Dynamically Structured Codes

M.L.Gassanenko

St.Petersburg Institute for Informatics and Automation of the Russian Academy of Sciences
14 liniya 39 SPIIRAN, St.Petersburg, 199178, Russia
mlg@forth.org

Abstract

The characteristic property of dynamically structured code is that relations between elements of executable code are established at run-time. The word "structured" underlines that these relations must be in some kind regular. What these relations must and/or may be is a subject for studies. We expect that the use of dynamically structured codes will allow to reduce the complexity of software and thus improve its safety and reduce its cost.

The paper discusses dynamically structured codes in general and proposes three new techniques. Two of them are improvements of the method of data execution (data-driven approach): *execution of data with a customized method of execution*, and *joint compilation of two data sets into a single data set* (and execution of the joint data set). The third technique — *dynamic construction of call hierarchy (DCCH1)* — may be used when we need to process a set of records having a complex format (examples: loading an object module, disassembly, instruction set emulation). DCCH1 is a regular way to build for each such record a customized code that will process that record.

1. Introduction

Our studies are a search of expressive means that would allow to reduce the complexity of software (and thus improve its safety and/or reduce the cost of creation and maintenance). This paper presents the results achieved in the direction of the use of dynamically structured code — *such code that relations between its elements are established at run-time.*

Although such techniques as self-modifying and dynamically generated codes are not popular, there are reports about their successful application [CUL89] ("*dynamic translation* of byte-coded methods into machine code transparently on demand at run-time"), [Koo90] ("self-modifying program execution with compiler-guaranteed safety", "The processor ... is *directly executing* the data structure") and harnessing them in the current research [KaK98U] ("the main problem ... with conventional languages like Pascal and C is impossibility to add or modify object code at run-time, ... to create a net at run-time").

In our studies we have taken as the starting point the method of data execution, the most structured technique among those using dynamic codes.

In this paper we propose three new techniques. Two of them are improvements of the data execution method: dynamic recompilation of multiple data sets into a single data set which then gets executed, and the use of non-traditional methods of execution (e.g. backtracking) to process data. The third technique (dynamic construction of call hierarchy, DCCH1) is a method, kindred to data execution and dynamic code generation, that works in the case that data are in non-executable form by the statement of the problem.

These results may be used in practical programming and serve as a basis for future investigations.

1.1 Self-modifying and dynamically generated codes: shortcomings and perspectives

First of all, we have to agree that the name "self-modifying and dynamically generated codes" is not descriptive. This name misses something important. Indeed, the name "constant codes" is equally non-descriptive: constant codes include both structured and object-oriented programming.

The idea of self-modifying and dynamically generated codes is known since the first von Neumann computers have appeared. These techniques were considered as a potentially powerful means, but somehow it happened so that they become considered as a bad style. There have been several reasons for this:

1. Self-modifying codes were machine-dependent.
2. Self-modifying programs are not reenterable.
3. Self-modifying codes were too often used to compensate insufficiencies in the instruction set.
4. When such discipline as structured programming began to appear, no analogous discipline for self-modifying codes was invented.
5. High-level languages did not support them.

A non-structured low-level method cannot compete with structured methods. The self-modifying codes were doomed in the world of mainstream programming.

Nevertheless, the Lisp community does use dynamically generated codes (namely, data execution). These methods survived because the Lisp code, even dynamically generated, is structured, and the motivation for their use was more serious than the one mentioned in the item 3 above.

The advantage of self-modifying codes is that their use allows to get rid of auxiliary entities (in the most trivial case of tricky assembly programming, it is a memory cell; in the case of data execution it is the loop and case statements that should otherwise control processing of data—that is, almost the whole program). This, in particular, lets self-modifying code execute faster than analogous constant code.

The arguments raised against self-modifying codes 30-40 years ago are not convincing now. These are evident arguments against several particular uses of self-modifying codes, but not against self-modifying codes in general. It is evident that it is wrong to use self-modifying codes at the level of auxiliary constructs (constructs that do not correspond to the concepts of the problem). It is evident that techniques based of self-modifying codes must be structured, and it is just not true that they cannot be structured.

The name *"Dynamically Structured Codes"* is itself an important result. It briefly describes the line of investigation, as well as the expected results. The distinctive feature of dynamically structured code is that *relations between its elements are established at run-time.* Impossibility of beforehand compilation (that is, of traditional compilation at compile-time) is also an indication of dynamically structured code.

We class data execution with dynamically structured codes because data (whether or not represented in an executable form) in general case are created and modified at run-time.

As yet, we cannot give general dynamic code guidelines. Nevertheless, we can point out two important principles occurring in the use of dynamic codes: *data execution* and *generation of customized code*. Yet another such principle — *separation in time* — does not necessarily imply the use of dynamic codes.

1.2 Data execution

The technique of *data execution* was independently invented at both sides of the 'Iron Curtain'. In the West it is known as 'data-driven approach', the Russian names are 'the functional methods of programming' [Tuz88R] (this name was not considered felicitous because of possible confusion with the functional programming) and 'process-oriented programming'.

Data execution is enough powerful to be used for AI [Rod90]. Tuzov tried to find a problem which could not be solved in the data execution style, but has not found one. The author of this paper has used data execution in compiler construction and in an experimental program for processing sentences in the Russian language.

Data execution may be applied where there is a set of inter-related data. In the simplest case, the data items are just ordered. In a more complex case, the data and relations between them may form a graph. The data may be of different types. The data in the set may form structures and substructures.

The data set is transformed into a superposition of functions (in the simplest case, a sequence of calls), and execution of this representation of data solves the problem. That is, the problem is solved via execution of the functional representation of data (this is why this technique was called 'the functional methods').

Let us provide an example (we assume a system of open interpreter class 1 or 2 [Gas98a], a FIG-Forth-like model with threaded code located in the data memory belongs to class 1):

```
\ data types
DEFER LS   \ list
DEFER NUM  \ number
DEFER STR  \ string

\ save/restore the current context
: ` ' >BODY STATE @ IF [COMPILE] LITERAL THEN ; IMMEDIATE
: save-ctx ( -- ) ( R: -- ctx-info )
   R>   ` LS @ >R  ` NUM @ >R  ` STR @ >R   >R ;
: rest-ctx ( -- ) ( R: ctx-info -- )
   R>   R> R> R>  ` LS !  ` NUM !  ` STR !   >R ;

\ Two lists
: list1   1 NUM   2 NUM   C" abc" STR   3 NUM   ;
: list2  10 NUM  20 NUM   ['] list1 LS  C" string" STR
         30 NUM  40 NUM   C" end" STR ;

\ Print a list
: .str [CHAR] " EMIT COUNT TYPE [CHAR] " EMIT SPACE ;
: .ls  ." ( " EXECUTE ." ) " ;
: print ( addr -- )
   save-ctx
   ['] .    IS NUM
   ['] .str IS STR
   ['] .ls  IS LS
           ." ( " EXECUTE ." ) "
   rest-ctx
;

\ Count numbers in a list (shallow, deep)
: DROP_1+ DROP 1+ ;
: count-numbers-shallow ( addr -- n )
   save-ctx
   ['] DROP    IS STR
   ['] DROP_1+ IS NUM
   ['] DROP    IS LS
          0 SWAP EXECUTE
   rest-ctx
;
: count-numbers-deep ( addr -- n )
   save-ctx
   ['] DROP       IS STR
   ['] DROP_1+    IS NUM
   ['] EXECUTE    IS LS
          0 SWAP EXECUTE
   rest-ctx
;
```

