Threaded Code Execution and Return Address Manipulations from the Lambda Calculus Viewpoint

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

This paper presents a lambda-calculus-based formal model of Forth's mechanism of code execution. This model is able to cope with control transfers implemented via the interpretation stack changes, a Forth technique that rarely can be found in other programming languages. The formal model may be used in compiler construction, program verification, and in computer science lecture courses.

Introduction

We usually assume that Forth uses Reversal Polish Notation (RPN) and the phrase

x f g

means **f**(**g**(**x**)). The words like **SWAP** are not a serious problem because we can consider Forth words as functions of the type *stacks-and-memory-state* → *stacks-and-memory-state* and thus preserve the RPN view. For example,

1 2 + is $f_{+}(f_{\mathtt{LIT2}}(f_{\mathtt{LIT1}}($ *initial-state* $)))$

Surprisingly, even this approach does not work in the general case, because there is a more deep inconsistency between Forth code and RPN. The most well-known counter-example is the word **EXIT**:

1 EXIT 1+ is in no way $f_{1+}(f_{\mathtt{EXIT}}(f_{\mathtt{LIT1}}($ *initial-state* $)))$

because the function **1+** does not execute at all! The techniques of return address manipulations may serve as another counter-example: from the RPN point of view, it is absolutely unclear how the words

```
: xxx R> R> TUCK >R >R >R ;
: yyy R> @ >R ;
```

work (these words alter the control flow by modifying the return stack; the second one is **BRANCH** and the first one is **SUCC**, a word that is used to implement backtracking [Gas94]).

Given the ability to manipulate with the return addresses, and that it is achieved by using Forth-specific means, two questions arise:

1) what is the possibility to access the return stack (in normal mathematical terms)?

2) how can one compile programs that use return address manipulations?

This paper gives two simple answers on these two questions.

In this paper we describe the execution mechanism of Forth in the language of lambda-calculus. We shall use the programming language Scheme avoiding destructive assignment.

This work is related to the work [Gas95] [Gas96] [Gas96R]. In this paper we construct a formal system which satisfies the axioms on which the formalism [Gas95] [Gas96] [Gas96R] is built. These two formalisms are built on different principles, solve different problems, and have different uses. [Gas95] solves the following problem: given the ability to manipulate with the return addresses, explain how this ability may be used. This paper's problem is: given the common mathematical concepts, construct a system which allows Forth-style return address manipulations. Relations between different formalisms are discussed in more detail in the "Related works" section.

Our notation for threaded code fragment addresses will be slightly different from that of [Gas95]: since the symbol `'` (quote) is defined in Scheme, we shall use the symbol `^` to denote the address of a threaded code fragment . The position of code (see [Gas95]) is assumed to be active (code is executed), passive position (code is read as data or written) is not considered.

The techniques of return address manipulations and their use to implement backtracking are described in [Gas94] [Gas96R]; the definition of *Open Interpreter,* an interpreter that permits access to its interpretation stack, may be found in [Gas98] or [Gas96R].

Global memory from the functional point of view

In the spirit of [Tuz84], the function **fetch** has two arguments: memory state and address, and returns the value associated with the specified address in the specified memory state. The function **store** has three arguments: memory state, address and value. Its result is the new memory state. A Scheme implementation of these functions is given in the Appendix 1.

fetch: *memory-state* × *address* → *value*
store: *memory-state* × *address* × *value* → *memory-state*

For stacks, we shall use three functions:

push: *value* × *stack-state* → *stack-state*
pop: *stack-state* → *stack-state*
top: *stack-state* → *value*
second: *stack-state* → *value*

push places a value onto the stack; **pop** removes the top element from the stack, **top** returns the stack top; **second** returns the second element of the stack, **second** ° l***s*. (top (pop *s*)).**

The interpreter NEXT

A Forth primitive is a function whose arguments are: the interpretation pointer IP, the return stack, the data stack, and the global memory. IP points to the next instruction in the threaded code. "Ordinary" Forth words let **next** execute this instruction by calling **next** with unchanged IP.

The Forth "inner" interpreter **next** fetches a function from memory and executes it with advanced IP.

```
(define next
 (lambda (ip rs ds mem)
  ((getfunc ip mem) (advance ip) rs ds mem)
))
```

It fetches a function from memory — `(getfunc ip mem)` — and executes it with unchanged *rs, ds, mem,* and advanced (incremented by the size of the "token" of the function) IP. This is indeed is how **next** works in most Forth systems.

The function `getfunc` fetches the function whose token is stored in the beginning of the threaded code fragment pointed to by IP, using the current memory state. For the purposes of this paper, we assume threaded code to be constant; for sake of simplicity, we implement such "read-only" memory using the Scheme global memory.

A Forth word, for example, **DUP** , corresponds to the following Scheme definition (the prefix "w_" stands for "Forth word"):

```
(define w_DUP
 (lambda (ip rs ds mem)
  (next ip rs
        (push (top ds) ds)
        mem
)))
```

We see the four arguments (*ip rs ds mem*), and we see that when `next` is called, only the data stack state argument is changed. This is how **DUP** works in most implementations, with the only difference that it is usually written in Assembly.

