Open Interpreter: Portability of Return Stack Manipulations

M.L.Gassanenko
SPIIRAN, St.Petersburg, Russia
mlg@forth.org, mlg@iias.spb.su

Abstract

To make the return stack manipulations portable, we introduce the notion of a stack machine with an open interpreter. The programming techniques, like implementation of new control structures via return stack manipulations, backtracking, and implementation techniques like keeping literals in the threaded code become portable across open interpreter Forths. The open interpreter approach may be used for distributed AI over a range of architectures.

Abbreviations

OI	Open Interpreter
RA	Return Address
IP	Interpretation Pointer
R-stack	the return stack
TC	threaded code
TCE	threaded code element
TCF	threaded code fragment

Introduction

During the last decade, a number of papers have appeared showing how return addresses (RA) manipulations may be used to implement backtracking efficiently on time, which enables one to solve AI problems and the problems like BNF parsing [Rod90a] and pattern matching [Cha91]. In addition, backtracking is a powerful expressive means of a programming language, since it introduces iterators - modules responsible for looking through a set of values(1). When changes in the looking-through algorithm are required, only one module will be affected. Today the problem with backtracking is that RA manipulations are not portable across ANS Forth systems.

Another impressing technique that benefits from return address manipulations is data execution (data-driven approach) [Tuz88][Tuz90][Koo90]. Access to the return stack provides a significantly larger degree of access to the data being executed and of control over the data execution process than may be expected in Lisp, and without any need in a custom data interpreter. Like backtracking, data execution is used in AI and in compiler construction, and other areas also may benefit from it.

The fundamental difference between data manipulations and return addresses manipulations is in that return addresses manipulations may result in control transfers. Control transfers made due to return addresses manipulations radically alter the pattern of code execution. Nevertheless, a formal description of such control transfers exists [Gas95][Gas96].

In Forth, the return stack may be used to:

- keep temporary data
- pass parameters stored in threaded code to procedures
- define new control structures (classical or untraditional)
- implement new methods of program execution (such as backtracking [Cha91][Cha92][Gas94][Rod89][Rod90a][Rod90b])
- implement executable self-processing data (data-driven approach) [Tuz88][Tuz90][Koo90][Rod90a][Rod90b])

The current standard [ANS94] allows only the first. The other methods that employ the return stack are important as either programming techniques or implementation techniques.

Let us consider the following definition:

```
: LIT  R@ @  R> CELL+ >R ;
```

It is an example on accessing data stored in threaded code, which is an important implementation technique, and its formal description has been one of the most significant challenges solved in [Gas95]. The word `LIT` is the most common means to implement the run-time semantics of `LITERAL` :

```
: LITERAL  COMPILE LIT , ; IMMEDIATE
```

In view of this definition, the following definitions are equivalent:

```
: t1  [ 5 ] LITERAL . ;
: t2  LIT [ 5 , ] . ;
: t3  5 . ;
```

Although there are only 6 words (don't forget the final `EXIT`), there are problems: it works on 16-bit one-segment Forths for DOS (e.g. Beta-Forth, Info-Forth, MPE Forths) and some 32-bit Forths, but on F-PC it should be:

```
: LIT  2R@ @L  2R> 2+ 2>R ;
```

on Win32Forth:

```
: LIT  R@ ABS>REL @  R> CELL+ >R ;
```

and on a 8-bit i8051 Forth:

```
: LIT  2R@ @C  2R> 1. D+ 2>R ;
```

(the latter resembles F-PC).

To make return stack manipulations that affect the control flow portable, we introduce the notion of *a stack machine with an open interpreter*. In terms of that machine, all the above definitions consist of words that perform the same functionality. These definitions are different only because they use words that reflect peculiarities of different architectures. ***If we rewrite the above definitions in terms of the open interpreter machine, their source codes will be identical.***

The open interpreter machine provides a semantics under which definitions that use RA manipulations can be written in a portable form. This provides a way to create portable applications that use backtracking and data-driven approach. The open interpreter approach may be also used for distributed systems, since it makes it possible, in principle, to execute the same code using the abovementioned AI techniques on various architectures.

