrolling (Fig. 5).

simple Matrix multiplication uses `f*vs` and `f+v`, each of which is implemented as a separate loop, with the intermediate vector stored in memory. The vector loops are written C with the GNU C vector extension and compiled to use AVX instructions (Fig. 6); the rest of the vector words is written in Forth.

[14] `http://theforth.net/package/matmul`

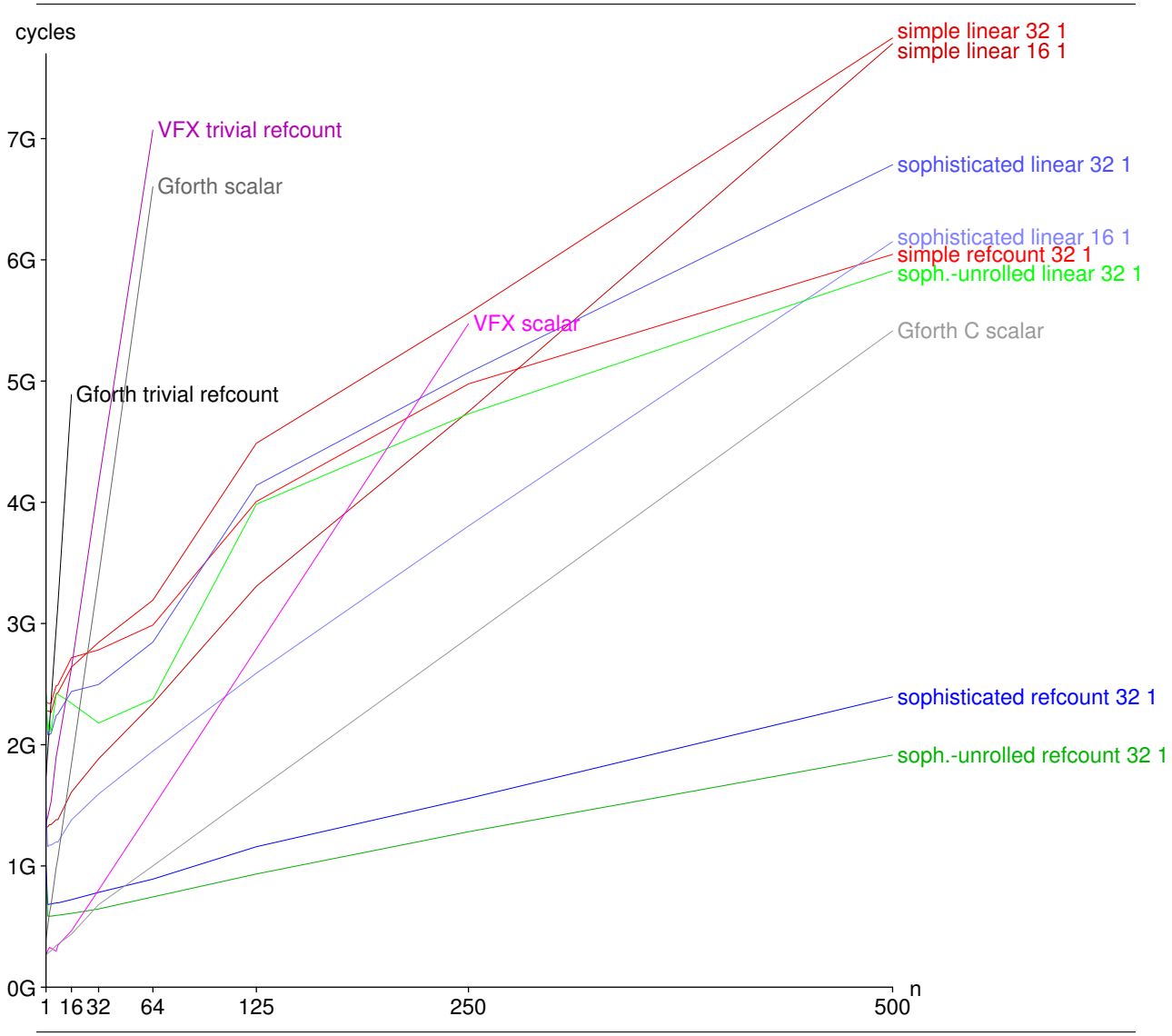


Figure 14: Timings for 20 matrix multiplications, each performing 250,000 times `f*vs f+v` (or equivalent) for n-wide vectors

sophisticated We have no sophisticated compiler, so we fake the effect by implementing a word `f*+vvs` and use that instead of the sequence `f*vs f+v`; `f*+vvs` combines Forth and GNU C in the same way as *simple* (Fig. 7).

soph.-unrolled We fake the effect of a sophisticated compiler with unrolling (factor 2) by writing a word that combines the effect of `v@ f*vs f+v v@ f*vs f+v`, resulting in the code shown in Fig. 10.

scalar The matrix multiplication code written in scalar Forth code, otherwise using the same algorithm. Unlike *trivial*, this version unrolls the loop by a factor of 4, providing a significant speedup on VFX (where the loop counter update otherwise limits performance).

C scalar The inner loop of the matrix multiplication uses scalar C code (Fig. 13). This is mostly useful for determining how good the (Forth) scalar implementation perfroms.

For all the vector (i.e., not scalar) variants, both *linear* and *refcount* was measured. For *simple*, *sophicticated*, and *soph.-unrolled*, SIMD sizes of 16 (AVX128) and 32 (AVX256) were measured, and vector-loop unrolling factors of 1, 2, and 4.

The benchmarks were run on a 4GHz Core i5-6600K (Skylake) running Debian 8 (glibc 2.19). Two Forth systems were used: `gforth-fast` (development version from August 2017) was used for all variants, VFX Forth 4.72 was used for *trivial* and *scalar*.

The benchmark multiplies a 500×500 matrix with a $500\times n$ matrix for varying n; given the algorithm of the benchmark, this always produces 250,000 instances of `f*vs f+v` (or the scalar equivalent), with

vector length n. I.e., for all n the overheads were the same. For a single matrix multiplication, compiling the vector wordset takes longer than some of the benchmark instances, so the benchmark performs 20 such matrix multiplications to mitigate this effect. Overall, the benchmark loads 10M×n FP values, stores 5M×n FP values, performs 5M×n FP additions and 5M×n FP multiplications.

Figure 14 shows a selection of the results. Showing all results would have overloaded the graph, so we only show the most relevant ones, but also discuss the other results here.

Unrolling the vector loop had little effect on performance, so here we show only unrolling factor 1.

SIMD size did not have the big effect I expected. SIMD size 16 (AVX128) was often slightly slower than SIMD size 32 (AVX256), but occasionally faster (some cases where it is faster are shown in Fig. 14). SIMD size 32 apparently produces some non-linear effects, especially for the variants that `allocate` and `free` memory, so I suspect some interference between AVX256 and the memory allocator. The AVX128 variants of the same benchmarks are closer to linear.

Refcount clearly beats *linear* for this benchmark, across all vector sizes. Apparently the overhead of `allocate` and `free` is much larger than that of reference counting, and for the larger vector sizes, you also have to pay for copying quite a bit of vector data on each `v@`. Still, the slowdown of *soph.-unrolled linear 32 1* over *soph.-unrolled refcount 32 1* is surprisingly large; one contributing factor is probably that *sophisticated* and *soph.-unrolled* with *refcount* doe not perform a single `allocate` or `free` in the core of the matrix multiplication. For the trivial implementations, we show only the *refcount* variants; the *linear* variants are slightly slower.

Given that much of the time is apparently spent in `allocate` and `free` for many of the results, repeating the benchmark on a platform with a different implementation of these words might give quite different results; in particular, a per-thread cache has been added to `malloc()` in glibc 2.26.[15] In the present case all the `allocate`d and `free`d vectors have the same size, so such a cache should work very well. One could also add such a cache to the vector wordset implementation, to reduce the performance dependency on the underlying `allocate` and `free` implementation.

Overall, as expected, for the larger vector sizes we have a big performance increase from *trivial* through *simple*, *sophisticated* up to *soph.-unrolled*, with the *scalar* results being between *trivial* and *simple*. The performance of *C scalar* is interesting, because it is faster than everything except *sophisticated* and *soph.-unrolled* (at small vector sizes, *C scalar* beats even them); however, when programming in Forth, we usually don't have a scalar C version at hand, so the (Forth) *scalar* results are more relevant.

By looking where the *scalar* lines cross those of various vector implementations, we can determine at what vector length using vector words starts paying off. For *Gforth scalar*, the crossover point with *soph.-unrolled refcount 32 1* is at vector length 3, with *sophisticated refcount 32 1* at vector length 4, *simple refcount 16 1* is faster at vector length 16 and *simple refcount 32 1* is faster at length 32.

VFX scalar is quite a bit faster, crossing over *soph.-unrolled refcount 32 1* only between vector lengths 16 and 32, and crossing over *simple refcount 32 1* between 125 and 250. However, these vector implementations all run on Gforth, and I expect that a SIMD-based vector implementation in VFX will run faster and reach crossover sooner.

Overall, we see that the vector words can provide a speedup, especially if the scalar code is not compiled optimally (whereas the inner loops of the vector words can be written in assembly language). However, vector words have an additional overhead, and that means that, for short vector lengths, using the vector words will produce a slowdown.

6 Related work

Related work in other languages has been introduced in Section 2.

The most significant difference between the vector wordset and the Fortran array sublanguage is that our vectors are stored separately from ordinary memory, avoiding alias problems, whereas the Fortran array sublanguage operates on arrays and subarrays that can be accessed in other ways, too, and therefore has to worry about alias problems.

The most significant difference between GNU C's vector extensions and the vector wordset is that GNU C's vector types have a fixed size that is restricted to be a power of 2, and in practice should be the same as the SIMD size; so it is essentially a way to express SIMD operations without resorting to architecture-specific intrinsics or assembler. In contrast, the vector words process vectors of arbitrary length, which does not even have to be a multiple of the SIMD length, thus providing a higher-level programming interface.

APL is much more sophisticated than the vector words, and includes operations that do not benefit from current SIMD instructions, and are hard to implement efficiently. If the vector words become popular and such features are asked for, the vector wordset may grow in the direction of APL in the future. Of course, you can instead use APL or J

[15] `https://lwn.net/Articles/729761/`

today (but then have to live without the features of Forth).

Closer to Forth, there is a Forth dialect designed for genetic programming that includes vector and matrix operations [HRvR07] in order to let the genetic programming system discover programs that benefit from such operations, such as signal processing. The Forth dialect uses a combined stack for all the types (including vectors and matrices), static type checking, and overloading resolution. Apart from that, the paper is very superficial in its description of the vector words. Our vector words are oriented towards the traditional Forth model of not performing type checking. The separate vector stack is a direct consequence of this model, especially because we want to treat vectors as opaque data type (unlike, traditionally, strings [Ert13]) to avoid aliasing.