A sample session:

```
' list1 print ( 1 2 "abc" 3 )  ok
' list2 print ( 10 20 ( 1 2 "abc" 3 ) "string" 30 40 "end" )  ok
' list1 count-numbers-shallow . 3  ok
' list1 count-numbers-deep . 3  ok
' list2 count-numbers-shallow . 4  ok
' list2 count-numbers-deep . 7  ok
```

Tuzov [Tuz88R] considered a list as a function with 3 parameters (**NUM**, **STR**, **LS** in this example). I believe that he wanted to write something like the following (in Scheme):

```
(define list1
 (lambda (num str ls)
  (begin
    (num 1)    (num 2)    (str "abc")    (num 3)
)))

(define list2
 (lambda (num str ls)
  (begin
    (num 10)    (num 20)    (ls list1)    (str "string")
    (num 30)    (num 40)    (str "end")
)))

(define write-lst
 (lambda (lst)
     (define _write
      (lambda (x)
        (write x)
        (display " ")
     ))
   (display "( ")
   (lst _write _write write-lst)
   (display ") ")
))

(define count-num-shallow
 (lambda (lst)
     (define n 0)
     (define _skip
      (lambda (x)
       ()
     ))
     (define _count
      (lambda (x)
        (set! n (+ n 1))
     ))
   (lst _count _skip _skip)
   n
))

(define count-num-deep
 (lambda (lst)
     (define n 0)
     (define _skip
      (lambda (x)
       ()
     ))
     (define _count-num
      (lambda (x)
        (set! n (+ n 1))
     ))
     (define _count-ls
      (lambda (x)
        (set! n (+ n (count-num-deep x)))
     ))
   (lst _count-num _skip _count-ls)
   n
))
```

A sample session:

```
STk> (write-lst list1)
( 1 2 "abc" 3 ) #[undefined]
STk> (write-lst list2)
( 10 20 ( 1 2 "abc" 3 ) "string" 30 40 "end" ) #[undefined]
STk> (count-num-shallow list1)
```

```
3
STk> (count-num-deep list1)
3
STk> (count-num-shallow list2)
4
STk> (count-num-deep list2)
7
```

The Forth and Scheme versions are different two aspects: at first, Forth lets us control data execution by means of return address manipulations. At second, Forth manipulates with call contexts explicitly while in Scheme creation of a context is a part of the function call. Undoubtedly, contexts as an expressive means deserve a specific study [Gas98b].

Let us demonstrate dynamic code generation (Tuzov used a garbage-collected heap for headerless dynamic definitions):

```
: CLITERAL ['] (C") HERE ROT COUNT [COMPILE] SLITERAL ! ; IMMEDIATE

: copyNum [COMPILE] LITERAL COMPILE NUM ;
: copyStr [COMPILE] CLITERAL COMPILE STR ;
: copyLs  [COMPILE] LITERAL COMPILE LS ; \ shallow copy

: concat
    save-ctx
    ['] copyStr    IS STR
    ['] copyNum    IS NUM
    ['] copyLs     IS LS
           >R >R :NONAME R> EXECUTE R> EXECUTE [COMPILE] ;
    rest-ctx
;
```

A sample session:

```
' list1 ' list2 concat alias list3  ok
' list3 print ( 1 2 "abc" 3 10 20 ( 1 2 "abc" 3 ) "string" 30 40 "end" )  ok
```

Analogous example in Scheme:

```
(define concat-lst
 (lambda (lst1 lst2)
     (define newlist '(begin))
     (define copyNum
      (lambda (x)
       (set! newlist (append newlist (list (list 'num x))))
     ))
     (define copyStr
      (lambda (x)
       (set! newlist (append newlist (list (list 'str x))))
     ))
     (define copyLs
      (lambda (x)
       (set! newlist (append newlist (list (list 'ls x))))
     ))
     (lst1 copyNum copyStr copyLs)
     (lst2 copyNum copyStr copyLs)
     (set! newlist
           (append '(lambda (num str ls)) (list newlist))
     )
     (eval newlist)    ;;; transform to excecutable form
))
```

A sample session:

```
STk> (write-lst (concat-lst list1 list2))
( 1 2 "abc" 3 10 20 ( 1 2 "abc" 3 ) "string" 30 40 "end" ) #[undefined]
```

There are two reasons why we do not define **concat** as

```
: concat ( list1 list2 -- list )
  2>r :noname 2r> swap compile, compile, postpone ; ;
```

1) If **concat** is defined as above, we cannot predict the contents of the return stack while lists execute, and lose the ability to control execution of code by means of return address manipulations.
2) In the general case, a list has two more parameters — **list-begin** and **list-end** (**DEFER** variables) and looks as
```
: list1a list-begin 1 NUM 2 NUM C" abc" STR 3 NUM list-end ;
```

An important feature of Tuzov's method is *the use of the existing code interpreter to execute data.*

One more important concept is *data compilation.* Transformation of data into the universal executable form is called data compilation. In Scheme, we can construct a new data set either in the text form or as a list. Transformation to the executable format requires calling the unstandard function EVAL. In Forth, we can directly generate the executable form.

Data execution is similar to object-oriented programming (OOP) in that both bind functions to data and both support polymorphism, allowing to process data of different types in a uniform way. The difference is that with OOP we concentrate on properties of a single data unit; with data execution, we concentrate on processing the data set according to the relations between data elements.

Let us outline the variations of the data execution approach:
1) one can either interpret the external representation of data or execute compiled data;
2) the data interpreter may be written in the implementation language, or the interpreter of the implementation language may be used to process data;
3) the association of data with the corresponding actions may be implemented by the interpreter or be built into the data representation;
4) polymorphism may be present or absent, that is, execution of data may support either a single operation or multiple operations;
5) in Forth, return stack manipulations may be used to control the process of code (that is, data) execution. In the sections 2 of this paper *we propose the following improvements:*
    1) we may choose the most adequate method of code execution, that is, it may be the traditional sequential execution, or execution with backtracking, or some other custom method;
    2) when processing requires access to multiple data sets, they may be either executed in parallel or compiled into a single data set;
    3) (DCCH1) if data are not executable by the definition of the problem, we can use the structure of data to construct custom code for processing the data.