The four classes of open interpreter Forths

We introduce 4 classes of open interpreter Forths:
Class 1. Return addresses have the same format as data addresses, the system uses threaded code which resides in data memory (2). The mentioned 16-bit Forths, and the MPE 32-bit Forth for DOS + DOS extender belong to this class.
Class 2. Return addresses are 1 cell wide, but their representation may be different from data addresses representation. Threaded code resides in data memory. Win32Forth belongs to this class.
Class 3. Return addresses are 1 cell wide, their representation may be different from that of data addresses. Threaded code may reside in a separate memory, and special words may be required to access that memory (to this class belong Harvard architecture systems(3)).
Class 4. Return addresses may be more than 1 cell wide, and special words may be required to access threaded code. Both F-PC and the Harvard 8-bit i8051 Forth belong to this class.
Each class is a subclass of the next class. (End of the definition.)

How do we do return addresses manipulations in a portable way for these systems?
For Class 1 systems, there are no problems: we may use the words `R>` , `R@` , etc. to manipulate with the return addresses, and the words `@` , `C@` , etc. to access data in threaded code.
For Class 2 systems, we introduce the words `RR>` , `RR@` , etc. to manipulate with the return addresses. These words perform the conversion from the return address format to the data address format and/or vice versa. (Such notation has an extra benefit: the mnemonics unambiguously indicates that we want to manipulate with return addresses, not with usual data.)
For Class 3 systems, we introduce the words `/@` , `/C@` , etc. that access the executable code memory.
For Class 4 systems, the words `>RR` , `RR@` , etc. manipulate with double-cell (single, triple, etc.) values, converting them to and from the return address format if necessary. The words `/@` , `/C@` , etc. take addresses in the format that is used to represent RA's on the data stack. We have to note that it is possible to manipulate return addresses and access threaded code elements portable even on Class 4 systems.

For native code compilers, we require that they emulate a system of one of these classes, if they (the compilers) pretend to provide open interpreter features. (A possible approach to implementation is to use a kind of threaded code as the intermediate code and compile to native code from that code).

Let us elaborate on the Class 4 architectures. The words like `DUP` and `SWAP` cannot be used to manipulate return addresses portably. We should use the phrase `>RR RR@ RR>` instead of `DUP` or `2DUP` to duplicate the return address on the data stack portably, and `>RR >RR< RR>` instead of `SWAP` or `2SWAP` ; the phrase `R> R> SWAP >R >R` or `2R> 2R> 2SWAP 2>R 2>R` becomes `RR> >RR< >RR` . The standard arithmetic operators cannot be used on return addresses. As in the Class 3 case, the standard words like `@` and `C@` cannot be used to access literals in the threaded code.

Class 1 is most suitable for scientific studies (the formalism [Gas95] [Gas96] is developed for Class 1 systems, although it will not be difficult to rewrite the proofs for Class 2 and Class 3; for Class 4 the formalism would become very cumbersome because the formalism needs to manipulate both return addresses and data on the data stack even if the code does not). Class 4 is the widest one, and the Class 4 code is most portable while still looking tidy. The problem with writing for Class 4 is that it prohibits the use of stack primitives like `DUP` to manipulate return addresses. If your code will never run on Class 4 systems, this restriction may look a bit too serious. From practical considerations, it may be recommended to write applications for Class 3 or at least for Class 2, mainly because the words like `RR>` clearly indicate that you did intend to manipulate with the return addresses rather than with the data (the same difference as between `2+` and `CELL+`).

The portable (Class 4) definition for LIT is

```
: LIT  RR@ /@  RR> /CELL+ >RR ;
```

The definition

```
: LIT  R@ @  R> CELL+ >R ;
```

is portable across Class 1 systems.

Definition of the open interpreter stack machine

The OI Forth executable code will be called *threaded code* (4).

Threaded code may be viewed as a sequence of *compiled tokens* of procedures (in the simplest case they are just addresses of procedures) (5).

To be more precise, TC may contain the following kinds of threaded code elements (TCEs):

- compiled tokens of procedures
- references to threaded code (branch destinations are represented in this format)
- in-line data

only compiled tokens of procedures are processed by the interpreter.

There are two kinds of procedures: primitives (procedures implemented in hardware, machine code or a language other than Forth) and high-level procedures (the body of a high-level procedure is a threaded code fragment (TCF), and execution of such procedure amounts to calling this code fragment).