What is surprising is that the fragment of threaded code that follows a compiled Forth word is *an argument* to it. Usually, a Forth word fetches the function at IP and invokes it. But not all Forth words do so. Some Forth words just ignore the code fragment at IP (for example, **EXIT**). A Forth word can call the threaded code fragment at IP multiple times. A Forth word can fetch data inserted into the threaded code (for example, **LIT** does this).

We shall use the following notation for addresses of threaded code fragments:

^[DUP 1+]

is the address of a threaded code fragment containing functions `w_DUP w_1+` . The fragment itself will be denoted as **[DUP 1+]**. (The brackets denote the code fragment boundaries and do not imply any structure, that is, **[[X]]** ≡ **[X]** and **[[X][Y]]** ≡ **[X Y]**).

Two threaded code fragments are said to be equivalent (functionally equivalent) if with any continuation α applications of `next` to their addresses are equivalent:

[X] ≡ **[Y]** ⇔ ∀ α (next **^[X** α **]**) ≡ (next **^[Y** α **]**),

that is,

∀α λ *rs, ds, mem*.(next **^[X** α **]** *rs ds mem*) ≡
≡ λ *rs, ds, mem*.(next **^[Y** α **]** *rs ds mem*)

Note that the threaded code fragments **[X** α **]** and **[Y** α **]** are located in memory *mem*, and the notation **^[X** α **]** denotes an address of a threaded code fragment located in *mem*. Strictly speaking, the above equation implies that both **[X** α **]** and **[Y** α **]** may be found in *mem*. In the program given in the Appendix 1, the Scheme memory is a read-only constant component of the "Forth system" memory, and threaded code fragments are implemented as Scheme lists.

Sequential execution of treaded code fragments

Let us assume that functions `w_DUP` and `w_1+` are defined as

`w_DUP` ≡ λ *ip, rs, ds, mem*.(`next` *ip rs* (`s_DUP` *ds*) *mem*)

```
w_1+  ≡ λ ip, rs, ds, mem.(next ip rs (s_1+ ds) mem)
```

The definitions of functions `s_DUP` and `s_1+` are evident and not that important for our purposes:

```
s_DUP ≡ λ ds.(push (top ds) ds)
s_1+  ≡ λ ds.(push (+ 1 (top ds)) (pop ds))
```

(the prefix "`s_`" means that the only effect of the function is changing the stack).

Let us consider execution of the threaded code fragment **`^[ DUP 1+ ... ]`** with the initial state (*rs, ds, mem*):

```
(next ^[ DUP 1+ ... ] rs ds mem)
```

This reduces to:

```
((getfunc ^[ DUP 1+ ... ] mem) (advance ^[ DUP 1+ ... ]) rs ds mem)
(w_DUP ^[ 1+ ... ]) rs ds mem)
(next ^[ 1+ ... ] rs (s_DUP ds) mem)
((getfunc ^[ 1+ ... ] mem) (advance ^[ 1+ ... ]) rs (s_DUP ds) mem)
(w_1+ ^[ ... ] rs (s_DUP ds) mem)
(next ^[ ... ] rs (s_1+ (s_DUP ds)) mem)
```

That is, the continuation **`[ ... ]`** is executed when the stack state is the result of evaluation of the superposition `(s_1+ (s_DUP ds))`.

We have just proven the following

Statement 1A.
If Forth words **X** and **Y** are implemented by functions `w_X` and `w_Y`

```
w_X ≡ λ ip, rs, ds, mem.(next ip rs (s_X ds) mem)
w_Y ≡ λ ip, rs, ds, mem.(next ip rs (s_Y ds) mem)
```

then execution of the threaded code fragment **`^[ X Y ... ]`**

```
(next ^[ X Y ... ] rs ds mem)
```

will be equivalent to

```
(next ^[ ... ] rs (s_Y (s_X ds)) mem)
```

Indeed, ***the proof*** may be obtained by replacing `w_DUP` and `w_1+` by `w_X` and `w_Y` in the above inference.

This is a well-known fact: if **f** and **g** are "ordinary" Forth words, then

```
x f g
```

evaluates the superposition **f**(**g**(**x**)).

What is really important for "ordinary" Forth words is that they do not change IP in any specific way (that is, they call **next** with the same value of IP as they receive).

Let us introduce triplets *M* = <*rs, ds, mem*> describing the whole state of memory. Applying a function with the "t_" prefix to triplets gives another triplet:

```
(t_DUP M) ≡ (t_DUP rs ds mem) ≡ <rs (s_DUP ds) mem>
```

In general,

```
(t_func M) ≡ <(r_func M) (s_func M) (m_func M)> ≡
≡ <(r_func rs ds mem) (s_func rs ds mem) (m_func rs ds mem)>
```

Analogously, *D* = <*ds, mem*> and

```
(d_func D) ≡ <(s_func D) (m_func D)>
```

Statement 1.