1.3 Implementation of untraditional methods of code execution via the return stack access

If we implement a new method of code execution as a program in the implementation language, we should expect at least a 10 times decrease of execution speed (by 1000%). In the works [Gas92R], [Gas94], [Gas96], [Gas96R], [Gas97R] the following method is proposed: we extend the existing interpreter with new operations. If these new operations execute require, say, 10 implementation language primitives each, and are executed in 10% cases, the decrease of execution speed is 10*10%=100%, that is, execution speed falls only by 2 times. In practice, in case of Forth we can extend the interpreter using Assembler, reducing the loss of execution speed to 10-20%, which is negligible. An extended explanation of the nature of return address manipulations may be found in [Gas99].

It may be recommended to consider IP and the return stack as a single *interpretation stack* (IP is the top, the return stack contains the other elements). The following rule explains how to write a procedure that makes desired changes on the interpretation stack: write code that does with the return stack what must be done with the interpretation stack; put this code into an auxiliary procedure. This procedure will do the required changes with the interpretation stack. The rule is correct because when a procedure is called, IP is pushed onto the return stack; **EXIT** loads IP from the return stack.

Let us give an example of backtracking in Forth [Gas92R], [Gas94]:

```
: ENTER >R ; \ ( tcf-addr -- ) call the threaded code fragment at tcf-addr
: SUCC COMPILE R@ COMPILE ENTER ; IMMEDIATE
: FAIL COMPILE R> COMPILE DROP COMPILE EXIT ; IMMEDIATE
: 1-10 ( --> i --- i --> ) \ generate numbers from 1 to 10
    0 BEGIN     1+ DUP 11 <
      WHILE     SUCC \ call the continuation, of type ( i -- i )
      REPEAT
      DROP
      FAIL ; \ exit the code fragment that contains the continuation
: //2 ( i --> i --- i --> i ) \ filter even numbers
      DUP 2 MOD 0=
      IF    SUCC \ call the continuation, of type ( i -- i );
                 \ (in the case of //2 we could just exit)
      THEN
      FAIL ; \ exit the code fragment that contains the continuation
: .even1-10 ( -- ) 1-10 //2 DUP . ;
```

A sample session:

```
.even1-10 2 4 6 8 10  ok
```

In this code we use the principle "success as a call of continuation". A procedure that performs success (the word **SUCC**) calls the residue of the threaded code fragment to which it is compiled. When a procedure fails, it forces an exit from the threaded code fragment to which it is compiled (that is, **EXIT** is a word that fails). You can see that the word **1-10** calls continuation multiple times, from within a loop. The word **//2** calls continuation only if the number at the stack top is even. The residue of the calling code fragment is an argument to a Forth word, just like numbers on the stack. The Forth word is allowed to pass control to this code fragment, to call it multiple times, to do nothing with it, or to do something else.

We have also to note that some words in the section 2.3 fetch data from threaded code to. When a word is assigned to a number of **DEFER** variable, it may know which namely variable called it by executing the phrase **RR@ CELL- @** . **RR@** takes a copy of the return address from the return stack and converts it to the representation used for data addresses, the use of two representations is a shortcoming of Win32For. The whole phrase returns the *execution token* (that is, the code field address) of the **DEFER** variable compiled in the code immediately before that address. This is the variable that called the word.

1.4 Used software

The Forth programs presented in this paper run on Win32Forth ver. 3.5 by Tom Zimmer, available at ftp.taygeta.com or via www.forth.org. According to the [Gas98a] classification, Win32Forth has an open interpreter of class 2 because return addresses are single-cell, threaded code is located in the data memory, but representation of return addresses (absolute) is different from that for data addresses (base-relative). There is no problem with the latter except for the code in the section 2.3 because the programs presented in these sections do not access threaded code. In the section 2.3, we define words **>RR** and **RR@** that both access the return stack and do necessary conversion.

The Scheme programs run on the STk interpreter version 3.1 by Erick Gallesio, available at http://kaolin.unice.fr/STk .

2 Proposed Techniques

2.1 DCCH1: Dynamic Construction of Call Hierarchy

"DCCH1" stands for "Dynamic Construction of Call Hierarchy, version 1". The digit 1 is appended because modifications of this approach are expected to appear in the future. The method is based on intense use of **DEFER** variables, variables that execute functions assigned to them.

The typical problem for which DCCH1 may be used is like following: we have to process a set of data records; the data records have a complex format, namely, include multiple optional variable-length fields, and the type and size of a field become known only as a result of analyzing the record. The fact that there

are multiple variable-length fields makes us access the fields sequentially. Each record may be considered as a command requiring the program to perform some actions.

Such complex record formats may be found in the area of systems programming. The 16-bit MS-DOS object module format and the Intel instruction set may serve as examples.

The idea of DCCH1 may be briefly explained as following. At first, we decide to separate processing into two phases: *analysis* and *treatment* (processing itself). At the phase of processing, we generate code that will work at the treatment phase. Then, we mention that we do not really need dynamic code generation because the use of **DEFER** variables would be enough.

The method of DCCH1 implies that:
1) processing of a record is separated into two phases: *analysis* and *treatment* (processing itself)
2) interaction between these two phases is done via assignments to **DEFER** variables or, only if the use of a **DEFER** is not possible (e.g. we need to pass a number), via traditional data structures.

The second item means that instead of

```
0 VALUE x
...
<subfield-type> TO x
...
CASE x
  0 OF <action1> ENDOF
  1 OF <action2> ENDOF
  2 OF <action3> ENDOF
ENDCASE
...
```

we write

```
DEFER action
...
CASE <subfield-type>
  0 OF ['] <action1> TO action ENDOF
  1 OF ['] <action2> TO action ENDOF
  2 OF ['] <action3> TO action ENDOF
ENDCASE
...
action
...
```

This requirement makes us write code in a better style.

Let us consider an example. A record has the following format:

```
header fieldA fieldB
```

where

- `fieldA` is of type either `A1` or `A2`
- `fieldB` is of type either `B1` or `B2`

The treatment is done according to the following scheme:

```
prologue(A) action(B) epilogue(A)
```

where

- `prologue(A)` and `epilogue(A)` are chosen according to the type of `fieldA` among `prologueA1`, `epilogueA1` and `prologueA2`, `epilogueA2`;

- `action(B)` is chosen according to the type of `fieldB` among `actionB1` and `actionB2`.

To perform `prologueA1` and `epilogueA1` we need *dataA1*, and to perform `prologueA2` and `epilogueA2` we need *dataA2* (these two sets of variables may have a common part). Analogously, to perform `actionB1` and `actionB2` we need *dataB1* and *dataB2* correspondingly. Procedures `getdataA1`, `getdataA2`, `getdataB1`, `getdataB2` fetch these data from the fields of corresponding types.