The threaded code interpreter (the "inner" interpreter of Forth) has:

- a register (IP, the interpretation pointer) that points to the next TCE to be processed, and
- a stack (the return stack), to which the interpreter saves IP when it calls a threaded code fragment and from which it loads IP exiting the threaded code fragment.

Together, IP and the return stack form the *interpretation stack.*

A *code pointer* is an address of a threaded code element (or, which is the same, an address of a threaded code fragment starting from that TCE).

There are two kinds of interpretation stack information. The return stack may contain:

- code pointers
- other data.

The code pointers reflect the currently unfinished calls. IP, the top element of the interpretation stack, always contains a code pointer. Only code pointers may be processed by the interpreter itself. The procedures that place "other data" onto the return stack procedures are written so that these data do not interfere with the interpreter.

The logic of procedure calls is the same as with other programming languages, with the difference that procedure parameters and local variables are not placed onto the same stack as the return addresses. We describe procedure calls in terms of threaded code fragment (TCF) calls because a high-level procedure call is a particular case of a TCF call (namely, a call of the procedure's body).

- When a TCF (a high-level procedure) is called, a new code pointer (the address of the TCF) is placed on top of the interpretation stack.
- When the TCF (the high-level procedure) is exited, its code pointer is removed from the interpretation stack.

Note that we introduce a dual view on the interpreter stack, which is the key to understanding the return addresses manipulations. The threaded code interpreter considers the return stack and the interpretation pointer (IP) as a single stack. The programmer manipulates only with the return stack, because IP changes while the programmer's code executes, and writing to IP will result in an immediate control transfer. On the other hand, this does not make a restriction: when we call an auxiliary procedure, the return stack becomes what the interpretation stack was. Any changes that have to be done with the interpretation stack, this procedure does with the return stack. When the procedure exits, the interpretation stack becomes what the return stack was.

So, the rule of thumb is: *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 compiled tokens of procedures are processed by the interpreter in the following way: the interpreter fetches the compiled token pointed to by IP, advances IP to the next compiled token, and executes the procedure denoted by the fetched compiled token.

We make no assumptions on how primitives are executed, we know only their effect. In the case of a high-level procedure, we have a choice: we may consider it as a black box with some effect, or we may choose to know that a new code pointer is placed on top of the interpretation stack and execution continues using that code pointer.

The word `EXIT` removes the top element of the interpretation stack. Typically, this word is used to return to the calling definition.

The stack notation

Our stack notation follows the [ANS94] notation:

Stack parameters input to and output from a definition are described using the notation:

```
( stack-id  before -- after )
```

where *stack-id* specifies which stack is being described, *before* represents the stack-parameter data types before execution of the definition and *after* represents them after execution. ... The control-flow-stack *stack-id* is "C:", the data-stack *stack-id* is "S:", and the return-stack *stack-id* is "R:". When there is no confusion, the data-stack *stack-id* may be omitted.

When there are alternate *after* representations, they are described by "*after1* | *after2*". The top of the stack is to the right. Only those stack items required for or provided by execution of the definition are shown.

In addition to the quoted above, when there are alternate *before* and *after* representations, they are described as

```
( stack-id11  before11 -- after11 ) ( stack-id12  before12 -- after12 ) ...
( stack-id21  before21 -- after21 ) ( stack-id22  before22 -- after22 ) ...
                                    ....
```

where stack effect descriptions placed on different lines refer to different alternatives.

The interpretation stack *stack-id* is "R&IP:".

The symbols used in *before* and *after* are those used in [ANS94] and the following:

`ra`	--	return address (an aligned threaded code element address)
`tca`	--	a threaded code element address (the same as *ra*)
`ra[ <data> ]`	--	return address, at which <*data*> are stored
`ra+`	--	advanced (by the size of data stored at *ra*) return address *ra*
`addr[ <data> ]`	--	address, at which <*data*> are stored
`u-tca`	--	an address of a threaded code element, unaligned.