If Forth words **X** and **Y** are implemented by functions w_X and w_Y

```
w_X ≡ λ ip, M.(next ip (t_X M))
w_Y ≡ λ ip, M.(next ip (t_Y M))
```

then execution of the threaded code fragment **^[X Y ...]**

```
(next ^[ X Y ... ] M)
```

will be equivalent to

```
(next ^[ ... ] (t_Y (t_X M)))
```

The ***proof*** is analogous to that of Statement 1A.

The word EXIT

The word **EXIT** is defined as

```
(define w_EXIT
 (lambda (ip rs ds mem)
  (next (top rs) (pop rs) ds mem)
))
```

It ignores the threaded code fragment stored at IP: for any two threaded code fragments α and β the threaded code fragment **[EXIT** α **]** is equivalent to **[EXIT** β **]**.

Statement 2. ∀ α, β ∀ *M* = <*rs ds mem*>

```
(next ^[ EXIT α ] M) ≡ (next ^[ EXIT β ] M)
```

Proof.

```
(next ^[ EXIT α ] rs ds mem)
```

reduces to

```
((getfunc ^[ EXIT α ] mem) (advance ^[ EXIT α ]) rs ds mem)
(w_EXIT ^[ α ] rs ds mem)
(next (top rs) (pop rs) ds mem)
```

Analogously,

`(next ^[ EXIT β ] rs ds mem) ≡ (next (top rs) (pop rs) ds mem)`

This statement is the Axiom 12 from [Gas95].

Let us show that **[^[τ] >R EXIT]** is functionally equivalent to **[τ]** . (That is, the phrase **>R EXIT** executes the threaded code fragment whose address is at the data stack top).

Statement 3. ∀ τ ∀ α ∀ *rs, ds, mem*

`(next ^[ >R EXIT α ] rs (push ^[ τ ] ds) mem) ≡ (next ^[ τ ] rs ds mem)`

Proof.

`(next ^[ >R EXIT α ] rs (push ^[ τ ] ds) mem)`

reduces to:

```
(w_>R ^[ EXIT α ] rs (push ^[ τ ] ds) mem)
(next ^[ EXIT α ] (push ^[ τ ] rs) ds mem)
(w_EXIT ^[ α ] (push ^[ τ ] rs) ds mem)
(next (top (push ^[ τ ] rs)) (pop (push ^[ τ ] rs)) ds mem)
(next ^[ τ ] rs ds mem) ,
```

the definition of `w_>R` may be found in the Appendix 1. This statement is the Axiom 1 from [Gas95].

Nesting calls

The function `nest` is defined as

`nest ≡ λ ip_new, ip, rs, ds, mem.(next ip_new (push ip rs) ds mem)`

A colon definition is application of `nest` to the colon definition's body. For example, the body of the colon definition

: 2* DUP + ;

is the threaded code fragment **[DUP + EXIT]** and the function `w_2*` is an application of `nest` to the address of that threaded code fragment:

```
w_2* ≡ (nest ^[ DUP + EXIT ]) ≡
≡ λ ip, rs, ds, mem.(next ^[ DUP + EXIT ] (push ip rs) ds mem)
```

Let us show that when a colon definition **X**

: X τ **;**

is called with the continuation υ, the resulting code fragment

[X υ **]**

is equivalent to

```
[ ^[ υ ] >R τ ]
```

(the code fragment τ receives control with the address of υ on the return stack).

Statement 4. ∀ τ ∀ υ ∀ *rs, ds, mem*
((nest ^[τ]) ^[υ] *rs ds mem*) ≡ (next ^[τ] (push ^[υ] *rs*) *ds mem*)

Proof may be obtained by substituting the definition of `next`.

This statement corresponds to the Axioms 2 and 3 from [Gas95] (one of these axioms must be considered as a definition of notation).

Access to data in threaded code

The function **LIT** assumes that a data element (a number) follows it; it fetches the number from threaded code and advances IP to the next threaded code element:

```
(define w_LIT
 (lambda (ip rs ds mem)
  (next (advance_lit ip)
        rs
        (push (getlit ip mem) ds)
        mem
)))
```

Control structures

The word **BRANCH** may be defined as

```
: BRANCH R> REF@ >R ;
```

It assumes that a reference to a threaded code fragment follows it.

```
(define w_BRANCH
 (lambda (ip rs ds mem)
  (next (fetch_ref ip mem)
        rs ds mem
)))
```

This word *does not pass control to the code following the reference*, unless the reference points to that code.