7 Conclusion

By having opaque vectors and a wordset for them, we can make use of SIMD instructions without unreliable compiler complications such as alias analysis or auto-vectorization. The wordset can be implemented in different ways: The *simple* implementation is easy to implement; its performance for large vector sizes is better in our benchmark than using scalar Forth code on VFX. The *sophisticated* implementation provides a better speedup, but requires more implementation effort. The source code can be found on `https://github.com/AntonErtl/vectors`.

Acknowledgments

Herbert Pohlai provided valuable knowledge about APL and J. Marcel Hendrix and the anonymous reviewers provided valuable feedback on the paper. Marcel Hendrix provided information on further *generate* patterns.

References

[Bak94] Henry Baker. Linear logic and permutation stacks — the Forth shall be first. *ACM Computer Architecture News*, 22(1):34–43, March 1994.

[Cha81] Alan E. Charlesworth. An approach to scientific array processing: The architectural design of the AP-120B/FPS-164 family. *Computer*, pages 18–27, September 1981.

[Ert13] M. Anton Ertl. Standardize strings now! In *29th EuroForth Conference*, pages 39–43, 2013.

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

[For14] Forth 200x Standardization Committee. *Forth Standard 2012*, 2014.

[HRvR07] Kenneth Holladay, Kay Robbins, and Jeffery von Ronne. FIFTH^{TM}: A stack based GP language for vector processing. In Marc Ebner et al., editor, *Genetic Progamming (EuroGP)*, pages 102–113. Springer LNCS 4445, 2007.

[MLC+92] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *25th Annual International Symposium on Microarchitecture (MICRO-25)*, pages 45–54, 1992.

[SKAH91] Mark Smotherman, Sanjay Krishnamurthy, P. S. Aravind, and David Hunnicutt. Efficient DAG construction and heuristic calculation for instruction scheduling. In *MICRO-24,* 24^{th} *Annual Intl. Symp. on Microarchitecture*, pages 93–102, 1991.

Special Words in Forth

Stephen Pelc
MicroProcessor Engineering
133 Hill Lane
Southampton SO15 5AF
England
t: +44 (0)23 8063 1441
e: sfp@mpeforth.com
w: www.mpeforth.com

Abstract

Over the last few years, I have become convinced that I do not understand the ANS Forth description of compilation and how this situation came about. The Forth 2012 description of compilation is the same as that of ANS. This paper describes the process of understanding that leads to being able to make a few proposals to make use of a new description of compilation. In essence, we are going to have to regard IMMEDIATE as a special case of our new situation. The model also allows us to build words that would previously have had to be state-smart.

Introduction

The Forth94 (ANS) and Forth 2012 standards talk about execution of a word in terms of semantics. In the Oxford dictionary, we find the definition of semantics to be:

The branch of linguistics and logic concerned with meaning. The two main areas are logical semantics, concerned with matters such as sense and reference and presupposition and implication, and lexical semantics, concerned with the analysis of word meanings and relations between them.

Wikipaedia says:

In **programming** language theory, **semantics** is the field concerned with the rigorous mathematical study of the **meaning** of **programming** languages. It does so by evaluating the **meaning** of syntactically legal strings **defined** by a specific **programming** language, showing the computation involved.

In terms of undestanding Forth standards, these do not help much. In practice semantics means action or behaviour. From the Forth 2012 standard:

compilation semantics: The behavior of a Forth definition when its name is encountered by the text interpreter in compilation state.

execution semantics: The behavior of a Forth definition when it is executed.

interpretation semantics: The behavior of a Forth definition when its name is encountered by the text interpreter in interpretation state.

In this paper we use semantics, behaviour and action interchangeably.

MPE's VFX code generator was written in the late 1990s just as the Forth94 standard was being adopted by most vendors. In particular, VFX took advantage of the then new word **`COMPILE,`** to attach code generators for a range of words. It did this while preserving the classic Forth interpreter loop, or so we thought.

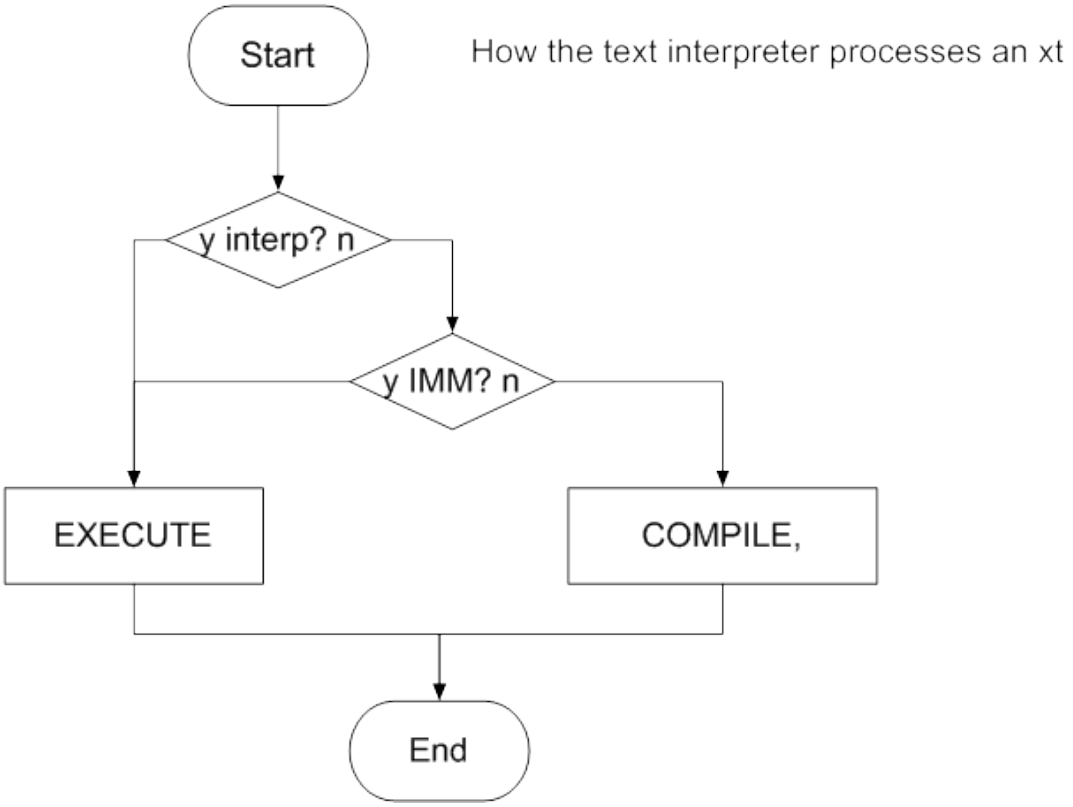


Illustration 1: Classical Forth interpreter loop

```
: process-xt    \ i*x xt -- j*x
  state @ 0 = if
    execute
  else
    dup immediate?
    if  execute  else  compile,  then
  then
;
```

The classical Forth interpreter loop has been used to describe the operation of Forth for over three decades now. It has been a useful model for many people. People regularly claim that they need to write a custom interpreter and that not all Forth systems permit this in a portable manner. We will see that a minor change to the loop and its associated structures brings it in line with Forth 2012 and expands the interpreter's facilities to take advantage of the Forth 2012 description of Forth words' action or behaviour or semantics.

Although this paper describes the interpreter in terms of the classic Forth interpreter loop, it should not be assumed that other techniques for writing interpreters are excluded. Exactly the same problems and solutions are present in techniques with different organisations including recognisers.

Smart COMPILE,

VFX Forth and other Forths take advantage of **`COMPILE, ( xt -- )`** by attaching optimisers to the words that they generate code for. For example, the word **`DUP`** has a word **`C_DUP`** that generates code for **`DUP`**. The xt for **`C_DUP`** is attached to **`DUP`**. Then when **`COMPILE,`** looks at **`DUP`** it executes **`C_DUP`** to generate the code for **`DUP`**.

The smart **COMPILE,** introduces the idea that a word (identified by one primary xt) may require one or more secondary xts. It has become common practice in desktop Forths for dictionary headers to contain more than just link, name and flags. This trend is particularly true in Forth systems that perform native code compilation (NCC).

The smart **COMPILE,** can completely separate the interpretation (execution of **DUP**) and compilation actions of a word. This technique can also be used for other words such as **IF**, with the deliberate intention that the interpretation and compilation actions of a word can be separated. However, **COMPILE,** is then broken as far as current standards are concerned because structure words such as **IF** produce or consume stack items, and string words parse the input stream. There may/will also be corner cases to do with **POSTPONE**.

Standards issues

The use of the smart **COMPILE,** for optimisation is not contentious. However, it opens a box that cannot and should not be closed. The Forth94 standard introduced a new way of talking about Forth words. Words have a number of actions, including interpretation and compilation actions. The only standard way to separate interpretation and compilation actions is, paradoxically, to define them as being the same and then to use **STATE** to separate them within the word. This is the state-smart nightmare that leads to bugs which are hard to find.

In the Forth94 and Forth 2012 world, very few words are defined as **IMMEDIATE** and there is no standard way to ask the system if the xt of a word is of an **IMMEDIATE** word.

In terms of the classical loop shown above, the only place at which non-default compilation semantics can be attached is **COMPILE,** and the system immediately becomes contentious, not least because some people insist that **IF** must be **IMMEDIATE** without stating any evidence for this. Another way to look at the problem is to state that the language of the standard does not match any Forth implementations except cmForth and Gforth. Chuck Moore's cmForth is as idiosyncratic as all Chuck Moore's other tools and was obsolete at the time of the ANS standard. Gforth's original design target was to be a model implementation of the Forth94 standard, i.e. the standard is correct. In my opinion this design target has lead to complexity. Correcting the disconnect between the current standard and real Forths while maintaining simplicity is the function of this paper.