The representation of RA's on the data and the return stacks may be different. The implementation may require that return addresses are in some way aligned, but adding the size of a cell to a properly aligned TCA must yield a properly aligned TCA. Compiled tokens and threaded code references must impose identical alignment requirements, adding the size of a threaded code reference to a properly aligned TCA must yield a properly aligned TCA. Only in-line data elements may be located at unaligned addresses.

Stored data notation:

`ref`	--	threaded code reference

`"abc"` -- a character string containing the text "abc"
`c"abc"` -- a counted string: text "abc" follows the count byte
`ct` -- compiled token

A dotted pair *format.data* describes the format in which *data* are stored.

Format specifiers:

symbol	size in memory	meaning
`x.`	1 cell	one cell stored "as is"
`ref.`	1 reference	a threaded code reference
`c.`	1 chars	one character
`ct.`	may vary	compiled token

Examples:

`ref.a-addr` -- cell-aligned address stored as a threaded code reference
`ref.tca` -- a threaded code element address stored as a threaded code reference
`x.a-addr` -- cell-aligned address stored "as is"
`x.xt1` -- execution token stored "as is"
`ct.xt1` -- the threaded code representation (interpretable by the inner interpreter) of the execution semantics identified by *xt1*

The default interpretation stack effect for a word is `( R&IP: ra -- ra )`, it is the interpreter who makes the IP advance. (The interpreter advances IP *before* calling the procedure; the system must, at least, keep this illusion).

Examples:

```
NOOP    ( R&IP: ra -- ra )
>R      ( x -- ) ( R&IP: ra -- x ra )
(S")    ( -- ra ) ( ra[ c"abc" ] -- ra+ )      \ run-time semantics of S"
BRANCH  ( R&IP: ra1[ ref.ra2 ] -- ra2 )        \ run-time semantics of AHEAD
?BRANCH ( false -- ) ( R&IP: ra1[ ref.ra2 ] -- ra2 ) \ run-time semantics of IF
        ( true -- ) ( R&IP: ra1[ ref.ra2 ] -- ra1+ )
LIT     ( -- n ) ( R&IP: ra[ n ] -- ra+ )
COUNT   ( c-addr[ c.len "abc" ] -- c-addr["abc"] len )
@       ( a-addr[ x ] -- x )
```

The number in brackets following the stack effect of a word shows the number of the class starting from which this word is necesary. For example, the notation

`>RR  ( ra -- ) ( R: -- ra ) [2]`

means that this word must be used if we write code for Classes 2-4 (for Class 1, we may use just `>R`).

The open interpreter wordset

`EXIT ( R&IP: ra1 ra2 -- ra1 ) [1]` Transfer control to the address which is at the return stack top. In terms of the interpretation stack, its top element *ra2* is removed. In terms of the return stack, IP is loaded with the address *ra1* taken off the return stack.

`>RR  ( ra -- ) ( R: -- ra ) [2]` Move *ra* from the data stack to the return stack.

`RR>  ( -- ra ) ( R: ra -- ) [2]` Move *ra* from the return stack to the data stack.

`RR@  ( -- ra ) ( R: ra -- ra ) [2]` Copy *ra* from the return stack top to the data stack.

`RRDROP ( -- ) ( R: ra -- ) [2]` Remove *ra* from the return stack.

`COPY>RR ( ra -- ra ) ( R: -- ra ) [2]` Copy *ra* from the data stack to the return stack (6).

`>RR< ( ra1 -- ra2 ) ( R: ra2 -- ra1 ) [2]` Exchange *ra1* at the data top with *ra2* at the return stack top (6). For Class 4 systems, where RA's may occupy any number of stack elements, this is the only means to change the order of return addresses.

`/ALIGNED ( u-tca1 -- tca2 ) [1]` *tca2* is the first address greater than or equal to *u-tca1* at which a compiled token or a reference may be located.

`/@  ( tca[ x ] -- x ) [3]` Fetch the one-cell literal data located at *tca*.

`/C@  ( u-tca[ c ] -- x ) [3]` Fetch the character literal data located at *tca*.

`/CELL+ ( u-tca1 -- u-tca2 ) [3]` Advance *tca1* by the size of a cell. For Class1 - Class 3 systems, this word is equivalent to `CELL+`.

`/+ ( n u-tca1 -- u-tca2 )` *[3]* Add *n* to *tca1*. The result *tca2* may be misaligned.