Backtracking

Let us construct a function that calls continuation multiple times. The following program may be found in the Appendix 1:

```
: ENTER >R ;
: 1..4 1 R@ ENTER 2 R@ ENTER 3 R@ ENTER 4 R@ ENTER RDROP EXIT ;
: print1..4 1..4 . ;
```

A sample session:

```
print1..4 1 2 3 4  ok
```

ENTER ≡ (nest **^[>R EXIT]** ≡

≡ λ *ip, rs, ds, mem*.(nest **^[>R EXIT]** *ip rs ds mem*)

(nest **^[>R EXIT ...]** *ip rs ds mem*)

reduces to

(next **^[>R EXIT ...]** (push *ip rs*) *ds mem*)
(w_>R **^[EXIT ...]** (push *ip rs*) *ds mem*)
(next **^[EXIT ...]** (push (top *ds*)(push *ip rs*)) (pop *ds*) *mem*)
(w_EXIT **^[...]** (push (top *ds*)(push *ip rs*)) (pop *ds*) *mem*)
(next (top *ds*) (push *ip rs*) (pop *ds*) *mem*)

We shall consider the following backtrackable word:

: XXX α **R@ ENTER** β **R@ ENTER** γ **RDROP EXIT ;**
: YYY XXX η **;**

Let *D* denote <*ds, mem*> (remember that (d_α *D*) ≡ (d_α *ds mem*) ≡ <(s_α *D*) (m_α *D*)>).

(next **^[** α χ **]** *rs ds mem*) ≡ (next **^[** χ **]** *rs* (d_α *D*))
(next **^[** β χ **]** *rs ds mem*) ≡ (next **^[** χ **]** *rs* (d_β *D*))
(next **^[** γ χ **]** *rs ds mem*) ≡ (next **^[** χ **]** *rs* (d_γ *D*)),

the latter three equations mean that we assume threaded code fragments α, β, γ to behave like sequences of "ordinary" Forth words, that is, they pass control to the continuation χ and make no changes on the return stack.

We assume that

(next **^[** η **EXIT ...]** *rs D*) ≡
≡ (w_EXIT ^[...]) *rs* (d_η *D*)) ≡
≡ (next (top *rs*) (pop *rs*) (d_η *D*))

(the code fragment **[** η **EXIT ...]** behaves like a sequence of "ordinary" Forth words followed by **EXIT**, the third expression is obtained from the second one by substituting the definition of w_EXIT).

The threaded code fragment **[R@ ENTER]** ≡ **[SUCC]** executes as:
(next **^[R@ ENTER ...]** *rs ds mem*)
(next **^[ENTER ...]** *rs* (push (top *rs*) *ds*) *mem*)
(next (top *rs*) (push **^[...]** *rs*) *ds mem*)

The threaded code fragment **[RDROP EXIT]** ≡ **[FAIL]** executes as:
(next **^[RDROP EXIT ...]** *rs ds mem*)
(next **^[EXIT ...]** (pop *rs*) *ds mem*)
(w_EXIT (pop *rs*) *ds mem*)
(next (top (pop *rs*)) (pop (pop *rs*)) *ds mem*)

Let us see how the threaded code fragment
[XXX η **EXIT ...]**
executes:
(next **^[XXX** η **EXIT ...]** *rs D*)
(nest **^[** α **SUCC** β **SUCC** γ **FAIL] ^[** η **EXIT ...]** *rs D*)

```
(next ^[ α SUCC β SUCC γ FAIL ] (push ^[ η EXIT ... ] rs) D)
(next ^[ SUCC β SUCC γ FAIL ] (push ^[ η EXIT ... ] rs) (d_α D))
(next ^[ η EXIT ...] (push ^[ β SUCC γ FAIL ] (push ^[ η EXIT ... ] rs))
(d_α D))
(next ^[ β SUCC γ FAIL ] (push ^[ η EXIT ... ] rs) (d_η (d_α D)))
(next ^[ SUCC γ FAIL ] (push ^[ η EXIT ... ] rs) (d_β(d_η(d_α D))))
(next ^[ η EXIT ...] (push ^[ β SUCC γ FAIL ] rs) (d_β(d_η(d_α D))))
(next ^[ γ FAIL ] (push ^[ η EXIT ... ] rs) (d_η(d_β(d_η(d_α D)))))
(next ^[ FAIL ] (push ^[ η EXIT ... ] rs) (d_γ(d_η(d_β(s_η(d_α D))))))
(next (top rs) (pop rs) (d_γ(d_η(d_β(s_η(d_α D))))))
```

So,

```
YYY ≡ (nest ^[ XXX η EXIT ])
XXX ≡ (nest ^[  α R@ ENTER β R@ ENTER γ RDROP EXIT ])
```

And

```
(YYY ip rs D) ≡ (nest ^[ XXX η EXIT ] ip rs D)
```

reduces to

```
(next ^[ XXX η EXIT ] (push ip rs) D)
(next (top (push ip rs)) (pop (push ip rs)) (d_γ(d_η(d_β(d_η(d_α D))))))
(next ip rs (d_γ(d_η(d_β(d_η(d_α D))))))
```

Does Forth really use RPN?

Yes, it does, but not always. Let us repeat the above example in other words:

```
: ENTER >R ;
\ SUCC ° R@ ENTER
\ FAIL ° RDROP EXIT
: XXX a SUCC b SUCC  g FAIL ;
: YYY XXX h ;
```

and execution of ***x*** **YYY** calculates

$$g(h(b(h(a(x)))))$$

A Forth word treats IP and the threaded code fragment at IP as *its arguments;* it can ignore it, or fetch data from it, or call it once, or call it multiple times.

