Efficient “reversibility” with guards and choice

Bill Stoddart
School of Computing and Mathematics
University of Teesside, North Yorkshire, U.K.

August 17, 2002

Abstract

We describe reversibility mechanisms incorporated into a native code Forth used an an intermediate language for a B-GSL compiler. In contrast to our previous work, information preservation is limited to what is needed to implement the B-GSL semantics for non-deterministic choice and guard. Design choices are discussed with reference to the Pentium architecture. The use of guards and choice in Forth is illustrated with the Knight's Tour.

1 Introduction

The work reported here is part of a project to integrate backtracking into the formal software development method B.[1]. B originated in the Programming Research Group at Oxford University in the 1980's. It provides a language for describing systems at various levels of abstraction. At the most abstract level, which is used for specification, the language is not generally executable, being able to describe the effects of an operation implicitly. At the implementation level it provides simple integer programming language, B0. Users write both a specification and an implementation. Proofs must be discharged to show that implementations satisfy their specifications and to ensure the preservations of invariant data properties. When integrating code from an already implemented module, formal analysis is done using the specification of the imported module, not its implementation. Commercial toolkits are available which generate the necessary proof obligations, attempt to discharge them automatically, and translate the execution level language, B0, into C, ADA or assembler according to customer requirements. A notable industrial application of B is was the complete control system of the Paris Metro line 14, which uses driverless trains. B was used to generate 100,000 lines of Ada. No errors were detected in this code at any stage of its testing by the French railway authorities, and as a result of this experience these authorities now accept proved B developments without acceptance testing.

A major problem with B arises from proof obligations not discharged automatically by the in-built prover. Keeping these to a minimum and using manual intervention to discharge the remaining ones requires time and skill. We are investigating whether this situation can be eased, for certain kinds of application, by increasing the expressive power of B0 to include the constructs “choice” and “guard”. These provide a form of backtracking.

In [10] we showed how semantics of reversibility are incorporated in the mathematical apparatus of B and suggested a reversible stack based virtual machine as an implementation vehicle. Reversibility was obtained by using multiple code field threading to effectively obtain three virtual machines in one, capable respectively of normal, conservative and reverse execution.

In this paper we turn our attention to the design of a reversible virtual machine where efficiency considerations favour a native code implementation. An implementation mechanism is described in detail for the Intel Pentium architecture. We use the term reversibility rather than backtracking in the title of this paper since our approach is based on designing a *virtual machine* which preserves information during computation. Reversibility provides mechanisms to guarantee the return of the system to any pre-selected previous state and deals automatically with the collection of any garbage generated during the corresponding forward computation. [2] [11]

The rest of the paper is organised as follows. In section 2 we briefly mention related work in Forth. In section 3 we present a simple case study: the Knight's Tour, in which we illustrate the use of the choice and guard constructs in Forth. In section 4 we give the formal semantics of backtracking. In section 5 we discuss the Intel architecture and attempt to justify the choice of a native code implementation. In section 6 we describe our virtual machine organisation for forward execution and discuss optimization. In section 7 we discuss reverse execution and guards. In section 8 we discuss the implementation of non-deterministic choice. In section 9 we draw our conclusions.

2 Backtracking and Reversible Computing in Forth

As an extensible language Forth has a long history of proposed techniques and extensions to handle backtracking. Most of these involve manipulation of the return stack, which is no longer permissible under ANS Forth. Brad Rodriguez described a backtracking Forth in his Masters Dissertation and also published a top down backtracking BNF parser [8]. Gordon Charlton wrote FOSM, a Forth String Matcher in which the Forth Data stack is used to hold pertinent events and recovery data which are used during backtracking [3].

Michael Gassanenko has explored the control mechanisms, including backtracking, which can be constructed within a model based Forth language, and has developed this work into a specification for an open standard for return stack semantics.[7]

Henry Baker has advocated Forth as a language for reversible computation and has drawn the connections between reversibility, information loss, thermodynamics and garbage collection [2]. Peter Bishop at Adelard, in work funded under the UK nuclear research program, has investigated using a reversible Forth to achieve fail-safety.

Some the above work builds on properties of a particular implementation of the Forth virtual machine. For example Michael Gassanenko assumes the correctness of the definition `: BRANCH R> @ >R ;` ANSForth, by abstracting away from any particular implementation, removes our ability to conjure a simple backtracking mechanism from existing primitives.[1].

In this work we have chosen to incorporate reversibility at the level of virtual machine design, partly for efficiency reasons, and partly because reversibility gives us exactly the correct semantics for choice and guard.

3 An Example: The Knights Tour

We use reversibility to add choice and guards to Forth. In B, Communicating Sequential Processes, and many other formal notations a choice between A and B is written $A \, [] \, B$. In Forth we need to bracket the choice construct and we use:

```
<CHOICE A [] B ... CHOICE>
```

At a choice construct execution makes a provisional choice, but may later reverse back to this point and make a different choice.