`/C, ( char -- )` *[3]* Reserve space for one character in the threaded code space and store *char* in the space.

`/, ( x -- )` *[3]* Reserve one cell of threaded code space and store x in the cell. If the threaded code space pointer is aligned on a TCE boundary when `/,` begins execution, it will remain aligned on a TCE boundary when `/,` finishes execution.

Note. Starting from Class 2, there is no guarantee that `>RR RR>` is equivalent to `NOOP` if the data stack top does not contain a proper return stack address.

The open interpreter extension wordset

`TOKEN, ( xt -- )` *[1]* Add a compiled token of the procedure identified by *xt* to the current threaded code fragment (that is, the one being currently compiled). The same as `COMPILE,` .

`TOKEN@ ( tca[ ct ] -- xt )` *[1]* Decode the compiled token at *tca* and return the execution token of the procedure which semantics is stored at *tca*.

`TOKEN+ ( tca1-- tca2 )` *[1]* Decode the compiled token at *tca* and return the address of the next threaded code element.

`TOKEN> ( tca1[ ct ] -- tca2 xt )` *[1]* Decode the compiled token at *tca* and return the address of the next threaded code element and the execution token of the procedure which semantics is stored at *tca*. Equivalent to `>RR RR@ TOKEN+ RR> TOKEN@` .

`REF@ ( tca1[ ref.tca2 ] -- tca2 )` *[1]* Return *tca2*--the address to which points the reference at *tca1*.

`REF+ ( tca1[ ref.tca2 ] -- tca3 )` *[1]* Advance *tca1* by the size of the reference stored at *tca1*.

`REF- ( tca3 -- tca1[ ref.tca2 ] )` *[1]* Decrease *tca3* by the size of the reference stored immediately before this address. The phrase `REF- REF+` is equivalent to `NOOP` .

`REF! ( tca-dest tca-orig -- )` *[1]* Store *tca-dest* to the reference at *tca-orig.* After execution of this word, the reference at *tca-orig* points to *tca-dest*.

`>TCODE ( xt -- tca )` *[1]* Return the address *tca* of the threaded code fragment which is called when the high-level procedure identified by *xt* is executed.

`>MARK ( -- orig )` *[1]* Start construction of a forward reference: add a reference to the current threaded code fragment, *orig* identifies that reference. The destination address of the reference is undefined until the reference construction is complete. See `>RESOLVE` .

`>RESOLVE ( orig -- )` *[1]* Complete construction of a forward reference: when a new element will be added to the current TCF, the reference at *orig* will point to this new TCE.

`<MARK ( -- dest )` *[1]* Start construction of a backward reference: the destination address *dest* identifies the next TC element that will be added to the current TCF.

`<RESOLVE ( dest -- )` *[1]* Complete construction of a backward reference: add a reference to the current TCF, *dest* identifies its destination address.

`/MOVE ( addr u u-tca -- )` *[3]* If *u* is greater than 0, copy the contents of *u* address units at *tca* to the *u* address units at *addr*. The source address *tca* is at the stack top because there are problems with manipulations with *tca*'s on Class 4 systems.

`/XSWAP ( u-tca x -- x ra )` *[4]* Exchange *u-tca* and *x* (*x* is at the stack top).

`X/SWAP ( x ra -- ra x )` *[4]* Exchange *x* and *u-tca* (*u-tca* is at the stack top).

`>RX< ( ra -- x ) ( R: x -- ra )` *[2]* Exchange *ra* (at the data stack top) and *x* (at the return stack top). A mnemonic rule: *ra* goes, *x* returns.

`>XR< ( x -- ra ) ( R: ra -- x )` *[2]* Exchange *x* (at the data stack top) and *ra* (at the return stack top). A mnemonic rule: *x* goes, *ra* returns.

Note 1. With the help of words `TOKEN>` and the word `>.NAME` (xt -- ; print the name of the procedure identified by *xt*) it is possible to write a portable decompiler.