A way forward

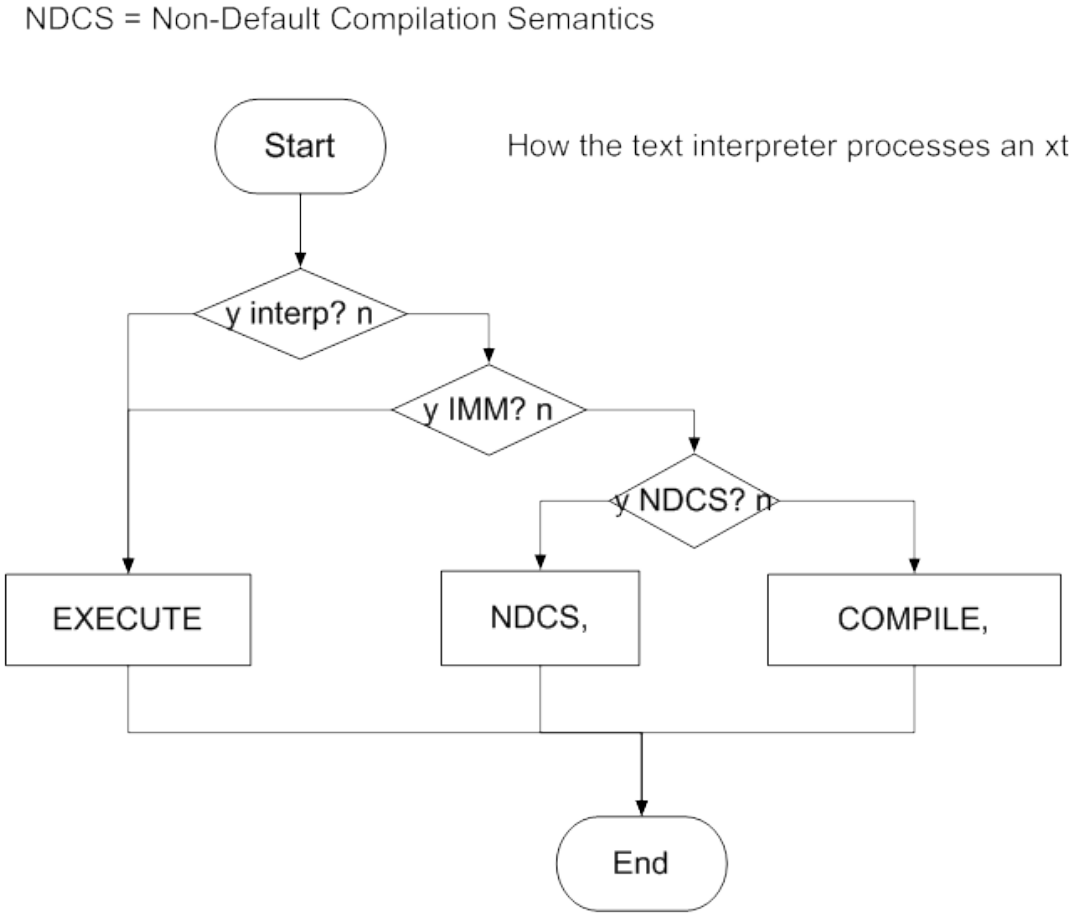


Illustration 2: Allowing for the Forth94 and Forth 2012 standards

```
: process-xt    \ i*x xt -- j*x
  state @ 0 = if
    execute
  else
    dup immediate? if
      execute
    else
      ndcs?
      if  ndcs,  else  compile,  then
  then
;
```

The picture illustrates a Forth interpreter/compiler loop that has been modified to cope with separated interpretation and compilation actions.

We also need a small number of new words that enable the loop to be constructed portably:
`IMMEDIATE?  Xt -- flag` ; return true if the word is immediate
`NDCS?  Xt -- flag` ; return true if the word has non-default compilation semantics
`NDCS,  i*x xt -- j*x` ; like **`COMPILE,`** but may parse.

In order to finish up, we need to understand what the word labelled **`NDCS,`** actually does. It finds the word that performs the non-default compilation semantics and then **`EXECUTE`**s it.

The next picture shows the loop using the definition of **IMMEDIATE** words as having the same interpretation and compilation semantics.

The significant change is the introduction of a dictionary header flag, NDCS, which indicates that a word has non-default compilation semantics.

Replace IMMEDIATE with NDCS

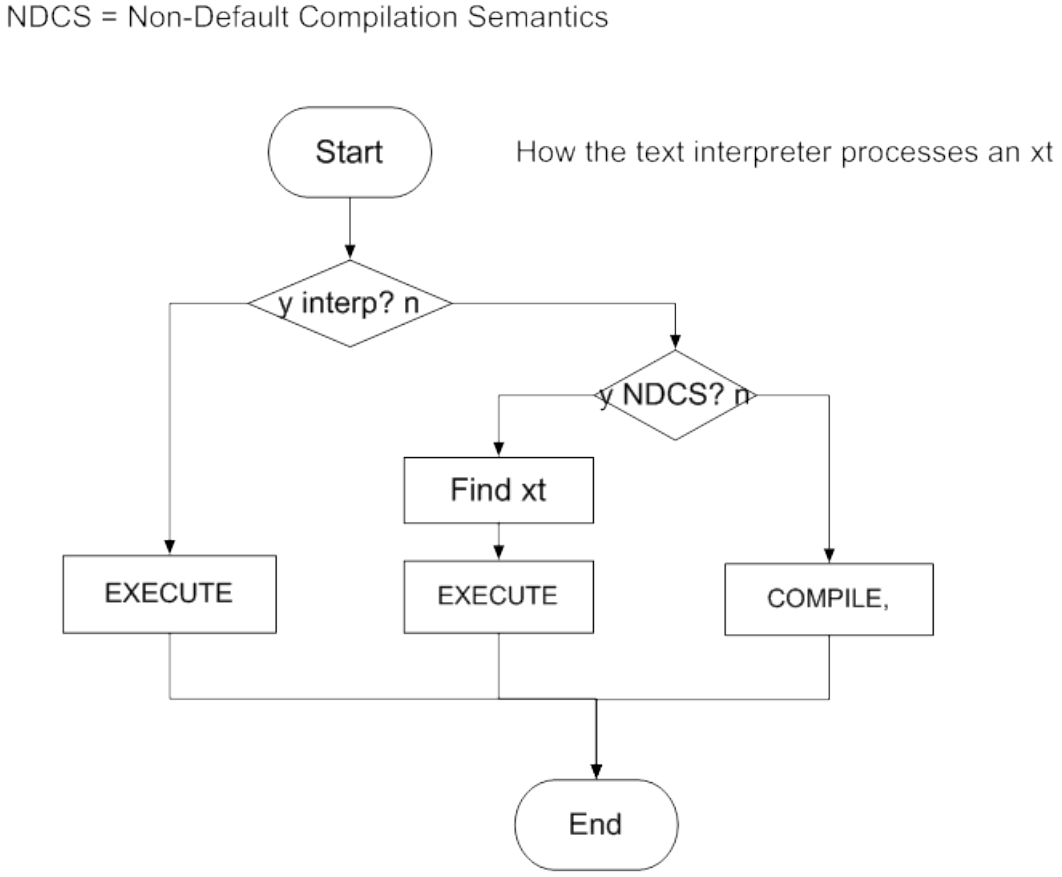


Illustration 3: The IMMEDIATE flag becomes the NDCS flag

The two boxes "Find xt" and "EXECUTE" were called NDCS, in the previous diagram. Here they are exposed to show that non-default compilation semantics are found in a system-specific manner.

```
: process-xt    \ i*x xt -- j*x
  state @ 0 = if
    execute
  else
    dup ndcs?
    if  find-ndcs-xt execute  else  compile,  then
  then
;
```

The immediate flag has disappeared because all immediate words have non-default compilation semantics. They are immediate if the NDCS xt is the same as the for interpretation xt. The definition of immediate is more complicated in standards-speak, but comes to the same thing. An alternative implementation strategy may be to keep a separate

immediate flag, but we should not hide the basic idea that immediate words have non-default compilation semantics.

Several modern Forth systems have interpreter loops that would be easy to convert to the new requirements. Coming back to our three new words:
`IMMEDIATE?  Xt -- flag` ; return true if the word is immediate
`NDCS?  Xt -- flag` ; return true if the word has non-default compilation semantics
`NDCS,  i*x xt -- j*x` ; like **`COMPILE,`** but may parse. Used by words such as **`IF`**.

We can see that the conventional immediate flag in a word's header becomes the NDCS flag, set for all words that have non-default compilation semantics. Comparison of the interpretation xt and the NDCS xt gives us a basis for the word **`IMMEDIATE?`** The word **`NDCS,`** just hides the system-specific action of obtaining the NDCS action from an xt.

Using NDCS words

The NDCS words and the notation below allow us to construct NDCS words, both for system use and for general use. It is worth considering whether a library or an application may want to construct NDCS words. The most common case that we see is when an application needs a "Domain Specific Language" (DSL) which is Forth-based. Such a DSL may wish to provide interpreted as well as compiled versions of **`IF`** … **`ELSE`** … **`THEN`** and **`DO`** … **`LOOP`**.

In the past this type of notation has been shunned because it required state-smart words. Words that use NDCS correctly in two portions that do not test **`STATE`** are not state-smart. Therefore the reasons to avoid such notations only have to do with programming taste and overcoming the limitations of 20+ years of dogma. The dogma arose because we did not have the structures to separate interpretation and compilation actions, even though the Forth94 and Forth 2012 standards described compilation in those terms. Once we have Forth words to implement such ideas, we can move forward.

Here's a potential way of building NDCS words. They illustrate a conventional **`IF`** … **`THEN`** pair. The word **`NDCS:`** modifies the previous word to have the following non-default compilation semantics – it defines a nameless word and sets system-specific flags and data.

```
: IF      \ C: -- orig ; Run: x --
\ This is the traditional interpretation behaviour
  NoInterp  ;
ndcs: ( -- orig )  s_?br>,  ;     \ conditional forward branch

: THEN     \ C: orig -- ; Run: --
\ This is the traditional interpretation behaviour
  NoInterp  ;
ndcs: ( orig -- )  s_res_br>,  ;  \ resolve forward branch
```

To produce an interpreted version, the interpretation behaviour is simply replaced by the new version. The next example shows how a contentious notation such as **`S"`** and friends becomes non-contentious.

```
: S"            \ Comp: "ccc<quote>" -- ; Run: -- c-addr u
\ Describe a string. Text is taken up to the next double-quote
\ character. The address and length of the string are
\ returned.
  [char] " parse >syspad
;
ndcs: ( -- ) postpone (s") ",  ;
```

Words we need to consider

```
: set-compiler  \ xt --
\ Set xt as the compiler (by COMPILE,) of the last
\ definition. The word whose xt is given receives
\ the of the word it is to compile ( xt -- ).
\ Used to define optimisers.

: comp:         \ xt --
\ Starts a nameless word whose xt becomes the compiler
\ for the last definition.

: set-ndcs      \ xt --
\ Set xt as the NDCS action of the last definition.
\ The word whose xt is given to SET-NDCS has the stack
\ action: i*x -- j*x

: ndcs:         \ i*x xt -- j*x
\ Starts a nameless word whose xt becomes the NDCS
\ action for the last definition.

: IMMEDIATE     \ --
\ Mark the last defined word as immediate by
\ setting the NDCS flag making the NDCS xt the same
\ as the interpretation xt.

: IMMEDIATE?    \ Xt -- flag
\ Return true if the word is immediate.

: NDCS?         \ Xt -- flag
\ Return true if the word has non-default compilation
\ semantics.

: NDCS,         \ i*x xt -- j*x
\ Like COMPILE, but may parse. Used to perform the action
\ at compile time of NDCS words such as IF.

: SEARCH-NAME   \ c-addr len -- ior | xt 0
\ Perform the SEARCH-WORDLIST operation on all wordlists
\ within the current search order. On failure, just an ior
\ (say -13) is returned. On success, the word's xt and 0
\ are returned.
```

During a review on the Forth200x mailing list, nobody liked the acronym NDCS for these words. The phrase "special complation semantics" was much preferred instead. I have left NDCS in the paper because Forth 2012 refers to "non-default compilation semantics" throughout. When the standard uses the new phrase, then the words above can change names.