The code written according to the DCCH1 technique looks like following:

```
\ === interface between the analysis and treatment parts
DEFER getdataA DEFER getdataB
DEFER prologue DEFER action DEFER epilogue
<declaration of data variables>

\ === access to fields
QUAN ^pos \ the component pointer points to the current field of the record
<definitions of procedures getdataA1, getdataA2, getdataB1, etc.>
\ these procedures fetch data using ^pos and then advance ^pos past the data

\ === analysing code
: examine
    <is fieldA of type A1 ?>
    IF    ['] getdataA1 IS getdataA
          ['] prologueA1 IS prologue
          ['] epilogueA1 IS epilogue
    ELSE  ['] getdataA2 IS getdataA
          ['] prologueA2 IS prologue
          ['] epilogueA2 IS epilogue
    THEN
    <is fieldB of type B1 ?>
    IF    ['] getdataB1 IS getdataB
          ['] actionB1 IS action
    ELSE  ['] getdataB2 IS getdataB
          ['] actionB2 IS action
    THEN
    <advance the component pointer past the header>
;
: analysis examine getdataA getdataB ;

\ === treatment code
<definitions of procedures prologueA1, prologueA2, actionB1, etc.>
: treatment prologue action epilogue ;

\ === the main definition
: processing  analysis treatment ;
```

For comparison, we present a "traditional style" version:

```
\ --- the "traditional" approach ---
QUAN ^pos \ the component pointer points to the current field of the record
<declaration of data variables>
<definitions of procedures using-^pos-do-prologueA1, using-^pos-do-actionB1,
 save-epilogue-dataA1, using-saved-data-do-epilogueA1, etc.>

0 VALUE wasA1
<declaration of a data structure to held information about the type of fieldB>
: process-record
```

```
    <save information about the type of fieldB>
    <is fieldA of type A1 ?>
    IF      <advance the component pointer>
            using-^pos-do-prologueA1
            save-epilogue-dataA1
            <advance the component pointer>
            TRUE TO wasA1
    ELSE    <advance the component pointer>
            using-^pos-do-prologueA1
            save-epilogue-dataA1
            <advance the component pointer>
            FALSE TO wasA1
    THEN
    <use saved information, was fieldB of type B1 ?>
    IF      using-^pos-do-actionB1
            <advance the component pointer>
    ELSE    using-^pos-do-actionB2
            <advance the component pointer>
    THEN
    wasA1
    IF      using-saved-data-do-epilogueA1
    ELSE    using-saved-data-do-epilogueA2
    THEN
;
```

A number of design errors may be noticed in this code. For example, three different sorts of code (for header examination, field access and data treatment) are intermixed. The variables are used in an irregular way. In general, there is only one main design error: in this program, we failed to develop an adequate view on the problem and therefore failed to develop an adequate factoring. Now that we know DCCH1, we can rewrite the code without using **DEFER** variables. The advantage of DCCH1 is that it introduces a discipline for dependencies within a program and thus makes us use a good style.

Let us give an example of a real life problem: decoding the memory operand in an i80x86 instruction (32-bit addressing mode). The memory operand has the following format:

```
ModR/M [SIB] [Displacement]
```

Where

`ModR/M` is a byte containing three fields:
- `mod` (2 bits, addressing mode: displacement of 0/8/32 bits or data in a register),
- `reg` (3 bits, register number, the first operand),
- `R/M` (3 bits, register number, either the second operand itself or the address base register)

`SIB` is an optional byte containing three fields:
- `scale` (2 bits, multiplication of the index by 1/2/4/8),
- `index` (3 bits, register number, the index register),
- `base` (3 bits, register number, the base register)

`displacement` is an optional field containing a 32-bit or 8-bit (sign-extended) number.

The general rule has the following exceptions:
`mod:R/M=00:101` indicates a 32-bit displacement with no base register;
`R/M=100` indicates presence of the `SIB` byte when `mod` specifies a memory operand;
`mod:base=00:101` indicates a 32-bit displacement with no base register;
`index=100` specifies no index.

We do not consider 16-bit addressing modes and commands containing opcode in the `reg` field. The full specification of the i80x86 instruction encoding may be found in [Int97]. The program presented below runs under Win32For 3.5. Please, remember that it is usually more convenient to read Forth programs starting from the end.

```
\ ======================================================================
\ Analyser outputs (values and functions):
0 value r1          0 value r2          0 value disp
0 value rbase       0 value rindex      0 value scale
defer ?work-r1      defer ?work-r2      defer ?work-disp
defer ?work-rbase   defer ?work-rindex

\ Auxiliary values and functions:
0 value _mod
defer ?exam-mem     defer ?exam-SIB     defer ?exam-disp

\ ======================================================================
\ Sample treatment -- printing all, assuming 32-bit registers.
\ If XXX-pres and XXX-abs are DEFER words, multiple kinds
\ of treatment are possible.
: .reg ( n--) 2* S" AXCXDXBXSPBPSIDI" drop + 2 type ;
: .scale ( n--) ?dup if 1- 2* S" *2*4*8" drop + 2 type then ;
  \ '-pres' stands for "present"
: r1-pres     ." E" r1 .reg ." ," ;
: r2-pres     ." E" r2 .reg ;
: disp-pres   disp . ;
: rbase-pres  ." [E" rbase .reg ." ]" ;
: rindex-pres ." [E" rindex .reg scale .scale ." ]" ;
  \ '-abs' stands for "absent"
: r1-abs ;          : r2-abs ;          : disp-abs ;
: rbase-abs ;       : rindex-abs ;

: work-all ?work-r1 ?work-r2 ?work-disp ?work-rbase ?work-rindex CR ;
\ ======================================================================
\ Access to the current field (that is, the current byte).
0 value ^pos
: ^@ ( --x) ^pos c@ ;     : ^++ ( --) ^pos 1+ to ^pos ;
: ^32@ ( --x) ^pos @ ;    : ^32++ ( --) ^pos 1+ to ^pos ;
: &^ ( mask--val)       \ val comes from bits specified by mask
    ^@
        begin   over 1 and 0=
        while   1 rshift swap 1 rshift swap
        repeat
    and
;
: c>s ( signed-char -- n ) DUP 128 AND 0<> -128 AND OR ;
\ ======================================================================
\ The analysis itself ('exam' stands for "examine").
\ Read this section starting from the last definition.
: exam-disp8    ['] disp-pres is ?work-disp   ^@ c>s to disp   ^++ ;
: exam-disp32   ['] disp-pres is ?work-disp   ^32@ to disp   ^32++ ;
: exam-SIB
        $C0 &^ to scale
        $38 &^ to rindex
        $07 &^ to rbase
    rindex 4 <> if ['] rindex-pres is ?work-rindex then
    rbase 5 =  _mod 0=  and
    if  ['] exam-disp32 is ?exam-disp   \ disp32 instead
        ['] rbase-abs is ?work-rbase    \ of [EBP]
    then
    ^++
;
: exam-modR/M
        $C0 &^ to _mod
        $38 &^ to r1    \ either reg or a part of opcode
        $07 &^ to r2    \ either 2nd reg or base reg
    r2 to rbase
    case _mod
    3 of    ['] r2-pres is ?work-r2
```

```
            ['] noop is ?exam-mem
            ['] rbase-abs is ?work-rbase
      endof
    2 of    ['] exam-disp32 is ?exam-disp endof
    1 of    ['] exam-disp8 is ?exam-disp endof
    0 of    r2 5 =
            if  ['] exam-disp32 is ?exam-disp   \ disp32 instead
                ['] rbase-abs is ?work-rbase    \ of [EBP]
            then \ code for mod=0 R/M=4 base=5 is in exam-SIB
      endof
    endcase
    rbase 4 =
    if  ['] exam-SIB is ?exam-SIB
    then
    ^++
;
: exam-mem32     ?exam-SIB ?exam-disp ;
: exam-reg/mem32 exam-modR/M ?exam-mem ;
: init
    ['] noop       is ?exam-SIB
    ['] noop       is ?exam-disp
    ['] exam-mem32 is ?exam-mem     \ !!! default = do examination
    ['] r1-abs      is ?work-r1
    ['] r2-abs      is ?work-r2
    ['] disp-abs    is ?work-disp
    ['] rbase-pres  is ?work-rbase  \ !!! default = present
    ['] rindex-abs  is ?work-rindex
;
\ ================================================================
: ttt ( addr -- )
    1+ to ^pos                  \ assuming 1-byte opcode
    init
    ['] r1-pres is ?work-r1     \ assuming reg, not opcode
    exam-reg/mem32 work-all
;
\ ================================================================
```