Note 2. There are specific problems on Class 4 systems. When we use the word `/+` , we may get an improper (e.g., improperly aligned) return address. We cannot place it onto the return stack with `>RR` because conversion to the internal return address format may lead to unpredictable results. We cannot use the stack primitives like `SWAP` to rearrange the data on the stack

because the size of a return address is unknown. Although it *is* still possible to write portable code for Class 4 architecture (finally, there are `/XSWAP` and `X/SWAP`), one should seriously consider writing for Class 3 instead.

Note 3. If you write code for Class 4, you have to test your code on at least two systems with different return address sizes to ensure that it does not use any system peculiarities which you did not mention. Of course, there is always a possibility that some peculiarity is present at both your systems, and you use namely this one. There is the same problem with the portable floating-point code (a standard system is allowed to keep FP numbers on the data stack or on a separate FP stack), and even with the size of a character: if you don't have a system where CHARS is not NOOP, you cannot be sure that you have not missed a CHARS calculating an argument for a MOVE.

Note 4. If the code is written using the open interpreter words, it is easier to find the places dependent on the interpreter architecture in the code. So, even if you probably do use peculiarities of the interpreter, the places where these peculiarities are important are explicitly marked.

The open interpreter toolkit wordset

`RP@ ( -- x )` ***[1]*** Return a system-dependent value identifying the current depth of the return stack. This value may be used by `RP!` .

`RP! ( x1 -- ) ( R: i*x -- j*x )` ***[1]*** Set the return stack depth to be the one specified by *x1*. If the new stack depth is greaer than the old stack depth, the contents of the new return stack elements are undefined. Ambiguous condition: *x1* does not correspond to any valid depth of the return stack.

The purpose of these words is to provide non-local exits (which is required, for example, for a Prolog-like *cut* statement). These words are proposed in recognition of the fact that they are present on most (if not all) Forth systems. The words `RDEPTH ( -- n )` and `RDEPTH! ( n -- )` (obtain and set the depth of the return stack, correspondingly) would also do for non-local exits; their disadvantage is that they are not used in the current practice.

`RADDR@ ( a-addr -- ra )` ***[4]*** Fetch the return address *ra* stored at *a-addr*.

`RADDR! ( ra a-addr -- )` ***[4]*** Store the return address *ra* at *a-addr*.

`RADDR+ ( a-addr1 -- a-addr2 )` ***[4]*** Add the size in address units of a RA to *a-addr1*, giving *a-addr2*.

`RADDR- ( a-addr1 -- a-addr2 )` ***[4]*** Subtract the size in address units of a RA from *a-addr1*, giving *a-addr2*.

These words enable us to make portable implementations of additional stacks that hold return addresses.

Examples I: Portable access to literals in the threaded code

The first example is the definition of the word `LIT` above. The second example is a portable (Class 4) definition of the standard word `."` .

```
: /TYPE ( tca[ c"text" ] -- )
    >RR
    1  RR@ /C@       ( i imax )
    BEGIN
            2DUP U> 0=
    WHILE
            OVER RR@ /+ /C@ EMIT
            SWAP 1+ SWAP
    REPEAT
    2DROP RRDROP
;
: (.") ( tca[ c"text" ] -- tca+ )
    RR@                                                 \ addr of the string
        RR@ /C@ CHAR+ RR> /+ /ALIGNED >RR      \ get around the string
    /TYPE
;
: ." ( Compilation: "text<">" -- )
    COMPILE (.")
    [CHAR] " WORD
    DUP C@ 1+  0
    ?DO
        DUP I + C@ /C,
    LOOP
    DROP
    /ALIGN
```

```
; IMMEDIATE
```

We cannot define the words `C"` and `S"` this way, because they return data space addresses. Their portable definition has no relation to the return addresses manipulations:

```
: C"
    POSTPONE AHEAD
    HERE >R
    [CHAR] " PARSE TUCK HERE PLACE 1+ ALLOT /ALIGN
    POSTPONE THEN
    R> POSTPONE LITERAL
; IMMEDIATE
\ : PLACE ( from len to -- ) 2DUP C! 1+ SWAP CHARS MOVE ;
```

In this definition, the only word required from the open interpreter wordset is `/ALIGN`.