Consequences for compilation

We can now define what happens during compilation of a Forth word, i.e. what happens when a source token has been recognised/found and the system is in compilation state.

NDCS = Non-Default Compilation Semantics, i.e. a special word

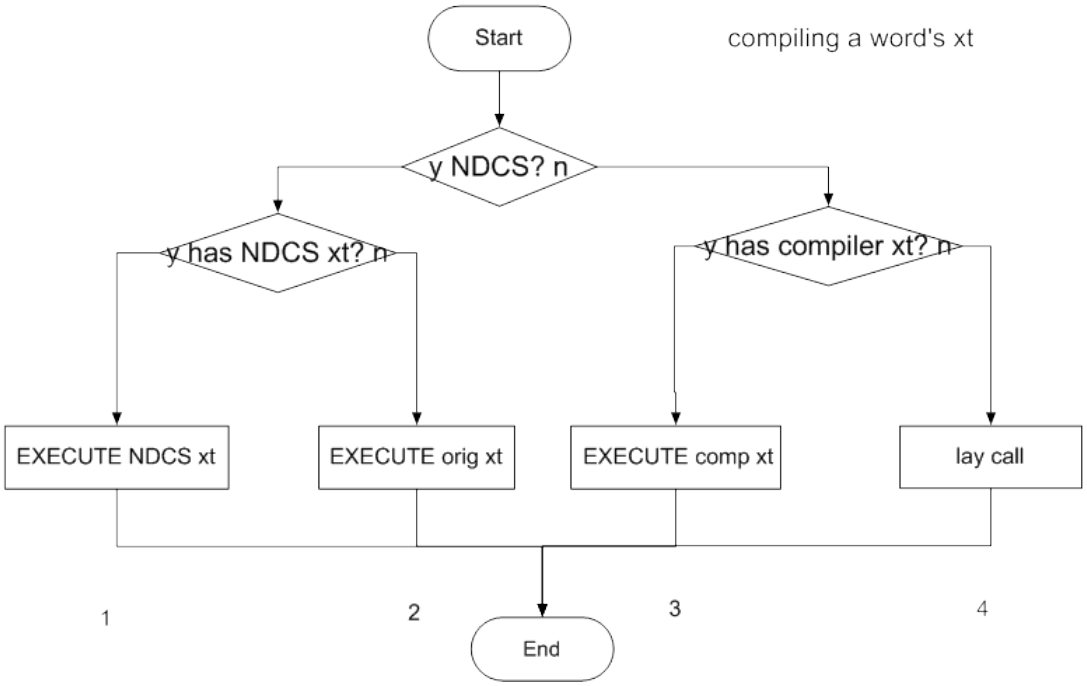


Illustration 4: Forth compilation

1. The word has an NDCS flag and an NDCS-specific action has been defined for it. The NDCS specific action is executed. These could be the compile-time actions of **`IF`** or **`S"`**.
2. The word has an NDCS flag but no NDCS-specific action has been defined for it. In this case the word's original xt is executed, corresponding directly to the current definition of an immediate word.
3. The word is normal and a code generator has been specified. The code generator is executed to lay down the required code.
4. The word is normal and no code generator exists. We just lay a Forth call to this word.

Cases 1 and 2 form the action of **`NDCS,`** in this paper. Cases 3 and 4 form the action of **`COMPILE,`** in the standard.

The test for an NDCS xt is optional. If a system can guarantee that all NDCS words have a separate xt for the NDCS portion, case 2 never happens and the check can be omitted. Similarly, systems without code generators can omit case 3.

Embedded and minimal systems

If we treat the NDCS flag as equivalent to the old immediate flag, then a minimal system can just provide cases 2 and 4 above. If such systems wish to provide both compilation and interpretation actions for words such as **`S"`** they can fall back to state-smart words, probably as they have always done.

Conclusions

The Forth94 standard (ANS) introduced the idea of "non-default compilation semantics" (NDCS) to the Forth world. However, the standard provided no facilities for dealing with NDCS words. NDCS makes immediate words a special case of NDCS. Simple changes to the Forth interpreter allow us to deal with and specify NDCS words. The classical Forth interpreter loop picture (Figure 1) needs a small change (Figure 3), and we need to introduce the words **`NDCS?`** and **`NDCS,`** to complete the picture.

Correct use of NDCS words also allows us to implement words such as **`S"`** without them being state-smart. This in turn permits us to define notations that have been deprecated for the last 20 years or so.

In implementing NDCS behaviour we find that immediate words are just a special case of NDCS. We can usefully remove the immediate flag and replace it with an NDCS flag.

It would be of benefit if we can find a name other than NDCS to describe "non-default compilation semantics".

Acknowledgements

Anton Ertl has tested my understanding of Forth standards for many years.

My belief that all standards contain bugs has sustained me over many years.

Anton Ertl, Bernd Paysan, Graham Smith and Gerald Wodni provided valuable comments on early drafts of the paper.

Security Considerations in a Forth Derived Language

Ron Aaron
Aaron High-Tech, Ltd.
HaSheminit 34, Ma'ale Adummim, Israel
t: +972 52 652 5543
e: ron@aaron-tech.com
w: https://8th-dev.com/

Abstract

Of major concern in modern application development is how to increase the robustness of program code while maintaining the programmers' productivity. Time-to-market directly affects ROI, but so do security problems arising from coding practices and tools. Our "8th" language, a Forth derivative, was conceived in part to address these issues.

Introduction

In seeking a cross-platform and secure programming language for a "secure application" which was to run on both mobile and desktop platforms, I looked for a language which was:

- easy to use
- truly "write-once, run-anywhere" (mobile and desktop alike)
- resistant to hacking
- proof against the most common security problems

Because I didn't find an appropriate language or set of tools, despite my best efforts to do so, I settled on creating a new language. Borrowing ideas from my former Reva Forth, which was also a Forth-derived language, I set out to address the above issues in the new language, "8th".

"Ordinary" security issues

Memory access:

In standard Forths, it is generally possible to access any memory in the system by means of `@` and `!`. Similarly, in C and C++, it is possible to assign an arbitrary value to a char * to do the same. While it is often useful to have such access, it is an open door for hackers to subvert a program. Therefore 8th does not permit arbitrary memory accesses. Instead, it uses strong typing along with specific memory-accessing types (strings and buffers, among others) to safely encapsulate memory accesses.

Memory allocation:

A common issue in C, C++, Forth, and others is memory allocation. There are several related issues: a) allocating an incorrect amount of memory, b) neglecting to properly initialize that

memory, c) neglecting to allocate memory at all, d) forgetting to de-allocate the memory when done, and e) de-allocating the memory more than once. Problems (a), (b) and (c), in particular, can also be security issues. The 8th solution is simply to not permit the programmer to allocate or deallocate memory (much like Java *et. al.*). The data-types native to 8th automatically manage any memory they require. They are also allocated from "pools" of that type, and they are reference-counted so that de-allocation of memory (and other resources) occurs when the item is no longer in use. This sort of "garbage-collection" amortizes the cost of the cleanup over the course of the application's run-time, similar to C or C++ (or Forth for that matter), and unlike Java whose GC can cause the application to pause for a significant amount of time at unpredictable intervals.

Stack smashing:

Possibly the most common exploit used by hackers is "stack smashing", or overflowing an item held on the stack and thus overwriting adjacent memory. This can be used to simply corrupt a value (and achieve some alternate code path), or it may be used to generate an alternate code address to return to, or even CPU opcodes to be executed.

Despite decades of battle against this particular foe, C and C++ remain quite vulnerable to it because the languages themselves make it easy to succumb to it. 8th makes it impossible to smash the stack, since data-items control access to their internal data, and ensure the access is within bounds, or dynamically grow as necessary. In no case is direct access to the CPU stack provided by any 8th words, unlike the standard Forth >r and r> which, often, are implemented as giving direct access to the CPU stack. In 8th, those words only access an auxiliary stack, so it is impossible to subvert the CPU stack.

Integer overflow:

While perhaps not as well-known, integer overflow continues to be an ongoing security issue in C and C++ in particular (as well as in some Forths). The plethora of integer types and the necessity to always be aware of those types, as well as the overuse and abuse of "casting" can lead to unexpected and extremely difficult-to-debug security issues. 8th handles this by automatically promoting an integer to a "big-integer" or "float" to a "big-float" as necessary. Adding 1 to 9223372036854775807 simply results in 9223372036854775808, and the underlying data type appears simply as a "number" to the user. This means that the programmer can be concerned solely with the correctness of the algorithm he or she is implementing, and needn't be worried about "gotchas" in the maths department.

Pointers:

Forth users are used to having generic "cells" on the stack, which may be treated as numbers or addresses (arbitrary memory access). C/C++ users rely heavily on pointers. The danger is that a pointer represents an open door into your data structures and perhaps your code as well. If an adversary can get access to a pointer to an internal structure, she or he can almost certainly find something "interesting" to do with it. Something which you will not approve of, most likely.

8th does not provide any pointers at all. This was hinted at in the “memory access” section. But it goes further, in that the data items returned by 8th to the user are opaque and it is not possible via plain 8th to access their internals. However, since the FFI (foreign function interface) must be able to pass pointers to external functions, it is possible for an external library to gain access via the FFI to internal structures. 8th attempts to reduce the footprint of this vulnerability by not passing any actual internal structures to external programs; however, a dedicated hacker might be able to find an opening. Therefore one must take extra steps to ensure external libraries are secured).

User input:

Another avenue for hackers to subvert an application is via uncontrolled (or more often, specifically controlled) user-input. In common with Forth, 8th allows users to enter arbitrary information via the REPL. The input in 8th is captured in strings, which are a self-adjusting data-type which can handle any amount of data (within system memory limits). In general, the REPL is not considered a vector for subverting the application. However, the application programmer can use **eval** to permit interpretration of arbitrary input text, or can even invoke the REPL from within an application.

To help the programmer avoid disaster, 8th makes it quite easy to limit the vocabularies (called “namespaces” in 8th) which are available to the REPL. Standard Forth has a similar facility. Thus one may control which words the user-interpreted input may access.

But unlike standard Forth, 8th uses JSON as its data-description language, primarily because it is so widely used and is easy to read. Further, 8th has “enhancements” to JSON which are intended as a convenience to the programmer. So for example, one may embed 8th code to be interpreted along with the JSON, in order to calculate some value at load-time. These enhancements are a potential avenue for hackers, and so a special word **json>** exists which interprets only standard JSON (and throws an exception if there is invalid JSON in the string).