Results

Now we can give answers on the questions mentioned in the Introduction section.

1) What is the possibility to access the return stack (in normal mathematical terms)?

The Forth interpreter **next** is the following construct:

next ≡ λ *ip, rs, ds, mem*.((getfunc *ip mem*) (advance *ip*) *rs ds mem*)

`getfunc` returns the function whose *compiled token* is in memory *mem* at address *ip*, `advance` returns the address of the next compiled token.

Each Forth function **f** has the following form:

f ≡ λ *ip, rs, ds, mem*.(**next** (**f_ip** *ip rs ds mem*)
(**f_rs** *ip rs ds mem*)
(**f_ds** *ip rs ds mem*)
(**f_mem** *ip rs ds mem*))

For "ordinary" Forth functions, **f_ip** returns just *ip:*

dup ≡ λ *ip, rs, ds, mem*.(**next** *ip rs* (**push** (**top** *ds*) *ds*) *mem*)
r> ≡ λ *ip, rs, ds, mem*.(**next** *ip* (**pop** *rs*) (**push** (**top** *rs*) *ds*) *mem*)
! ≡ λ *ip, rs, ds, mem*.(**next** *ip rs* (**pop** (**pop** *ds*)) (**store** *mem* (**top** *ds*) (**second** *ds*)))

If **f_ip** returns something different than just *ip*, then there is a transfer of control in threaded code. For example,

branch ≡ λ *ip, rs, ds, mem*.(**next** (fetch_ref *ip mem*) *rs ds mem*)

The function **stop** stops execution of threaded code and returns the state of IP, stacks and memory:

stop ≡ λ *ip, rs, ds, mem*.(list *ip rs ds mem*)

This function allows us to speak about results of threaded code evaluation, but is not used in real application software.

An important point is that *ip, rs, ds, mem* are arguments to Forth functions. This explains why Forth functions are allowed to do anything with them.

The return stack is not a hidden system mechanism to implement function call nesting. Instead, Forth provides a means which can, in particular case, implement nesting calls.

nest ≡ λ *code_addr, ip, rs, ds, mem*.(**next** *code_addr* (**push** *ip rs*) *ds mem*)
exit ≡ λ *ip, rs, ds, mem*.(**next** (**top** *rs*) (**pop** *rs*) *ds mem*)

2) How can one compile programs that use return address manipulations?

According to the "compilation as partial evaluation" principle. When you know that the code is read-only (you probably will have to find this out), you can calculate the effect of **next** at compile-time. If you can detect that threaded code is not read as data, you can eliminate it. In 99% of cases you will encounter common control structures, but if you find out that the programmer has implemented a control structure that you cannot translate to machine code, you can use the following optimization: a threaded code fragment whose internals are not accessed from outside (as data or transferring control into it) is replaced with an equivalent threaded code fragment consisting of a single compiled token. Inside the function whose token is compiled there, you will be again able to do optimizations.

3) This formalism enables us to reason about properties of threaded code fragments. For example, in this paper we prove statements that correspond to axioms of [Gas95].

Related works and discussion

Now there is a number of formalisms that describe Forth.

The first stream, Poial-Stoddart-Knaggs' algebra of stack effects [Poi91] [StK91] [StK92] [Poi93] [StK93] [Sto93] [Poi94] [Sto96], describes stack effects of Forth words and is able to introduce a type system to Forth. These works are focused on "ordinary" Forth words and do not consider control transfers due to return address manipulations. Among these works, only [Sto96] adequately describes the mechanism of threaded code execution.

The Gassanenko's calculus of threaded code fragments [Gas95] [Gas96] [Gas96R] describes how return address manipulations affect the control flow. [Gas95] describes control flow alone; it assumes that when knowledge about effects of stack operators, etc. is required, it is available from some other source.

This work introduces a lambda-calculus-based model of Forth. In this model, we were able to prove statements that correspond to the most important axioms of [Gas95]. This formalism and [Gas95] are based on different principles, give different results, and their possible uses are also different.

Basic principles. [Gas95] enables us to rewrite threaded code fragments preserving the functional semantics. [Gas95] can tell us what some code fragment is equivalent to, but it cannot tell us what that threaded code fragment does. This formalism enables us to compare threaded code fragments examining what they reduce to (what they do).

Results. [Gas95] is a language that enables us to reason about threaded code fragments with respect to return address manipulations. It elucidates the properties of threaded code fragments and interference between return stack access and procedure calls. One of the most significant results is explanation of how a threaded code fragment must be transformed when it is put into a colon definition, given the possibility of interference between return address manipulations and the mechanism of procedure calls, and assuming that the procedure must be functionally equivalent to the threaded code fragment.