Examples II: Portable implementations of control structures

This is a portable (Class 4) definition of the *recursive block* control structure. The words `START` and `EMERGE` denote the block boundaries, the word `DIVE` denotes the recursive call of a block.

```
: (CALL)   RR@ REF+ >RR< REF@ >RR ;
: (START)  RR@ REF@ >RR< REF+ >RR ;
: START     ?COMP COMPILE (START) >MARK <MARK 201 ; IMMEDIATE
: EMERGE    ?COMP 201 ?PAIRS DROP COMPILE EXIT >RESOLVE ; IMMEDIATE
: (DIVE#) ( an .. a2 # --> )
    DEPTH 1                  ( n 1 )
    DO                       ( an .. a1 # )
        I PICK 201 =
        IF
                I 2 + PICK 201 =
                I 2 + PICK 201 = AND
                IF
                        1- DUP 0=
                        IF
                                DROP
                                COMPILE (CALL) I 2 + PICK <RESOLVE
                                UNLOOP EXIT
        THEN    THEN    THEN
    LOOP
 1 ABORT" more than the nesting level"
;
: DIVE        \ call the enclosing START-EMERGE block
      ?COMP 1 (DIVE#) ; IMMEDIATE
: DIVE# ( "n" -- ) \ call the n-th enclosing START-EMERGE block
                   \ DIVE# 1 is equivalent to DIVE
      ?COMP BL WORD NUMBER DROP (DIVE#) ; IMMEDIATE
: .FACT ( n -- )
    DUP .
    START
        DUP 0>
        IF      DUP 1- DIVE *
        ELSE    DROP 1
        THEN
    EMERGE
    ." ! = " .
;
```

The word `.FACT` is a trivial example on this control structure. When we execute `5 .FACT`, it prints "`5 ! = 120`".

The non-portable version of the word `(DIVE#)` analysed the compiled code to check if the stack element below the number 201 is a reference after the compiled token of `(START)`. In the portable version, the number 201 is placed on the stack 3 times. Strictly speaking, nothing can guarantee that `START` is the only word in the system that leaves 201 on the stack 3 times, but it is very improbable. Other approaches to correctness checking are possible, e.g. to keep the stack depth corresponding to the word `START` in a variable, save the old value of the variable in the word `START` and restore it in the word `EMERGE`.

Examples III: Portable backtracking

Here we will give only the simplest example of backtracking. The word `TTT` prints numbers from 1 to 10.

```
: ENTER  ( tca -- ) ( R&IP: -- tca ) \ call the code fragment at tca
       >RR ;
: SUCC POSTPONE RR@    POSTPONE ENTER ; IMMEDIATE
: FAIL POSTPONE RRDROP POSTPONE EXIT ; IMMEDIATE
: 1-10
    1
    BEGIN
        SUCC
        1+
    DUP 11 =
    UNTIL
    DROP FAIL
;
: TTT
    1-10 DUP .
;
```

Why this technique is important. If we write a new interpreter in any programming language, the new interpreter becomes, in the best case, an order of magnitude slower. If we extend the existing interprerter with the new operations, only these new operations slow the execution. If the interpreter extensions are invoked in 10% cases, and each invokes 10 Forth primitives, the overall speed decreases only by a factor of 2. If the interpreter extensions are written in Assembly, the loss of speed is negligible. In addition, the interpreter extension approach is less labor-consuming and less error-prone, since the old interpreter is reused.

Related work

C.Jakeman in [Jak96] proposed a small wordset (6 words) that allowed him to make the FOSM string matcher package [Cha91][Cha92] an ANS Forth program with environmental dependencies. Jakeman noticed that to implement backtracking, only a few top return stack elements have to be manipulated; this is absolutely true. To rearrange return addresses, Jakeman stores them temporarily onto the data stack and into an auxiliary global variable (it is not possible to use the stack primitives because the size of return addresses may vary across different systems). The 6 words move RA's between the stacks, duplicate and remove RA's, read a RA from the auxiliary variable and store it there. Access to threaded code elements is not considered.

Conclusion

A strict definition of the stack machine with an open interpreter has been given. For an OI Forth, the mechanism of code execution is defined, and a set of operations is provided to access the interpreter stack and the threaded code. The return stack techniques like backtracking, control structures implemented via return stack manipulations, keeping literals in the threaded code become portable across the open interpreter Forths.

Notes

(1) The word "`1-10`" in the "Examples III: Portable backtracking" section may serve as an example of such module.