A sample session:

```
see 2@
2@ IS CODE
          1238 8B443B04         mov     eax , 4 [ebx] [edi]
          123C 8B1C3B           mov     ebx , [ebx] [edi]
          123F 50               push    eax
         ok
cr $1238 ttt $123C ttt
EAX,4 [EBX][EDI]
EBX,[EBX][EDI]
 ok
```

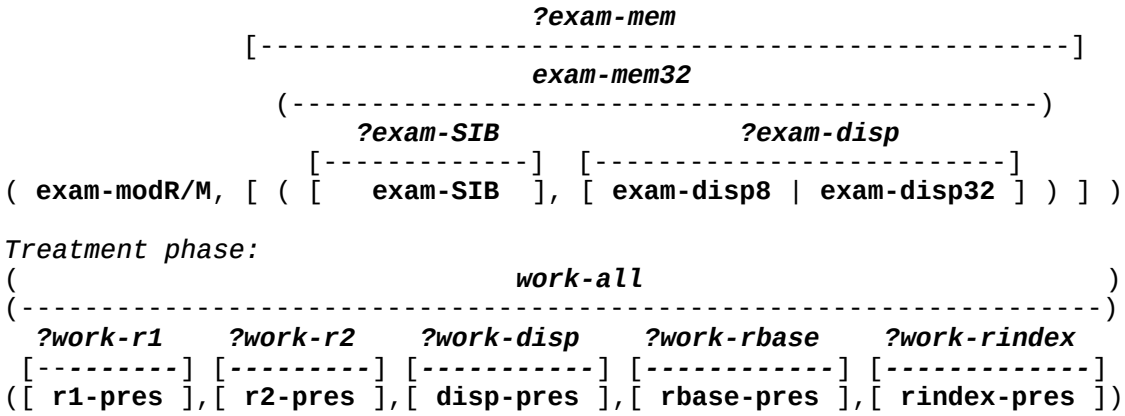
Notes on programming. Practice shows that in case of DCCH1 the traditional Forth debugging (re-reading code, trying it and examining variables after execution, including **SEE**ing) is more helpful than tracing. Probably, this derives from the fact that the Win32Forth debugger does not display the state of **DEFER** variables (although it allows interpretation of Forth text input by the user, in particular, **SEE**ing). Programming with DCCH1 is a bit different from other kinds of programming, but there is no need in special training. The programmer just has to mind that not all methods that work well with traditional style of programming may be applicable for DCCH1. If some debugging technique does not give results, just stop using it and think. Maybe, you have to modify it or try another technique.

Call hierarchy diagram: an approach to documenting. The hierarchy of calls of the above program may be shown as the following diagram (**(a,b)** denote sequential execution, "first **a** then **b**"; **[a|b]** denote "**a** or **b** or nothing"; **forth-names** denote "activities" while ***forth-names-in-italics*** denote words whose purpose is to call or not to call these "activities", the group of words controlled by such names is shown by dashes below them):

Examination phase:

exam-reg/mem32

(----------)



It may be seen that the logics of these two phases do not match each other exactly, although there is no complete mismatch. With the traditional approach, the more analysis and treatment match, the easier is programming. With DCCH1, such match or mismatch has no influence on programming.

DCCH1, data execution and dynamic code generation. Data execution lets the structure of data control the processing of data. In the case of DCCH1, we initially have a non-executable data record. We analyse that record get information about the structure of data. Knowing this structure, we build a customized program that can process the record (this, and, probably, only this record). Then, we execute that custom-built program. In principle, we could use dynamic code generation, but the practice shows that the use of just **DEFER** variables is enough. DCCH1 also resembles syntax-directed translation.

DCCH1 guidelines. In DCCH1, we factor code by separating processing into phases. We introduce at least two phases: the phase of analysis and the phase of treatment. We consider analysis code and treatment code as two sorts of code which must never be intermixed. (In addition to these two, we may recognize other qualitatively different sorts of code, which should also be as much separate as possible). Then, we design an interface between these sorts of code. The general rule is that **DEFER** variables should be used where possible; data that would be used as auxiliary values to control execution of IF and CASE statements must not be passed between different sorts of code. That is, at the phase of analysis we make decisions but we do not perform them; instead, we perform **DEFER** variable assignments.

Let us now turn to the treatment code. We construct the treatment code as an executable "frame" with "holes", the structure of this frame describes treatment in the general case; each particular case may be obtained from the general schema by filling the "holes" with procedures. We implement such "executable frames" as procedures that invoke **DEFER** variables ("holes"). There may be several layers of such "frames", that is, "holes" may be filled with "frames".

So, at the phase of analysis of a data record we construct customized code that will process the record (that is, fill the "holes" in the "frame"). Then, at the phase of treatment, we execute this customized code.