In the usual case of user-input hacking, the hacker attempts to overflow a buffer in order to perform a stack or heap-crash. Because the strings used by 8th are self-adjusting, this tactic cannot succeed.

An additional protection 8th provides is “auto-wipe” buffers and strings. When one of those data types has **set-wipe** invoked on it, it will have its allocated memory wiped (zeroed out) before being deallocated. This is important in the case of a buffer holding decrypted information or an encryption key. The encryption words set the wipe flag automatically to ensure private or sensitive data are not leaked.

“Extraordinary” security issues

There are two more specific issues which 8th attempts to deal with: subversion via required libraries and validation of the application's code.

In any system where external libraries may be loaded at run-time (e.g. all the platforms 8th runs on), it is possible for an adversary to replace a “good” library with a “bad” one. Windows has been a non-stop target of such exploits, but neither macOS nor Linux are immune. 8th reduces the attack

surface by fully incorporating most of the libraries it requires into the runtime binary. This has the obvious disadvantages of increasing the size of that binary, as well as making it impossible to update the version of a library used in a particular 8th binary. However, it was felt that the advantages of significantly reducing the attack surface for this kind of hack outweighed those disadvantages. In particular, because 8th undergoes very regular updates, the incorporated libraries are also regularly updated as necessary.

The “Professional” edition of 8th also incorporates validation of the application as well as a measure of protection against decompiling or exposure of the programmers intellectual property. It does this by encrypting and signing the “deployment package” (DP) of code and resources.

In practice, the DP is a zip-file containing all the code and resources for a particular application. The Professional edition allows programmers to encrypt the DP (with AES-256-GCM) and sign the application with their specific PK keys (generated by the 8th system using Ed25519, not GPG etc.). Then when the 8th runtime unpacks the DP, it will fail if any part of it has been tampered with, providing a strong measure of confidence that the DP is legitimate. The DP's encryption makes it difficult (though not impossible) for an adversary to decrypt it and expose the intellectual property contained therein.

Conclusion

There is no rest for the weary.

We are continually seeking to improve the hack-resistance of 8th and improve its stability. At present, the FFI represents the single biggest entry point for hackers, and we are researching methods of hardening it.

Likewise, we would like to find a better method of encrypting the DP, to make it still more difficult for an attacker to decrypt. The weakness there is that the keys must be known at runtime, and so must be stored in a manner the 8th runtime can access on arbitrary hardware. We've not found any particularly robust solutions to that problem, so we are careful to not make any extreme claims for the encryption.

As new exploits are found in libraries we use or in other code, we work to find ways to offset them.

Finally, we are researching methods of confounding timing and other attacks against the encryption libraries we use (TomCrypt).

If you have comments, questions, or criticisms, please feel free to address them to me at the email listed at the start of this document.

RFC

Andrew Read (AR) / Ulli Hoffmann (UH)
EuroForth 2017

Forth: A New Synthesis

1. Introduction and objective

It is well-known that Forth scales-down well to the smallest platforms and applications. However, it is less obvious that Forth scales-up well to large applications or development projects. Our hypothesis is that some of the features of Forth that enable it to minimize so successfully, are constraints on the language in scaling up.

These days small code size and fast execution speed are relative rather than absolute merits. This project aims to produce a new synthesis of Forth that rebalances the requirement for scaling-down against the opportunity for scaling up.

We propose a “new synthesis” of Forth in a similar spirit to the Forth Modification Laboratory workshops.

Any new synthesis of a computer language runs the risk of being formed solely out of individual preferences and experiences. That remains the case with this project, but to provide guidance two governing principles have been adopted: biological analogy and disaggregation.

2. Principle: biological analogy

The biological cell provides a fascinating model of a complex system that processes stored information. With imagination, direct analogies can be drawn between the component parts of a cell and the component parts of a Forth system (Appendix I)

The most compelling rationale for consulting our understanding of the biological cell for ideas is that the biological cell is a proven-successful system that scales-down (to virus and single-celled life-forms), scales-up (to self-conscious Homo sapiens), and diversifies (to the different kinds of cell specialization within a single organism and to the multitude of life-forms on Earth), whilst broadly maintaining a common internal architecture of structure and function.

3. Principle: disaggregation

The project aims to identify the component parts of Forth and separate them, even when this separation may be to the detriment of efficiency. Experience suggests that proper disaggregation by itself frequently solves existing problems. Table 1 makes some specific proposals

	Traditional Forth	**New synthesis**	**Implications**
1	Forth words have direct access to byte-by-byte contiguous memory	Interpose an allocation based memory management system between system memory and the Forth system	Allows further disaggregation of the dictionary and heap (e.g. S" or colon data) data structures
2	Dictionary and the heap may be built up together in memory	Impose the separation of the dictionary and the heap	The dictionary and the heap may independently be rewound or modified
3	Dictionary may contain headers and code	Use the allocable memory mode to separate headers and code	All Forth words can be erased and rewritten
4	The input stream is locked into the INTERPRET loop	Interpose an extendible text processor that exclusively handles the input stream	Rather than using parsing words, the text processor provides a general model for extending the language
5	Both interpret or compile states are handled by the INTERPRET loop and state-smart words	Disaggregate by eliminating state: everything is compiled, but depending on context some compilations may immediately be executed and then rewound	Removes ambiguity in word definitions and divergence in the way the language is extended

Table 1. New synthesis disaggregation proposals

4. Practical experiments and observations

UH prepared two model systems for experimentation since EuroFORTH 2016

i. A Forth system built on the GO language that implements an allocable memory model foundation for a Forth system. The reason for wanting this memory model was to enable the redefinition of dictionary Forth words with new code. We experimented along two dimensions, as follows

What to do with old instances of the word? / what to do with recursive references?	**Not true recursion - reference the prior word definition, if available**	**True recursion – reference the new definition**
Old instances retain old code	Traditional Forth	“Mixed”
Replace all instances with the new definition	n/a	LISP-like

One interesting implication of the LISP-like model is that it might allow a Forth system to completely replace its dictionary with a new set of definitions. This is explored further in future directions, below.

ii. An extension to Forth that “opens” the INTERPRET loop and interposes an extendible text processor. The key finding was that with this model there is no need for recognizers as their function is already a natural part of the system. See "Recognizers Dissolved", Ulli Hoffmann, EuroForth 2017

5. Conceptual next steps

We are interested in developing the new synthesis, guided at a high level by the biological analogy. Some specific thoughts are as follows

i. All living organisms on the Earth share a common (i.e. highly evolutionary preserved) fundamental core (DNA/RNA, the genetic code, proteins, cells, etc.). Yet the complexity and diversity of life on Earth depends on the variation between their biological systems. For example, the fore limb apparatus can be adapted into arms and hands by primates, into legs by four-legged animals, or into wings by birds.

Should the same be true for Forth systems? A common operating framework (WORDS, STACKS and BLOCKS?) with a minimal dictionary in

all Forth systems, but then wide variation at higher levels between systems in both the vocabulary of available words and their definitions. This would be contrary to the ANSI approach, but could it be sympathetic to Chuck Moore's original vision?

ii. All multi-cellular organisms grow from a single cell, typically the egg.

Should a similar approach be taken to 'growing' Forth onto a new target? For example: The Forth 'egg' in ROM implements the minimal framework and dictionary. It has the capability to read an input-stream from a small EEPROM, but otherwise no input/output firmware. The EEPROM contains plain-text Forth code in byte-by-byte format. The egg reads from the EEPROM and 'grows' by implementing further i/o and other firmware, including if necessary a FAT file system on SD card. An SD card may contain further Forth files with application software. At each stage the Forth system will be re-engineering its input stream apparatus with a more sophisticated version.

iii. Biological systems are not static – they continue to evolve.

Should Forth be the same? For example, rather than standardizing more and more of the language, should the curation of Forth encourage "forking" of new versions of Forth, even if most die out.

iv. Some biological systems are super-organisms, such as colonies of ants or bees. Individual organisms are specialized into different roles, and the loss of one individual does not jeopardize the hive.

Should some Forth systems strive to adopt a distributed model, with a "queen" system spinning off specialized, limited capability "worker" Forth systems on very low cost peripheral processors.

v. Biological organisms are single-purpose: a dog is a dog, a cat is a cat and a man is a man. Conventional computer systems are now multipurpose: a desktop computer is a word-processor and a CAD platform and a music player. This results in the "layered" operating system / middle-ware / package approach with standardized interfaces, all programmed by thousands of different individuals

Is Forth better off devoting itself to single-purpose systems that are developed by individuals or small teams? In this domain inter-compatibility, wide library support, the layers and standardized definitions are not critical. (See also unikernal.org for some interesting thinking on this topic.)

In this model, ANSI Forth remains important for thought leadership and communication, but not for its word definitions *per se*.

In exploring these ideas we will certainly need to address at least two questions.

Firstly, what is the basic operating framework needed across all Forth systems, analogous to the highly conserved structures of the biological cell. For example, memory allocation, a dictionary, a heap? Specified at what level of detail?

Secondly, what should a Forth ‘egg’ contain? In common with a biological egg it should contain the operating framework and a small dictionary. What needs to be in the dictionary? How is the egg ‘primed’ at power on, etc..

6. Conclusion

We are exploring the future of Forth in the spirit of the Forth Modification Laboratory. Our work is motivated by the observation that Forth has been left behind as computer systems have scaled up, and by our optimism that somehow Forth might still be refreshed and reinvented in interesting new ways.

Two principles are guiding us. Firstly, on modern hardware minimalization of code size and optimization of execution speed are no longer extreme necessities. We are prepared to make some sacrifices on these dimensions for the sake of a more disaggregated architecture that has the potential to scale up more easily. In other words we want to be more aware of the degrees of freedom on any given target and make conscious decisions that may differ from historical precedent.

Secondly, we note an exciting analogy between Forth and biological systems and wish to see if this inspiration can guide us to a “Cambrian Explosion” in the diversity and sophistication of Forth Life.