(2) "Data memory" is an ANS Forth term, it is the memory which may be accessed by the words `@` , `C@` , etc.

(3) The Harvard systems with single-cell return addresses and no conversion from the RA format to the data-in-code-memory address format could form a class at the level of Class 2. Such class ("Class 2b") has not been introduced because the classification would become less understandable; in addition, the Win32Forth is an example of a system which may be assigned to either "Class2a" (the present Class 2) or "Class2b", but not to both, because RA conversion may be done either by R-stack access words like `RR>` or by TC access words like `/@` . In such situation, incompatible applications for the same system could appear. Finally, the choice was made in favour of "Class2a" because such systems are sometimes used for scientific purposes (for lack of a Class 1 system), while Harvard systems are used mostly in industrial applications where explicit indication of RA manipulations should be encouraged.

(4) Over a long time, the term "threaded code" was used to underline that the code is different from processor's native code. At present, there are executable codes that may be assigned to both threaded and native codes (e.g. stack processors' code, subroutine-threaded code (STC) and STC with primitive inlining (STCI)), and, what is more, differentiation between native

and non-native code is not much meaningful from programmer's point of view. We will use the words "threaded code" to indicate that the executable code has some definite structure and is processed using some definite discipline.

(5) A compiled token is an element of threaded code that denotes a procedure (when the code is executed, the procedure is performed). In the cases of ITC and DTC, a compiled token has a fixed length, typically one cell, and contains the CFA of the procedure. In the case of STC, a compiled token is an instruction that calls the procedure. In the case of STCI, a compiled token is either an instruction that calls the procedure or a sequence of instructions that perform the semantics of the procedure (that is, are functionally equivalent to the procedure call). There are also other varieties of threaded code.

(6) The Class 1 analogs of the words `>RR<` and `COPY>RR` should be called `>R<` and `COPY>R`. For Classes 2-4, `>R<` and `COPY>R` may operate only on values that are not RA's.

References

[ANS94] ANSI, 1994. American National Standard for Information Systems - Programming Languages - Forth. ANSI X3.215-1994 - American National Standard Institute, Inc. - 212 p.

[Cha91] Charlton G., 1991. FOSM, A FOrth String Matcher.//1991 FORML Conference Proceedings, euroFORML'91 Conference, Oct 11-13, 1991, Marianske Lazne, Czechoslovakia, Forth Interest Group, Inc., Oakland, USA, 1992. - P.313-329.

[Cha92] Charlton G., 1992. FOSM, A FOrth String Matcher, continued.//euroForth'92 Conference Proceedings, MPE Ltd., Southampton, UK. - P.113-122.

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

[Gas95] Gassanenko M.L., 1995. Formalization of Return Addresses Manipulations and Control Transfers. //Proc. of the euroFORTH'95 conference, 27-29 October 1995, International Centre for Informatics, Dagstuhl Castle, Germany, 18 p.

[Gas96] Gassanenko M.L., 1996. Formalization of Backtracking in Forth. //Proc. of the euroFORTH'96 Conf., 4-6 October 1996, Hotel Rus, St.Petersburg, Russia, 26 p.

[Jak96] Jakeman, C.M., 1996. Portable Backtracking in ANS Forth. //Proc. of the FORML'96 conf.

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

[Rod89] Rodriguez B., 1989. Pattern matching in Forth.//Proc. of the 1989 FORML Conf.

[Rod90a] Rodriguez B., 1990. A BNF Parser in Forth.//ACM SIGForth Newsletter, vol.2, no.2, December 1990, p.13-18.

[Rod90b] Rodriguez B., 1990. Rules Evaluation through Program Execution.//Proc. of the 1990 Rochester Forth Conf., p.123-125.

[Tuz88] Tuzov V.A., 1988. Funktsional'nye metody programmirovanija./ Instrumentalnye sredstva programmirovanija - L:LIIAN. - p.129-143. (The Functional Methods of Programmming [data execution is a powerful programming technique], in Russian)

[Tuz90] Tuzov V.A., 1990. Jazyki predstavlenija znanij. - L:LGU. (The Languages of Knowledge Representation [what an ideal language of knowledge representation should be], in Russian.)