Additional notes. One can notice that in the disassembly example we build *two* customized programs: the first one controls fetching data from the instruction subfields, the second one is treatment code. It is interesting to note that the first customized program begins execution *before* its construction is complete; it executes and builds itself.

An advantage of DCCH1 is that it may be used in Algol-derivative languages, because these languages allow execution of functions assigned to variables.

Assessment of complexity. The numerical metric showing that DCCH1 programs are less complex is the number of sorts of dependency between the analysis code and the treatment code.

In the case of the traditional approach, there are three sorts of dependency between the analysis code (A) and the treatment code (T):

1) A↔T dependency due to sharing the same data structures (first of all, the component pointer);
2) A↔T dependency due to sharing the same flow of control;
3) A→T the analysing code directly controls execution of the treatment code.

In the case of DCCH1, there is only one sort of dependency:

1) A→T the analysing code indirectly (via **DEFER** variables) controls execution of the treatment code.

DCCH1 prevents us from introducing unnecessary dependencies and thus makes our code simpler.

2.2 Data execution with customized data interpreter

The method of data execution becomes more powerful if we allow the use of untraditional methods of code execution. An example of a combination of data execution and backtracking is given below. The following program prints all subsets of the set {*first, second, third*}:

```
: ENTER >R ;
: el  R@ ENTER DROP ;
: .{} CR ." { " DEPTH 0 ?DO I PICK COUNT TYPE SPACE LOOP ." } " ;
: subsets C" first" el C" second" el C" third" el .{} ;
```

This is what this program prints:

```
subsets
{ third second first }
{ second first }
{ third first }
{ first }
{ third second }
{ second }
{ third }
{ }  ok
```

How this works. The word **.{}** prints the strings whose addresses are left on the stack. The word **ENTER** calls a threaded code fragment whose address is at the data stack top. The word **R@** lets a procedure obtain the address of the residue of the calling procedure's code. With the phrase **R@ ENTER** the procedure calls this residue of the calling procedure's code. The word **el** receives control when a counted string address in on the stack. This word first executes the residue of the calling word with this address on the stack, then **DROP**s this address and lets the residue of the calling word execute again, this time without that address on the stack. So, when the word **subsets** executes, each word **el** executes the residue of **subsets**'s code twice: with the address of corresponding string on the stack, and without it. If there are *N* elements in the set, the word **.{}** (which is in the residue of the last **el**) executes 2^*N* times, printing all possible combinations.

The reader may note that this code assumes that the stack is initially empty; this is not important now. If we used the "counted array" format (*xn ... x1 n*) for subsets, the word **el** would be

```
: el SWAP 1+ R@ ENTER 1- NIP ;
```

and the definition

```
: _begin-set 0 R> ENTER DROP ;
```

had to be executed before execution of elements. It is possible to improve this code if it is necessary.

In this example we use backtracking as a method of data execution. In general, when we have to process data, we can use among various methods of code execution. It may be: the traditional sequential code execution with procedure calls; backtracking; parallel execution; two-level backtracking [Gas96R], [Gas97R]. Finally, we may design a custom method of execution.

Assessment of complexity. Programs that use data execution are shorter and simpler because of the reuse of the code interpreter for processing data. The method of choosing the most adequate method of execution extends the area of applicability of data execution.

It is interesting to compare the number of words in programs that solve the same problem (printing subsets of a set) with and without backtracking and data execution. The table given below summarizes the results of such measurements. The phrases that declare data (like **`C" first" el`** above) are not counted. The considered programs are: `Tr` — two versions that use traditional control structures; `Ba` — uses backtracking (requires BacFORTH [Gas94]); `DE` — uses data execution; `DE&B` — uses both backtracking and data execution.

The number of words in equivalent programs that use different methods

	Tr	Tr(*)	Ba	DE(*)	DE&B	DE&B(+)
declaration of data	14	16	14	9	2	2
printing the set	14	12	14	12	14	14
generation of the sets	26	32	22	20	9	5
the latter includes:						
access to data structures	4	7	4	7	0	0
stack manipulations (**)	6	8	4	1	0	0
the logic (the rest)	16	17	14	12	9	5
total	54	60	50	41	25	21

(*) uses a different algorithm (incrementing arbitrary length numbers instead of the try-and-backtrack approach; does not "reuse" the stack as a storage for sets).
(**) in the cases of `Ba` and `DE&B`, including backtrackable stack operators; some stack manipulations are assigned to the logic group.
(+) considering that the phrase **`R@ ENTER`** is defined as **`SUCC`** in BacFORTH.

Additional note. An important feature of backtracking is the ability to create modules responsible for iteration and thus (1) reuse the code responsible for iteration and (2) improve maintainability of programs by avoiding duplication of such code; in the program `Ba` these advantages did not work at all because the problem was too simple.

2.3 Joint compilation vs. parallel execution

Data execution is a good principle, but let us consider the (theoretical) example of addition of two numbers. You can represent both sequences of digits in the executable form, but how you will execute them? Tuzov [Tuz88R] proposed to use parallel execution. The first process yields the first digit, the second process yields the second one, and the third (or, maybe, the second) process adds them.

We propose to use *data recompilation* in such cases. The first and the second data sets are compiled together into a *joint data set.* The types of items from the two data sets must be different in the general case because such elements may be required to behave differently. Execution of the joint data set will give the desired result.

If data in the two sets initially have the same type (e.g. both are numbers), we may change their type in the process of recompilation (that is, we will have numbers two types: numbers from the first list and numbers from the second list; the numbers from the first list will behave differently from the numbers from the second list).

Alternatively, we can use compilation of the third, "consumer" data set with the two "source" data sets. The consumer data set elements may be uninitialized, but they must have a different type. Two source data elements execute and leave their data on the stack. Then, the consumer element processes these data and stores the result into itself. The resulting data set may be extracted in an additional pass (consumer elements

append data to a new data set, source elements do nothing). Total, 3 passes are needed: re-compilation, processing, extraction of the result.

These 3 passes are not slower than the traditional approach. With the traditional approach, the computer spends much time controlling code that processes data (loops, conditional statements, etc.). In the case of data execution, data themselves control their processing.