The theorem of 1-st order call equivalence [Gas95]. If `R1( τ )` exists, then

$$\tau \equiv [\text{r R1}(\ \tau\ )\ ']$$

where `[r τ ']` denotes a call of the threaded code fragment τ;

`R1( τ )` is a such code fragment that $\forall N$ (where N is a one-cell value) N `>R R1( τ )` $\equiv \tau\ N$ `>R` .

Examples:

```
R1( DROP ) ≡ DROP
R1( R> DROP ) ≡ R> R> DROP >R
R1( >R ) ≡ R> SWAP >R >R
R1( EXIT ) ≡ R> DROP EXIT
: myEXIT R> DROP ; ( the word ; compiles EXIT , you can see that myEXIT ≡ EXIT )
R1( LIT ) — considered meaningless, the residue of code in passive position is a problem
```

The phrase "1-st order" means that we consider calls that are used with the traditional sequential code execution; 2-nd order calls are used with backtracking [Gas96].

In terms of our formal model, we can propose the following expression as an equivalent to `R1( τ )` : if τ processes its continuation only by means of **next** (that is, in the context τ β , β is in active position, either processed by **next** or not processed at all),

```
R1( τ ) ≡ [ λ ip0, rs0, ds0, mem0 .
        (next ^[  τ  [ λ ip1, rs1, ds1, mem1 .(next ip0 (push (top rs0) rs1) ds1 mem1) ] ]
              (pop rs0) ds0 mem0 ) ]
```

The proof shows that applications of **next** to both `^[` N `>R R1( τ ) ... ]` and `^[ τ` N `>R ... ]` reduce to
`(next ^[ τ [ λ ip, rs, ds, mem .(next ^[ ... ] (push` N `rs) ds mem) ] ] )`

Construction of R1(τ) in terms of Forth words in the general case is still a problem. At least, $\neg\exists\alpha,\omega\ \forall\tau$ (R1(τ) ≡ α τ ω) ∧ (τ is in active position in this context).

The following rule of thumb may be inferred from the theorem of 1-st order call equivalence or from the observation that when a procedure is called, IP is pushed onto the return stack. To write a procedure that

changes IP and the return stack: consider IP and the return stack as a single *interpretation stack* (IP is its top); decide which changes must be done with 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.

Possible uses. The formalism presented in this paper is easier to use in compilers. The [Gas95] axioms are "bidirectional": one can substitute the right-hand side for the left-hand one, or vice versa. To use such rules on the computer, some kind of AI is required. In the case of lambda calculus, there is an algorithm for evaluation of expressions, and we do not need AI. On the other hand, [Gas95] notation is more laconic and its approach is more adequate to human reasoning. [Gas95] enables *humans to reason* about properties and behaviour of threaded code fragments; this work enables *computers to calculate* the results of return address manipulations.

Additional notes on dynamic code generation and comparison of formalisms may be found in Appendices 1 and 2.

Another important feature of this formal model is its simplicity: it may serve as an introduction to return stack manipulations for people with no knowledge of Forth.

Conclusion

This paper elucidates one more aspect of return address manipulations. It proposes a relatively simple system built from lambda terms that allows return address manipulations. The key point is that a function compiled into a threaded code fragment treats the residue of the threaded code fragment as an argument. The proposed formal model may be used in compiler construction, program verification, and in computer science lection courses.

References

[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.

[Gas95] Gassanenko, M.L. 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.

[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.)

[Gas98] Gassanenko M.L., 1998. Open Interpreter: Portability of Return Stack Manipulations Proc. of the euroFORTH'98 Conf., September 18-21 1998.

[Poi91] Poial J. Multiple Stack Effects of Forth Programs. Proc. of the euroFORML'91 Conf. (Oct 11-13, 1991, Marianske Lazne, Czechoslovakia), in: 1991 FORML Conference Proceedings, Forth Interest Group, Inc., Oakland, USA, 1992, 400-406.

[Poi93] Poial J. Some Ideas on Formal Specification of Forth Programs. 9th euroFORTH conference on the FORTH programming language and FORTH processors, Oct 15-18, 1993, Marianske Lazne, Czech Republic, 1993, 4 pp.

[Poi94] Poial, Jaanus. Forth and Formal Language Theory. Proc. of the EuroForth'94 conference, 4-6 November 1994, Royal Hotel, Winchester, UK, 6 pp.

[StK91] Stoddart B., Knaggs P. Type Inference in Stack Based Languages. Proc. of the euroFORML'91 Conf. (Oct 11-13, 1991, Marianske Lazne, Czechoslovakia), in: 1991 FORML Conference Proceedings, Forth Interest Group, Inc., Oakland, USA, 1992, 407-421.

[StK92] Stoddart B., Knaggs P. The (Almost) Complete Theory of Forth Type Inference. Proc. of the 1992 EuroForth Conference.

[StK93] Stoddart B., Knaggs P. Type Inference in Stack Based Languages. Formal Aspects of Computing, BCS, 1993, 5, 289-298.