Appendix I – Forth biological analogy

	Cell	**Forth**	**Analogy**
1	DNA is the genetic material that defines the functioning of the cell	Source code is the material that defines the functioning of the application	DNA <=> Source code
2	The DNA of a cell is located in the chromosomes	The source code of a Forth application is located in blocks	Chromosome <=> Blocks
3	Foreign DNA may be expressed in a cell (viruses / genetic engineering)	The input stream may be directed to the keyboard, serial line or other port	Foreign DNA <=> Keyboard input
4	DNA that is to be expressed is translated into mRNA	Source code that is to be run is directed to the input stream	mRNA <=> Input stream
5	Gene expression is mediated by controlling the transcription of DNA into mRNA	Source code may be chosen by directing the input stream to the relevant blocks	Gene expression control <=> Redirection of the input stream
6	mRNA is translated into proteins; proteins make the cell function	Source code is compiled into words; words make the application function	Proteins <=> Words
7	Proteins are comprised of amino acids	Words are comprised of assembly language instructions	Amino acids <=> assembly language instructions
8	Translation takes place at the active sites of ribosomes	Source code is compiled into words by the compiler	Ribosomes <=> Compiler
9	Translation is mediated by tRNA molecules that parse the sequences of the genetic code	Compilation is mediated by recognizers that parse the input stream	tRNA <=> Recognizers
10	mRNA coding regions begin with start codons and end with stop codons	The input stream defines words with colon and semi-colon	Start codon <=> Colon Stop codon <=> Semi-colon

EuroForth 2017
In Cahoots
Forth, GTK and Glade working secretly together

N.J. Nelson B.Sc. C. Eng. M.I.E.T.
R.J. Merrett B.Eng.
Micross Automation Systems
Unit 6, Ashburton Industrial Estate
Ross-on-Wye, Herefordshire
HR9 7BW UK
Tel. +44 1989 768080
Emails: njn@micross.co.uk rjm@micross.co.uk

Abstract
Forth is a very good language for working with other tools and libraries. In this paper we will introduce some techniques to make GTK and Glade work with Forth as seamlessly as possible.

1. Introduction
Cahoots in this instance does not refer to that well-known town in New York state on the banks of the Hudson River.[1] Its alternative meaning is when two or more parties conspire to act together secretly. The parties in this case are:

Forth

Our favourite language for conciseness, readability, and in this case ease of interoperability.

GTK

This is one of several open source toolkits for graphical programming.
It was chosen because it is being very actively developed, and has a straightforward interface method.

Glade

This is a graphical design tool for GTK. It produces XML code that can be loaded by GTK as required.

2. Compilers, versions and targets
We have been working with the MPE VFX Forth compiler, using GTK+ version 3, for the Ubuntu Linux operating system on both single-board computers with ARM processors, and industrial PCs with x86 CPUs.

1 Roger S. Brody RDP, Chairman of the Smithsonian Museum Philatelic Research Committee.

3. Library bindings

The MPE compiler came with a basic set of bindings to the GTK+ V2 libraries. We adapted these to GTK+ V3, added many new bindings and enumerations as needed, and removed features that we concluded were dead ends. Since GTK+ is written in C, the bindings are very straightforward e.g.

```
extern: void  "c" gtk_button_set_image( int * button, int * image );
```

4. Maintaining interactivity

A major difference between MPE VFX for Linux and MPE VFX for Windows, is that the Linux version runs straight from a standard Linux terminal. This means that interactivity is lost as soon as the GTK message pump starts. Since we regard interactivity as an essential debugger tool, it was necessary to restore it somehow.

Because there are always small differences needed in behaviour between programs when run in debug and when run normally (e.g. so that logon is not necessary every time you run in debug), we always create two different build files which set or clear a debugging flag e.g.

Debug

```
TRUE VALUE DEBUGGING          \ Set debugging mode
include PackingLabel.bld      \ Main build file
.BadExterns                   \ Report any library failures
PACKINGLABELMODULE            \ Run in debug
```

Compile

```
FALSE VALUE DEBUGGING         \ Clear debugging mode
include PackingLabel.bld      \ Main build file
PACKINGLABELMODULE            \ Run, to get it all set up
save PackingLabel             \ Save ELF file
```

This debugging flag can then be used to start the GTK message pump in a separate thread, when in debug mode.

```
TASK MAINTASK \ For GTK in debug
: MAINACTION ( --- ) \ GTK action, when in debug
  gtk_main              \ Start the message pump
;
```

```
...
  INIT-MULTI                              \ Initialise the multitasker
  MULTI                                   \ Start the multitasker
  DEBUGGING IF                            \ Running in debug
    ['] MAINACTION MAINTASK INITIATE      \ Start main in separate thread
  ELSE
    gtk_main                              \ Start the message pump
  THEN
...
```

Forth commands can then continue to be run from the Linux terminal, when in debug mode.

5. Structuring the Glade files

The VFX Forth comes with a nice wrapper which both loads the Glade XML file, and resolves the signals, in one operation. However, this is restricted to a single Glade file, and in a real application a single Glade file soon becomes too big to handle. We started to split the files by function, which also makes for re-usability. However, a single builder object is used for all the Glade files, so that all windows, dialogs and other features can be handled together. We also separated the file load from the signal resolution, because of the next feature.

```
: LOADGLADE { | be[ cell ] -- } \ Loads the glade files
  gtk_builder_new -> PBUILDER                              \ Create builder
  PBUILDER IF
    be[ OFF
    PBUILDER Z" SW1015.glade" be[ gtk_builder_add_from_file      \ Main glade
    PBUILDER Z" Logon.glade"  be[ gtk_builder_add_from_file AND  \ Logon glade
\   PBUILDER Z" next file .." be[ gtk_builder_add_from_file AND  \ More ...
    0= IF
      be[ @ 2 cells + @ .z$                                    \ Error string
      be[ @ g_error_free
      PBUILDER g_object_unref
    THEN
  THEN
;
```

6. Handling the handles

In order to do anything with a GTK+ widget, you need to know its magic number - the equivalent of a "handle" in Windows. When designing in Glade, you specify a name, then at run time you can ask the "builder" into which you loaded the Glade file, for the number of an object, from its name. You can then store it in a value (typically of the same name).

```
: GETHANDLE ( z$---h ) \ Get handle of builder element from name
  PBUILDER SWAP gtk_builder_get_object
;
```

```
Z" Mybutton" GETHANDLE -> MYBUTTON
```

That was OK for simple applications, but then we soon realised that we were typing the name of every widget three times before we even used it - once in the Glade design, once to declare the value, and once to get the magic number.

In any other language other than Forth, you are stuck with that.

But as so often happens, the unique ability of Forth to do things during compilation time as well as during run time, comes to the rescue. We soon discovered that it's possible to get the builder to create a list of all objects, which can then be scanned for names.

```
: MAKEGLADENAMES { | pslist pobject -- } \ Create values for every object
  PBUILDER gtk_builder_get_objects -> pslist   \ Make list of objects
  pslist g_slist_length 0 ?DO                 \ For all objects
    pslist I g_slist_nth_data -> pobject      \ Get data
    pobject gtk_buildable_get_name            \ Get name
    pobject ZVALUE                            \ Create value for each name
  LOOP
  pslist g_slist_free                         \ Free list
;
```

This uses a very cunning feature - the ability to create Forth values automatically.

```
: ZVALUE ( zname, ival --- ) \ Creates a new value with name and initialisation
  SWAP ZCOUNT ($CREATE)
    , ['] valComp, set-compiler
  interp>
    valInterp
;
```

Note: you need an up-to-date version of VFX to make this work.

All that is necessary during a debug, is to call both LOADGLADE and MAKEGLADENAMES during the compile, and all the values are ready for you to use. However, when you then run an executable, you've got the value names, but not their magic numbers. It's necessary to distinguish between debug and normal run mode again, to load the numbers when necessary.

```
: SETGLADEVALS { | pslist pobject -- } \ Set values for glade objects
  PBUILDER gtk_builder_get_objects -> pslist   \ Make list of objects
  pslist g_slist_length 0 ?DO                  \ For all objects
    pslist I g_slist_nth_data -> pobject       \ Get data
    pobject gtk_buildable_get_name             \ Get name
    zcount search-context IF                   \ Name is in dictionary
      >body pobject SWAP !                     \ Set value
    THEN
  LOOP
  pslist g_slist_free                          \ Free list
;
```

Notice that it's rather important to make sure the Glade widget names are Forth-unique, otherwise strange things happen.

Now in our initialisation, we simply include

```
...
  PBUILDER 0= IF              \ Glade not loaded
    do_gtk_init               \ Initialise GTK
    LOADGLADE                 \ Load glade files
    SETGLADEVALS              \ Set values for glade objects
  THEN
  RESOLVEGLADE                \ Resolve Glade signals
...
```

7. To do - automatic resizing

Most of the applications that we have written recently have been for touchscreens, in "kiosk" mode i.e. the operator has no access to the underlying operating system. This is far easier to achieve in Linux than it is in Windows, where it has become more and more difficult to eliminate the infuriating little things that Windows "pops up" without being asked.

Of course, any kiosk applications must run full screen. But screen resolution may vary. In Windows, the size and position of each element is under the exact control of the programmer. We used to design each display based on the minimum plausible resolution (say, 800 x 600 pixels) then use a Forth word that ran through all possible windows, and resized and repositioned them according to the actual screen resolution. The fonts were also resized to match the vertical resolution.

```
: CTRL2RES { ahctrl -- } \ Set size and position of control
  ahctrl HIROANIM @ =                     \ Animation control
  ahctrl HIRONSETUP @ = OR IF             \ or, superimposed button
    ahctrl                                \ Move only, do not size
    ahctrl WINDOW-X ahctrl WINDOW-Y 0 0
    RESVAR-XYWH 2DROP WINDOW-AMOVE
  ELSE                                    \ All other controls
    ahctrl 0                              \ Resize and move
    ahctrl WINDOW-X ahctrl WINDOW-Y
    ahctrl WINDOW-WIDTH ahctrl WINDOW-HEIGHT
    RESVAR-XYWH
    SWP_NOZORDER SWP_NOSENDCHANGING OR
    WINSETWINDOWPOS DROP
    ahctrl RESVAR-FONT                    \ New font
  THEN
;
```

```
: WIN2RES ( Whndl --- ) \ Set size and position of window and all controls
  DUP 0 0 WINDOW-AMOVE
  DUP CURRHORZRES @ CURRVERTRES @ WINDOW-ASIZE
  DUP RESVAR-FONT
  WINGETTOPWINDOW ?DUP IF
    BEGIN
      DUP CTRL2RES
      GW_HWNDNEXT WINGETNEXTWINDOW ?DUP
      0=
    UNTIL
  THEN
;
```

Unfortunately, this is not so easy in GTK. There is a heavy emphasis on automatic sizing of widgets. Before rendering, each container widget asks all the contained elements right down the chain, for the size they'd like to be. This can be a minimum size that has been set in Glade, but it is usually not possible to set a maximum size. So if you have a label widget, and increase the length of its string or the size of its font, and it will automatically resize itself, which in turn will resize its container, and so on up the chain. If the topmost window is now too big for the screen resolution, it will create scrollbars for itself, and worse still, reveal the Ubuntu toolbar.

We have still not fully resolved this problem, and for the time being there is the very irritating and time-consuming process of making a different set of Glade files for each screen resolution.

We're sure we cannot be the only people with this issue, and suggestions are very welcome.

8. Conclusion

Only in Forth, can one successively improve the compilation process so that each application becomes more compact and easier to write.