Let us give two examples. Both programs compare two lists in the functional representations. The following code shows the method of data representation and four sample lists:

```
DEFER NUM   DEFER STR   DEFER LS   DEFER BEGL   DEFER ENDL
: list3 BEGL 1 NUM 2 NUM C" abc" STR ENDL ;
: list4 BEGL 10 NUM 20 NUM ['] list3 LS 100 NUM C" end" STR ENDL ;
: list5 BEGL 1 NUM 22 NUM C" abc" STR ENDL ;
: list6 BEGL 10 NUM 20 NUM ['] list5 LS 100 NUM C" end" STR ENDL ;

\ save/restore the current context
: ` ' >BODY STATE @ IF [COMPILE] LITERAL THEN ; IMMEDIATE
: save-ctx ( -- ) ( R: -- ctx-info )
  R>  ` LS @ >R  ` NUM @ >R  ` STR @ >R  ` BEGL @ >R  ` ENDL @ >R  >R ;
: rest-ctx ( -- ) ( R: ctx-info -- )
  R>  R> R> R> R> R>  ` LS !  ` NUM !  ` STR !  ` BEGL !  ` ENDL !  >R ;
```

The first example demonstrates parallel execution of two lists. The symbol **I:** labels interpretation stack diagrams, the interpretation stack consists of IP (the top) and the return stack (the rest).

```
\ tasks have been utterly simplified:
\ a task state is determined by a single return address
: SWAP-TASKS   R> R> SWAP >R >R ;    ( I: a1 a2 -- a2 a1 )
: RROT-TASKS   \ task switch happens when the calling def-n exits
   R>  R> R> R> -ROT >R >R >R  >R ;  ( R: a1 a2 a3 -- a2 a3 a1 )
\ switch tasks now; the same as  : ROT-TASKS R> R> R> -ROT >R >R >R ;
: ROT-TASKS RROT-TASKS ;             ( I: a1 a2 a3 -- a2 a3 a1 )

: str=  ( c-addr1 c-addr2 -- f ) COUNT ROT COUNT COMPARE 0= ;
: ls=   ( list1 list2 -- f )
    EXECUTE EXECUTE  \ let lists start as tasks
    BEGIN            (  )
        ROT-TASKS    ( elem func elem func ) \ tasks leave the data
        ROT OVER =   \ the same function?
        IF     EXECUTE      \ NB: exec-n of endl= forces an exit
        ELSE   2DROP DROP 0
        THEN
    0= UNTIL          \ exit the loop when mismatch is found
    2R> 2DROP         \ remove 2 other tasks
    0
;
: endl= ( x1 x2 -- true ) ( I: a1 a2 a3 -- )
    R> R> R> DROP DROP DROP   \ exit from ls= and the two lists
    2DROP TRUE                \ returning true
;
\ NB: task rotation happens when the definition is EXITed
: cmpnum    ['] =     RROT-TASKS ; ( x -- x xt )
: cmpstr    ['] str=  RROT-TASKS ; ( x -- x xt )
: cmpls     ['] ls=   RROT-TASKS ; ( x -- x xt )
: cmpendl 0 ['] endl= RROT-TASKS ; ( -- x xt ) \ doesn't receive x

: eqlst ( list1 list2 -- f )
    save-ctx
      ['] SWAP-TASKS IS BEGL
      ['] cmpnum     IS NUM
      ['] cmpstr     IS STR
      ['] cmpls      IS LS
      ['] cmpendl    IS ENDL
      ls=
    rest-ctx
```

```
;
```

A sample session:

```
' list4 ' list4 eqlst . -1  ok
' list4 ' list6 eqlst . 0  ok
```

We have to note that we cannot straightforwardly rewrite this program in Scheme because in Scheme there is no return stack access, and the program will have to use some other approach.

The second example demonstrates joint compilation of two lists into a single data set. To access threaded code, we need the following definitions:

```
: >RR COMPILE REL>ABS COMPILE >R ; IMMEDIATE
: RR@ COMPILE R@ COMPILE ABS>REL ; IMMEDIATE
: ENTER >RR ; \ ( addr -- ) call the threaded code fragment at addr
```

We cannot use just **>R** and **R@** because Win32For uses two different representations for return addresses and data addresses. Since both representations are single-cell, we can use words like **R>** for return address manipulations like removal or reordering of return addresses, but to access threaded code we need words like **RR@**.

```
0 VALUE ^tmp \ top of the area for temporary code

\ === joint recompilation
\ the words any0, any1, get an additional argument, the
\ DEFER word that called them, fetching it from the threaded code
: any0,  ( xt x -- xt) ['] LIT , ,   RR@ CELL- @ ,  3 CELLS ALLOT  DUP , ;
: endl0, ( xt -- xt)   ['] LIT , 0 ,    ['] ENDL ,  3 CELLS ALLOT  DUP , ;
: any1,  ( xt x -- xt) 3 CELLS ALLOT  ['] LIT , ,   RR@ CELL- @ ,  DUP , ;
: endl1, ( xt -- xt)   3 CELLS ALLOT  ['] LIT , 0 ,    ['] ENDL ,  DUP , ;

: recompile-together ( ls1 ls2 xt -- addr )
    HERE >R
    save-ctx
    ^tmp DP !
      ['] any0,  IS NUM      ['] any0,  IS STR      ['] any0,  IS LS
      ['] NOOP   IS BEGL     ['] endl0, IS ENDL
        ROT ( ls2 xt ls1 ) EXECUTE ( ls2 xt )
    HERE >R
    ^tmp DP !
      ['] any1,  IS NUM      ['] any1,  IS STR      ['] any1,  IS LS
      ['] NOOP   IS BEGL     ['] endl1, IS ENDL
        SWAP ( xt ls2 ) EXECUTE ( xt ) DROP
    ^tmp
    HERE R> MAX TO ^tmp
    rest-ctx
    R> DP !
;
\ === the comparison itself ==
\ I: denotes an interpretation stack effect diagram
\ l  denotes a position in the code of a list
: xx=? ( x1 xt1 x2 xt2 -- )    ( I: l -- l ) \ if match
       ( x1 xt1 x2 xt2 -- 0 )    ( I: l -- ) \ if mismatch
       ( x1 xt1 x2 xt2 -- true ) ( I: l -- ) \ if xt is _endl=
    ROT OVER =             \ the same function?
    IF     EXECUTE       \ NB: exec-n of _endl= forces an exit
    ELSE   2DROP DROP 0
    THEN   ( f )
    0= IF R> DROP 0 THEN \ exit list with 0 if mismatch
;
: _str=  ( c-addr1 c-addr2 -- f ) COUNT ROT COUNT COMPARE 0= ;
: _ls=   ( list1 list2 -- f )
    ^tmp >R
    ['] xx=? recompile-together
      ENTER        \ execute the recompiled code
    R> TO ^tmp ;
```

```
: _endl= ( x1 x2 -- true ) ( I: l ra[xx=?] -- )
    2R> 2DROP  2DROP TRUE ; \ exit list and xx=? with true

: cmp-num    ['] =      ; ( n -- n xt )
: cmp-str    ['] _str=  ; ( s -- s xt )
: cmp-ls     ['] _ls=   ; ( ls -- ls xt )
: cmp-endl   ['] _endl= ; ( 0 -- 0 xt )
: eq-lst ( list1 list2 -- f )
    save-ctx
    ['] cmp-num  IS NUM    ['] cmp-str  IS STR    ['] cmp-ls   IS LS
    ['] NOOP     IS BEGL   ['] cmp-endl IS ENDL
    _ls=
    rest-ctx
;
HERE $1000 + TO ^tmp
```

A sample session:

```
' list4 ' list4 eq-lst . -1  ok
' list4 ' list6 eq-lst . 0  ok
```

Our code crucially relies on the ability to predict the size of a list element (3 cells in our case). If we were not able to reference strings compiled into the original list, we would need parallel execution to perform joint compilation. With Scheme lists, the size of an element would not be a problem.