[Sto93] Stoddart B. Self Reference and Type Inference in Forth. 9th euroFORTH conference on the FORTH programming language and FORTH processors, Oct 15-18, 1993, Marianske Lazne, Czech Republic, 1993, 9 pp.

[Sto96] Stoddart, B. An Event Calculus Model of the Forth Programming System.//Proc. of the euroFORTH'96 Conf., 4-6 October 1996, Hotel Rus, St.Petersburg, Russia, 17 p.

[Tuz84] Tuzov, V.A. Matematicheskaya model' yazyka. Leningrad: Izd-vo LGU, 1984, 176 p. (A Mathematical Model of the Language, in Russian.)

Appendix 1. A Forth interpreter in Scheme (only most essential elements)

```
; === stacks ===
(define push cons)      ; (elem stack) --> stack'
(define top car)        ; (stack) --> elem
(define second cadr)    ; (stack) --> elem
(define pop cdr)        ; (stack) --> stack'

; === memory ===
; memory is a list of associations (address, value)
; NB: host cells (read-only, they include code memory) are not in this list.
(define allocate
 (lambda (mem addr val) ; --> mem'
  (cons (list addr val) mem)
))
(define free
 (lambda (mem addr) ; --> mem'
    (define remove-assoc
     (lambda (newmem oldmem addr)
      (cond
        ( (null? oldmem) newmem )
        ( (equal? (caar oldmem) addr)
              (remove-assoc newmem (cdr oldmem) addr)
        )
        ( else
              (remove-assoc
                  (cons (car oldmem) newmem)
                  (cdr oldmem)
                  addr
    ))) )     )
    (remove-assoc () mem addr)
))
(define fetch
 (lambda (mem addr) ; --> val
  (define val (assoc addr mem))
  (if val
      (cadr val)
      (fetch-host addr) ; code memory is a part of memory
)))
(define store
 (lambda (mem addr val)
   (allocate (free mem addr) addr val)
))
; host memory (the Scheme's global memory) is constant and read-only
; we place threaded code in this memory
(define fetch-host car) ; addr --> val
```

```
(define nextaddr
 (lambda (addr)
   (if (number? addr) ; if the address is a number,
       (+ addr 1)    ; increment it
       (cdr addr)    ; else addr must be a head of a list
)) )

; === access to "tokens" of functions in the code ===
(define getfunc
 (lambda (ip mem) ; --> func
  (fetch mem ip)
))
(define getlit getfunc) ; (ip mem) --> literal
;; advance IP by the size of function token
(define advance nextaddr)    ; (ip) --> ip'
;; advance IP by the size of literal
(define advance_lit nextaddr); (ip) --> ip'

; === the code interpreter of Forth ===
(define next
 (lambda (ip rs ds mem)
  ((getfunc ip mem) (advance ip) rs ds mem)
))
(define w_EXIT
 (lambda (ip rs ds mem)
  (next (top rs) (pop rs) ds mem)
))
(define nest
 (lambda (ip_new ip rs ds mem)
  (next ip_new (push ip rs) ds mem)
))

; === arithmetic functions ===
(define w_+
 (lambda (ip rs ds mem)
  (next ip rs
       (push (+ (top ds) (second ds)) (pop (pop ds)) )
       mem
)))
(define w_-
 (lambda (ip rs ds mem)
  (next ip rs
       (push (- (second ds) (top ds)) (pop (pop ds)) )
       mem
)))
(define w_.
 (lambda (ip rs ds mem)
  (display (top ds))
  (display " ")
  (next ip rs (pop ds) mem)
))

; === stack operators ===
(define w_DUP
 (lambda (ip rs ds mem)
  (next ip rs
       (push (top ds) ds)
       mem
)))
(define w_SWAP
 (lambda (ip rs ds mem)
  (next ip rs
       (push
           (second ds)
           (push
               (top ds)
               (pop (pop ds))
       )   )
       mem
)))

; === access to memory ===
(define w_@
 (lambda (ip rs ds mem)
  (next ip rs
       (push
```

```
            (fetch mem (top ds))
            (pop ds)
        )
        mem
)))
(define w_!
 (lambda (ip rs ds mem)
  (next ip rs
        (pop (pop ds))
        (store mem (top ds) (second ds))
)))

; === access to the return stack ===
(define w_R>
 (lambda (ip rs ds mem)
  (next ip
        (pop rs)
        (push (top rs) ds)
        mem
)))
(define w_R@
 (lambda (ip rs ds mem)
  (next ip rs
        (push (top rs) ds)
        mem
)))
(define w_>R
 (lambda (ip rs ds mem)
  (next ip
        (push (top ds) rs)
        (pop ds)
        mem
)))
(define w_RDROP
 (lambda (ip rs ds mem)
  (next
        ip (pop rs) ds mem
)))

; === literals ===
(define w_LIT
 (lambda (ip rs ds mem)
  (next (advance_lit ip)
        rs
        (push (getlit ip mem) ds)
        mem
)))

; === a colon definition ===
; --- : 2 DUP + ;
(define w_2*
 (lambda  (ip rs ds mem)
  (nest (list w_DUP w_+ w_EXIT) ip rs ds mem
)))

; === an example of backtracking ===
; --- : ENTER >R ;
(define w_ENTER
 (lambda  (ip rs ds mem)
  (nest (list w_>R w_EXIT) ip rs ds mem
)))

; --- : 1..4 1 R@ ENTER 2 R@ ENTER 3 R@ ENTER 4 R@ ENTER RDROP EXIT ;
(define w_1..4
 (lambda  (ip rs ds mem)
  (nest
    (list
        w_LIT 1 w_R@ w_ENTER
        w_LIT 2 w_R@ w_ENTER
        w_LIT 3 w_R@ w_ENTER
        w_LIT 4 w_R@ w_ENTER
        w_RDROP w_EXIT
    )
    ip rs ds mem
)))
```

```
; --- : print1..4 1..4 . ;
(define w_print1..4
 (lambda  (ip rs ds mem)
  (nest
    (list
        w_1..4 w_. w_EXIT
    )
    ip rs ds mem
)))

; === stop, "print all and stop", run ===
; returns the state of stacks and memory
(define stop
 (lambda (ip rs ds mem)
  (list ip rs ds mem)
))

; shows the state of stacks and memory
; following the common practice, stack top is shown as the rightmost element
(define write&stop
 (lambda (ip rs ds mem)
  (display "\n=== RS: ")
  (write (reverse rs))
  (display " === DS: ")
  (write (reverse ds))
  (display " === MEM: ")
  (write (reverse mem))
))

;; Example of use:
;; to run   1 2 . .
;; type in  (run w_lit 1 w_lit 2 w_. w_. )
(define run
 (lambda x
  (next (append x (list write&stop)) () () () )
))
(define w_.s
 (lambda (ip rs ds mem)
  (newline)
  (write (list ip rs ds mem))
  (next ip rs ds mem)
))

; === how Forth code runs ===
; executing:    LIT 1 . stop
; note that "." is an argument to "LIT" !
(define test_1_.
 (lambda ()
  (w_LIT (list 1 w_. stop) () () ()))
))

; ====================================
(define w_1+
 (lambda (ip rs ds mem)
  (next ip rs
        (push (+ 1 (top ds)) (pop ds))
        mem
)))
(define w_2/
 (lambda (ip rs ds mem)
  (next ip rs
        (push (/ (top ds) 2) (pop ds))
        mem
)))
```