NJN
RJM
30/8/17

cryptoColorForth

less is more

Presentation at EuroForth 2017, Bad Vöslau, Austria,

Howerd Oakford www.inventio.co.uk

Aim

To create the simplest possible secure communication, data storage and user interface for the You-Me Drive (YMD).

For more details on the YMD please go to :http://you-me.one/ , a summary is :

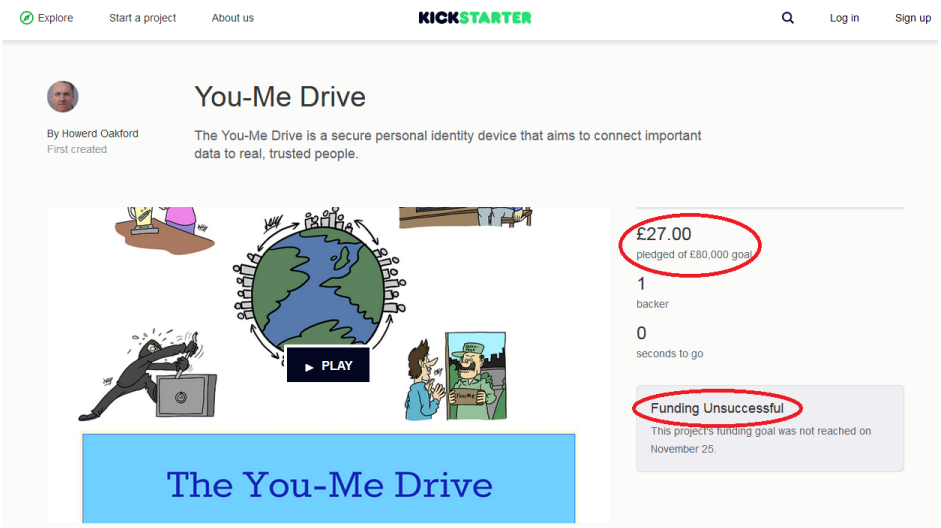
“The You-Me Drive is a secure personal identity device that aims to connect important data to real, trusted people”.

It does this by providing a WiFi USB drive, peer to peer networking software, multi-signed file encryption and a user interface that boots directly on a PC.

Kickstarter

My first, and almost certainly last, Kickstarter project was an attempt to get funding to speed up the development of the You-Me Drive.

https://www.kickstarter.com/projects/inventio/you-me-drive



Even though the Kickstarter campaign was a failure, it provided the impetus to document the project : http://you-me.one/You-Me_Drive_2016Oct01.pdf
I also learned how hard it is to explain a complex idea in a way that appeals to a non-technical audience.

So development continues as time allows, with the current sub-project being the **colorForth** inspired software infrastructure.

Why colorForth?

The YMD requires the highest possible level of security, and this rules out a conventional operating system.

Commercially successful operating systems are designed to lock users into a particular manufacturer's brand of complexity, in order to maximise profitability. The complexity generated to achieve this is so high that it is not possible to evaluate potential security weaknesses.

There are several less complex operating systems, but my personal favourite amongst the FOSS ones has always been **colorForth** - *less is more* in this context.

User-friendly colorForth

There are several versions of **colorForth** published online, Chuck Moore's original (colorforth.com), and many others derived from this, the SourceForge version and the GreenArrays *ArrayForth* GA144 development environment. Since they are all based on Chuck's original version they all share some of Chuck's original design decisions – for example the use of the ANS Forth standard names **or** and **?dup** to have non ANS functionality, the use of 32 bit cell addressing, and the use of the **eax** register for the Top Of Stack instead of **ebx**.

Some of the **colorForth**s have optional QWERTY keyboard text entry, but my view is that this should be available for user text entry only, and not included as part of the programming environment. Also, most **colorForth**s require actual floppy disk hardware to function.

So **cryptoColorForth** (so far) makes some user-friendly changes :

1. ANS Forth names – **or** → **xor** , **?dup** → **qdup**
2. Byte addressing for **@**, **C@**, **!** and **C!**
3. BIOS sector read/write → operation from USB drive
4. QWERTY text input

Further colorForth development

There are many useful functions defined in ANS-like Forths for the x86 architecture - crypto libraries using big numbers, modular exponentiation, TEAN, RSA etc.
With the simple changes listed above, and a meta-data replacement for the conventional file system, **cryptoColorForth** can be made to work with this source code, allowing these building blocks to be imported easily.

Summary

colorForth has an extraordinary "power to weight ratio", its small size (12K bytes for the kernel) and extreme simplicity makes it the ideal platform to develop secure applications, with the You-Me Drive being one example.

Howerd Oakford, 31st August 2017

A multi-tasking wordset for Standard Forth

Andrew Haley
Consulting Engineer
8 September 2017

Background

Forth has been multi-tasking for almost 50 years. It's time to standardize it.

- Chuck Moore's early Forths had a simple and efficient multi-tasker
- This was refined by others at Forth, Inc. and eventually became the core of polyFORTH
- Many Forths have used a version of this multi-tasker since then, and because of that there is some practical portability of multi-tasking programs between Forth systems. These include products from MPE and Forth, Inc. as well as free systems such as F83

Goals

- To make it possible to write multi-tasking programs in Standard Forth
- These standard multi-tasking programs will run unaltered on both co-operative (round-robin) and time-sliced schedulers, on hosted and freestanding systems

Design criteria

- No innovation!
  - Wherever possible, use established practice from Forth systems
  - Where no established practice exists in Forth, take inspiration from other programming languages, especially C
- This should be a low-level wordset
  - Standardize the most basic elements of multi-tasking, eschewing more complex objects such as queues and channels. These can be provided by libraries, based on this wordset

Design criteria

- Completeness
  - This wordset must provide everything that is necessary to write libraries and multi-tasking programs without such needing to use carnal knowledge
- Simplicity
  - Given that this is a Forth standard, simplicity hardly needs mentioning, but it must be paramount after completeness
  - Simplicity mostly “falls out” of the design as a result of following common Forth practice

Design criteria

- Efficiency
  - While the greatest possible efficiency will always result from a system-specific wordset, we can get very close with a standard wordset
  - This wordset should work well on large multi-core systems but not impose a significant burden on very small single-core systems
- Portability
  - The wordset shouldn't require anything that is unavailable on a system that is capable of realistic multi-tasking. This means that the wordset should be usable on a machine with some kbytes of memory, not megabytes

Round-robin versus pre-emptive scheduling

- One of the surprising things (well, it surprised me!) was the realization that we need to say hardly anything about the differences between round-robin and pre-emptive schedulers
  - We make no guarantees about forward progress (doing so in a portable standard in a meaningful way is almost impossible) so it's not necessary to discriminate between these
  - Non-normative language must point out that on some systems you need to PAUSE or perform I/O from time to time, but that's all
  - Programs written with this wordset will work well with either type of scheduler

Memory ordering

- We have to say something about what happens when more one task accesses the same memory at the same time
- Real systems have some surprising behaviours when you do this. These include, but are not limited to
  - *Word tearing*, where a fetch sees a partial update of a multibyte word
  - Memory updates to different cells appear in different orders to different tasks
  - Memory reads can appear to go backwards in time, so that a counter is not monotonic
  - Memory can temporarily have unexpected values.
  - Many other things

Memory ordering: SC-DRF

Or, “sequentially consistent / data race free.”

- I believe that the best memory ordering model for Forth is SC-DRF.
  - The best reference for this is Hans Boehm, *Foundations of the C++ Concurrency Memory Model*, www.hpl.hp.com/techreports/2008/HPL-2008-56.pdf
  - Hans Boehm: “IMHO, the closest we have [to a language-independent memory model] that is actually solid and understandable is the basic DRF model, with undefined semantics for data races.”

Memory ordering: SC-DRF

Or, “sequentially consistent / data race free.”

- A *data race* is defined as a concurrent (non-atomic) access to shared memory
- SC-DRF allows tasks to use relaxed memory ordering locally, but requires them to use SC atomic operations when communicating with other tasks
- We give *no* semantics to programs with data races. The hardware might do all manner of things. We don't have to care: a data race might be benign on some hardware, but it won't be portable
- This isn't the dreaded nasal daemons: we only have to warn that tasks may observe misordering, word tearing, apparent loss of causality, and so on

Memory ordering: SC-DRF

Or, “sequentially consistent / data race free.”

- Sequential consistency, defined by Lamport, is the most intuitive model. Memory operations appear to occur in a single total order (i.e., atomically); further, within this total order, the memory operations of a thread appear in the program order for that thread
- We could define all Forth memory operations to be SC, but this would seriously restrict many compiler and hardware optimizations
- The best route is to allow tasks to use relaxed memory ordering locally, but require them to use SC atomic operations when communicating with other tasks

Memory ordering: SC-DRF

Or, “sequentially consistent / data race free.”

- A program which has no data races can be proved equivalent to a program in which every fetch and store are SC, i.e. they appear to all threads to happen in the same order
- This is easy for programmers to understand and it is reasonably easy to specify
- Other weaker memory models exist, but such mixed memory models become far more complicated and unintuitive

Creating a task

TASK <taskname> [polyFORTH]

Define a task. Invoking taskname returns the address of the task's Task Control Block (TCB).

/TASK (- n) [new]

n is the size of a Task Control block. [This word allows arrays of tasks to be created without having to name each one.]

CONSTRUCT (addr --) [polyFORTH]

Instantiate the task whose TCB is at addr. This creates the TCB and and possibly links the task into the round robin. After this, user variables may be initialized before the task is started

Starting a task

ACTIVATE (xt addr –) [polyFORTH]

Start the task at addr asynchronously executing the word whose execution token is xt. [This differs from Forth, Inc. practice, which uses the “word with an exit in the middle” technique of DOES>.]

What should we say about a task which reaches the end of this word, i.e. it hits the EXIT ? Traditionally, Forth systems would crash, and in order to prevent that you'd have to end an activation with

```
BEGIN STOP AGAIN
```

IMO, we'd be better saying that the task terminates

USER variables

A programmer may define words to access variables, with private versions of these variables in each task (such variables are called “user variables”).

USER (n1 --) [polyFORTH]

Define a user variable at offset n1 in the user area.

+USER (n1 n2 -- n3) [polyFORTH]

Define a user variable at offset n1 in the user area, and increment the offset by the size n2 to give a new offset n3.

#USER (– n) [polyFORTH]

Return the number of bytes currently allocated in a user area. This is the offset for the next user variable when this word is used to start a sequence of +USER definitions intended to add to previously defined user variables.

USER variables

A programmer may define words to access variables, with private versions of these variables in each task (such variables are called “user variables”).