In our code, we implemented context switching using assignments to **DEFER** variables. Alternative approaches to context switching [Gas98b] include: virtual method tables, multiple definitions forming a CASE statement, and the use of search order.

We can mention a large amount of technical details in the code. We consider this as an indication of that we do not yet have adequate expressive means in the programming language.

We do not state that joint recompilation is better than parallel execution or vice versa; we only state that both are applicable in analogous cases. In the case of using methods of code execution other than the traditional sequential one, the method of joint compilation may be preferable. In general, this is one more expressive means.

3. Results and discussion

This paper introduces the concept of *dynamically structured code*, with the characteristic property that relations between elements of executable code are established at run-time. Currently, we can mention three most important principles used with the dynamic codes are: *data execution*, *generation of customized code*, *separation in time*. In general, it is not that important whether code is constant or dynamic, what is important is the structure of code.

The name *dynamically structured codes* is itself an important result. The word "structured" underlines that the relations must be in some kind regular. Although a number of results have been obtained in this direction, what these relations must and/or may be still remains a subject for studies.

Dynamically structured codes, and, in particular, data execution, are perspective programming techniques. We may expect dynamically structured codes to evolve into a discipline of programming that would allow to reduce the complexity of software and thus improve its safety and reduce its cost. Today, programming languages have no special support for dynamically structured codes.

Three new techniques have been proposed in this paper.

The first proposed technique is the method of *dynamic construction of call hierarchy*, DCCH1. It simplifies processing of data records that have complex format, such problems may be found in the area of systems programming. Simplification is indicated by the following metric: instead of three kinds of dependency

between two sorts of code, we get dependency of only one sort. The method is based on two ideas: at first, separation of the record processing into at least two phases (analysis and treatment); at second, at the first phase we construct code that will work at the second phase. This method allows us to find an adequate view on the problem, and develop an adequate factoring. *Call hierarchy diagrams* are proposed as a means of documenting the structure of a DCCH1 program. *DCCH1 is not Forth-specific:* it may be used even in Algol-derivative languages.

The second proposed technique is *data execution with a customized method of execution*, in particular, the use of backtracking to execute data. This technique expands the possible use of data execution.

The third proposed technique is *joint recompilation* of two data sets into a single data set. This method is an alternative to parallel execution of two data sets. We do not state that one of them is better than the other. The proposed technique expands the possible use of data execution.

4. Conclusion

Dynamically structured codes can help us create more simple and therefore more reliable and more maintainable programs. This direction deserves and needs further investigation. What is really important with dynamically structured codes is the structure, and not whether the code is modified at run-time.

The three proposed methods are interesting from both theoretical and practical viewpoints.

References

[CUL89] Craig Chambers, David Ungar, and Elgin Lee, "An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes", in N. Meyrowitz, ed., OOPSLA '89 Conference Proceedings, ACM, Oct., 1989, published as SIGPLAN Notices, 24(10), Oct., 1989.

[Int97] Intel Architecture Software Developer's Manual. Volume 2: Instruction Set Reference. (Order Number 243191) Intel Corporation, P.O. Box 7641, Mt. Prospect IL 60056-7641. Also available at the site http:\\www.intel.com (file name: 24319101.pdf).

[Gas92R] Gassanenko M.L. Novye sintaksicheskie konstruktsii i be`ktreking dlja jazyka Fort.//Problemy tehnologii programmirovanija - SPb: SPIIRAN, 1992, p.148-162. (New Control Structures and Backtracking for the Forth Language, in Russian.) (Note: there's a confusion with the year of publishing, libraries may assign it to 1991, 1992 or 1993).

[Gas94] Gassanenko, M.L. BacFORTH: An Approach to New Control Structures. Proc. of the EuroForth'94 conference, 4-6 November 1994, Royal Hotel, Winchester, UK p.39-41.

[Gas96R] Gassanenko, M.L. Mehanizmy ispolneniya koda v otkrytyh rashiryaemyh sistemah na osnove shitogo koda. Dissertatsiya na soiskanie uchenoj stepeni kandidata fiz.-mat. nauk. SPb, 1996. (Mechanisms of Code Executions in Open Threaded Code Systems. Ph.D. thesis.)

[Gas97R] Gassanenko, M.L. Rasshirenie vozmozhnostej perebora s otkatom (bektrekinga). // Informatsionnye tehnologii i intellektual'nye metody. SPb: SPIIRAN & Izd. TOO "Anatoliya", 1997, p.23-35. (Enhancing the Capabilities of Backtracking, in Russian, see [Gas96]).

[Gas98a] Gassanenko M.L., 1998. Open Interpreter: Portability of Return Stack Manipulations Proc. of the euroFORTH'98 Conf., September 18-21 1998.

[Gas98b] Gassanenko M.L., 1998. Context-Oriented programming. Proc. of the euroFORTH'98 Conf., September 18-21 1998.

[Gas99] Gassanenko, M.L. "Threaded Code Execution and Return Address Manipulations from the Lambda Calculus Viewpoint", proc. of the EuroForth'99 conf. *(in the proceedings of this conference)*.

[Koo90] Koopman, Philip. "TIGRE: Combinator Graph Reduction On The RTX2000", Proc. of the 1990 Rochester Forth Conf., p.82-86.

[KaK98U] Kazimir V.V., Kujvashev D.V. "Do pytannya pro programmnu realizatsiyu merezhi Petri z dynamichnoyu strukturoyu". In: Visn. Chernig. tehnolog. i-tu, 1998. -- no.6, pp.35-42. ("About software implementation of dynamically structured Petri nets", in Ukrainian).

[Rod90] Rodriguez,Bradford J. "Rules Evaluation Through Program Execution." Proc. of the 1990 Rochester Forth Conf., p.123-125.

[Tuz84R] Tuzov, V.A. Matematicheskaya model' yazyka. Leningrad: Izd-vo LGU, 1984, 176 p. (A Mathematical Model of the Language, in Russian.)

[Tuz88R] Tuzov, V.A. Funktsional'nye metody programmirovaniya. // Instrumental'nye sredstva podderzhki programmirovaniya. - Leningrad:LIIAN, 1988. - p.129-143. (The Functional Methods of Programmming (that is, the Russian version of data-driven approach), in Russian.)