Notes

1. In our model, each number is a valid address of a cell.
2. In our model, each pointer to a pair is an address of a "host system's" read-only cell (*car* returns the contents of that address).
3. Although it is extremely convenient to represent threaded code fragments as Scheme lists (that is, to place code into the "host" memory), this interpreter is able to execute the phrase

```
(run  w_lit w_lit w_lit 1 w_!      w_lit 1234  w_lit 2 w_!
      w_lit w_.   w_lit 3 w_!      w_lit stop  w_lit 4 w_!
      w_lit 1 w_>R w_EXIT)
```

that is,

['] lit 1 !   1234 2 !   ['] . 3 !   ['] stop 4 !   1 >R EXIT

This phrase will print 1234.

We have to note that although our model is able to represent dynamic code generation, we still do not know much about this technique. We have a strictly defined mathematical object whose properties are unknown. The only thing we know is that it reduces to **`1234 .`** plus some side effect, which may be enough to compile it. Return address manipulations are in a much better way because aside from this formal model we have both formal and informal knowledge about their properties and use.

Appendix 2

Here we show how the [Gas95] formalism copes with the backtracking example:

```
: ENTER >R ;
\ SUCC ° R@ ENTER
\ FAIL ° RDROP EXIT
: XXX a SUCC b SUCC g FAIL ;
: YYY XXX h ;
```

In the [Gas95] notation,

```
ENTER == [r >R ; ']
XXX == [r α SUCC β SUCC γ FAIL ']
YYY == [r XXX η ; ']
```

At first, ∀ α

```
'[ α ] ENTER == '[ α ] [r >R ; '] == [r '[ α ] >R ; '] == [r α ']
'r[ α ] SUCC =='r[ α ] R@ ENTER == 'r[ α ] [r α ']
```

Then we get (the notation α:R means that α neither depends on the return stack state nor changes it) :

∀ α:R, β:R, γ:R, η:R

```
YYY == [r XXX η ; '] ==
[r [r α SUCC β SUCC γ FAIL '] η ; '] ==
[r 'r[ η ; ] α SUCC β SUCC γ FAIL '] ==
[r α 'r[ η ; ] SUCC β SUCC γ FAIL '] ==
[r α 'r[ η ; ] [r η ; '] β SUCC γ FAIL '] ==
[r α 'r[ η ; ] η β SUCC γ FAIL '] ==
[r α η β 'r[ η ; ] SUCC γ FAIL '] ==
[r α η β η γ 'r[ η ; ] RDROP ; '] ==
[r α η β η γ ; '] == α η β η γ
```