HIS (addr1 n -- addr2) [polyFORTH]

Given a task address addr1 and user variable offset n, returns the address of the ref- erenced user variable in that task's user area. Usage:

<task-name> <user-variable-name> HIS

- This is very useful for initializing user variables before a task runs

STOP and **AWAKEN**

- These have been present in some form since the earliest days of Forth
  - **STOP** blocks the current task unless or until **AWAKEN** has been issued
  - Calls to **AWAKEN** are not “counted”, so multiple **AWAKEN**s before a **STOP** only unblock a single **STOP**
  - A task invoking **STOP** might return immediately because of a "leftover" **AWAKEN** from a previous usage. However, in the absence of an **AWAKEN**, its next invocation will block
  - **STOP** is the most OS-independent low-level blocking primitive I know of: it is a leaky one-bit semaphore
  - **STOP** and **AWAKEN** can fairly easily be used to create locks, blocking queues, and so on
  - **STOP** and **AWAKEN** correspond to BSD UNIX's `_lwp_park(2)` and `_lwp_unpark(2)`

Atomic operations

All of these are data race free

ATOMIC@ (a-addr -- x)[new]

Atomically load x from a-addr. The load is sequentially consistent. Equivalent to C11's atomic_load().

ATOMIC!(x a-addr --)[new]

Atomically store x at a-addr. The store is sequentially consistent. Equivalent to C11's atomic_store().

- These words are part of the total order so can be used for synchronization between threads

Atomic operations

All of these are data race free

`ATOMIC-XCHG` (x1 a-addr – x2) [new]

Atomically replace the value at a-addr with x1. x2 is the value previously at a-addr. This is an atomic read-modify-write operation. Equivalent to C11's atomic_exchange().

`ATOMIC-CAS` (expected desired a-addr – prev) [new]

Atomically compare the value at a-addr with expected, and if equal, replace the value at a-addr with desired. prev is the value at a-addr immediately before this operation. This is an atomic read-modify-write operation. Equivalent to C11's atomic_compare_exchange_strong().

- These words are part of the total order so can be used for synchronization between tasks

redhat.

GET and RELEASE

MutExes provide mutual exclusion

`MUTEX-INIT` (addr) [new]

Initialize a mutex. Set its state to released.

[In polyFORTH, this was just `0 addr !` .]

`/MUTEX` (– n) [new]

n is the number of bytes in a mutex.

[In polyFORTH, a mutex was a simple variable.]

redhat.

GET and RELEASE

MutExes provide mutual exclusion

`GET` (addr --) [polyFORTH]

Obtain control of the mutex at addr. If the mutex is owned by another task, the task executing GET will wait until the mutex is available.

[In a round-robin scheduler, this word executes PAUSE before attempting to acquire the mutex.]

`RELEASE` (addr –) [new]

Relinquish the mutex at addr

- These words are part of the total order

redhat.

And also...

`PAUSE` (–) [polyFORTH]

Causes the execuiting task temporarily to relinquish the CPU.

- In a system which uses round-robin sheduling, this can be used to allow other tasks to run
- However, this isn't usually needed because I/O causes a task to block. All words which do I/O should, unless they are extremely high priority, execute PAUSE

redhat.

MINOΣ2 — A GUI for net2o

Widgets and Layout Engine

Bernd Paysan

EuroForth 2017, Bad Vöslau

4 Years after Snowden

What has changed?

Politics
- Fake News/Hate Speech as excuse for censorship #NetzDG
- Crypto Wars 4.0: Discuss about ban of cryptography
- Legalize it (dragnet surveillance)
- Kill the link (EuGH and LG Humbug)
- Privacy: nobody is forced to use the Interwebs (Jim Sensenbrenner)

Competition
- faces Stasi–like Zersetzung (Tor project)

Solutions
- net2o starts becoming useable

Outlook from 2013

- The next presentation should be rendered with MINOΣ2
- Texts, videos, and images should be get with net2o, shouldn't be on the device
- Typesetting engine with boxes and glues, line breaking and hyphenation missing
- a lot less classes than MINOΣ — but more objects
- add a zbox for vertical layering
- integrated animations
- combine the GLSL programs into one program?

MINOΣ2 vs. MINOΣ

Rendering: OpenGL (ES) instead of Xlib, Vulkan backend planned
Coordinates: Single float instead of integer, origin bottom left (Xlib: top left)
Typesetting: Boxes&Glues closer to LaTeX — including ascender&descender
Glues can shrink, not just grow
Object System: Mini–OOF2 instead of BerndOOF
Class number: Fewer classes, more combinations

MINOΣ2 Widgets

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

actor base class that reacts on all actions (clicks, touchs, keys)
widget base class for all visible objects
glue base class for flexible objects
tile colored rectangle
frame colored rectangle with borders
text text element
edit editable text element (`text with cursor`)
icon image from an icon texture
image larger image
animation action for animations
canvas vector graphics (TBD)

MINOΣ2 Boxes

Just like LaTeX: Boxes arrange widgets/text

hbox Horizontal box, common baseline
vbox Vertical box, minimum distance a baselineskip (of the hboxes below)
zbox Overlapping several boxes
grid Free widget placements (TBD)

There will be some more variants for tables and wrapped paragraphs

MINOΣ2 Displays

Render into different kinds of displays

texture Into a texture, which can be used as image, also used as viewport (TBD)
display To the actual display

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
icon draw items of the icon texture
thumbnail draw items of the thumbnail texture
image images with one draw call per image
text text round
marking cursor/selection highlight round

Bonus page: BlockChain

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

Literature&Links

Bernd Paysan *net2o fossil repository*
https://fossil.net2o.de/net2o/

A Recognizer Influenced Handler Based Outer Interpreter Structure

Ulrich Hoffmann

many pictures taken from leo brodies famous book "starting forth" (c) forth, inc

outer interpreter: what needs to be done?

token →

is it a word? — run it

is it a number? — push it

haeh? — consult user

over view

- recognizers
- outer interpreter: what needs to be done?
- handlers
  - idea
  - code
  - design options
    - possible stack effects
    - haeh?
    - token scanning
    - search order
- summary
- disussion

outer compiler: what needs to be done?

is it a word?

is it a immediate? — run it

compile it

is it a number? — compile push it

haeh? — consult user

recognizers

- new extensible outer interpreter[1] structure proposed by mathias trute
- on its way to become a standard's committee supported proposal
- interpret/compile/postpone structure for syntactic classes that describes their treatment in the outer interpreter
- stack structure for combining recognizers

[1] http://amforth.sourceforge.net/pr/Recognizer-rfc-D.html

outer interpreter: extensions

is it a word? — run it

is it a number? — push it

is it a float? — fpush it

haeh? — consult user

outer interpreter: what needs to be done?

"the text interpreter scans the input stream, looking of strings of characters separated by spaces."

scan for token

handle it

outer interpreter: extensions

is it a word? — run it

is it a number? — push it

is it a char? — push it

haeh? — consult user

outer interpreter: extensions

is it a word?	run it
is it a number?	push it
is it a hex ?	push it
haeh?	consult user

outer interpreter: extensions

is it a word?	run it
is it a number?	push it
is it a char?	push it
is it a hex ?	push it
is it a float?	fpush it
haeh?	consult user

handlers idea

- give the token to a list of handlers one handler at a time until one can cope with it
- if a handler can cope with it, it does it and reports
- if it cannot, it reports

handlers code

```
Variable handlers

: interpret ( -- )
    BEGIN  parse-name dup
    WHILE
      handlers @ length handle
      0= IF -13 throw THEN
    REPEAT 2drop ;
```

handlers code

```
Variable handlers

: interpret ( -- )
    BEGIN  parse-name dup
    WHILE
      handlers @ length handle
      0= IF -13 throw THEN
    REPEAT 2drop ;
```

and ***state?***

handlers code interpret words

```
\ interpret words in forth wordlist

:noname ( c-addr u1 -- i*x true | c-addr2 u2 false )
   2dup  forth-wordlist  search-wordlist
    IF  nip nip execute true EXIT THEN false ;
```

difference to recognizers?

- 1 task vs. 3 in 1
- immediate copeing vs. later execution

handlers code
compile words

```
\ compile words in forth wordlist

:noname ( c-addr u1 -- i*x true | c-addr2 u2 false )
   2dup forth-wordlist search-wordlist
   dup 0< IF ( not immediate )
             drop compile,
             2drop true EXIT THEN
          IF ( immediate )
             nip nip execute
             true EXIT THEN
   false ;
```

handlers properties

- modular **extensible** (1. dimension)
  - interpreter (extensible) ⟶
  - compiler (extensible) ⟶
  - postponer (extensible) ⟶
- more **extensions** (2. dimension) ↓
  - target compiler
  - remote compiler
  - DSL compiler

handlers code
interpret character literals

```
\ interpret character literals

: charlit ( c-addr u1 -- i*x true | c-addr2 u2 false )
   dup 3 = IF over c@ [char] ' = 2 pick c@ [char] ' = and
   IF  drop char+ c@ true EXIT THEN THEN false ;
' charlit
```

handlers properties

- handlers are **simply** colon definitions

- composing handlers give new handlers
- handler lists
  - layed out in memory with **`create`** and **`,`**
  - **`n@ n!`** operate on cell counted lists
  - handler lists can be in **`allocate`**d memory
  - handler chained in :-definitions

handlers code
compile character literals

```
\ compile character literals

[: ( c-addr u1 -- i*x true | c-addr2 u2 false )
   charlit IF postpone literal true EXIT THEN  false ;]
```

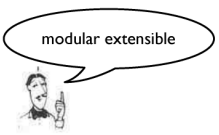
handlers design options

- possible stack effects
- haeh?
- token scanning
- search order

- prototypes for each options on git branches

possible handlers

- words
- base numbers (single cell)
- base prefix numbers (hex decimal bin)
- character literals
- string literals
- s"
- double precision numbers
- floating point numbers
- namespace scoped identifiers
- object systems
- date&time
- ...



handlers design options
possible stack effect

- what stack effect shall a handler have?
  - (c-addr u1 -- i*x true | c-addr2 u2 false)
  - (c-addr u -- i*x true | false)
  - (c-addr u -- i*x c-addr u true | c-addr u false)

handlers design options haeh?

- if no handler can cope with the token, what should be done?
  - signal error (`-13 throw`)
  - ignore

may the **`swap`** be with you!

discussion

handlers design options token scanning

- shall handlers work on pre scanned tokens?
- of shall they inspect the input stream on their own?

handle code

```
: handle ( c-addr1 u1 addr u -- i*x true | c-addr2 u2 false )
  cells bounds
  ?DO ( c-addr1 u1 )
      I @ execute ?dup IF ( i*x ) UNLOOP EXIT THEN
  cell +LOOP
  false ;
```

handler design options search order

- shall a handler search the search order
- or look into a single word list?
  - the search order will be a sublist of handlers

summary

- **simple**
  - handlers are ordinary :-definitions
  - handler lists are easy to build and manage
- **extensible**
  in 2 dimensions:
  1. extending handler lists with new handlers
  2. different compilers/interpreters (postponers)