The guard $\boxed{\texttt{-->}}$ removes a flag from the stack. If true, execution continues ahead. Otherwise execution reverses to the most recent choice still having an unexplored alternative. If no such choice the user is given a response of "ko" rather than "ok" to signify an impossible request.

We also need reversible versions of words which change memory. The reversible version of $\boxed{\texttt{!}}$ is named $\boxed{\texttt{!\_}}$ and so on.

We now present a simple example in which we use guard and choice to help solve the Knight's Tour problem. Our algorithm takes as input a starting position and attempts to find a path by which a knight can visit all the remaining squares of a chessboard without visiting any square twice. We use numbers 0..63 to represent the squares of the board. A brute force approach is used. The method is to code a loop in which the loop body proposes, checks and records a move. The loop terminates when a path which covers the whole board has been found.

```
VARIABLE POSN \ holds the current position of the knight

CREATE VISITS 256 ALLOT \ records visited squares

CREATE ROUTE 64 ALLOT \ holds the route

VARIABLE MOVES \ counts moves 0 .. 64

: INIT ( -- )
  VISITS 64 CELLS ERASE
  ROUTE 64 ERASE
  0 MOVES ! ;

: VISIT ( n --  record a visit to square n)
  CELL * VISITS +  -1 SWAP !_ ;

: ?VIRGIN ( n -- f  return true iff n has been visited)
```

[1]But see the communication from A Ertl at www.computing.tuwien.ac.at/forth/backtracking-in-ansforth

```
  CELL * VISITS + @ NOT ;

( Obtaining rank and file of a given square )
: RANK ( n1 -- n2 ) POSN @ 2/ 2/ 2/ ;
: FILE ( n1 -- n2 ) POSN @ 7 AND ;

: SELECT-MOVE ( -- n )
( choose a valid move from current posn, n is new posn )
  <CHOICE
    RANK 6 <  FILE 7 <  AND  -->  16 1 +
  []
    RANK 6 <  FILE 0 >  AND  -->  16 1 -
  []
    RANK 7 <  FILE 6 <  AND  -->  8 2 +
  []
    RANK 7 <  FILE 1 >  AND  -->  8 2 -
  []
    RANK 0 >  FILE 6 <  AND  -->  -8 2 +
  []
    RANK 0 >  FILE 1 >  AND  -->  -8 2 -
  []
    RANK 1 >  FILE 7 <  AND  -->  -16 1 +
  []
    RANK 1 >  FILE 0 >  AND  -->  -16 1 -
  CHOICE> POSN @ +  ;

: CHECK-MOVE ( n -- n  reverse if square n has been visited )
  DUP ?VIRGIN --> ;

VARIABLE TOTAL-MOVES

: RECORD-MOVE ( n -- )
  DUP POSN !_
  DUP VISIT
  ROUTE MOVES @ + C!_
  1 MOVES +!_  ;

: KTOUR ( n --  construct a knight's tour from square n)
  INIT RECORD-MOVE
  BEGIN
    SELECT-MOVE
    CHECK-MOVE RECORD-MOVE
    MOVES @  64 =
  UNTIL ;

( Example run )
63 KTOUR ROUTE 64 OCTAL DUMP 07 EMIT

77  65  73  61  53  74  66  54 75  67  55  76  64  72  60  52
71  63  51  70  62  50  42  34 46  27  35  56  44  36  57  45
```

```
37  16  24  43  31  10  2  23 4  12  0  21  40  32  20  41
33  14  6  25  17  5  13  1 22  30  11  3  15  7  26  47 ok
```

In the top level word we enter a loop in which we choose a move, check its valiidity and record it in the path. The loop terminates when we have a path of 64 moves (counting the initial placement of the knight as a move). The example solution is found after 17,739,768 provisional moves.

4 Reversible Computing and Program Semantics

Logical analysis of programs for correctness implies some way of expressing the meaning of program operations in a logical form. We do this in B using the method of predicate transformers. When reading this forget C. $x = 3$ is nothing to do with assignment. Rather it is a predicate whose truth is based on the value of x. $x := x + 1$ is an assignment operation which, by changing x, can transform the truth of a predicate based on x.

Write $[S]Q$ for the condition that operation S will establish predicate Q. For example:

$$[x := x + 1]x = 3$$

is the condition that executing $x := x+1$ will establish $x = 3$. That condition is $x = 2$, so:

$$[x := x + 1]x = 3 \Leftrightarrow x = 2$$

i.e. the operation $x := x + 1$ will establish $x = 3$ if and only if $x = 2$. Mechanically we can calculate this by substituting $x + 1$ for x in $x = 3$ giving $x + 1 = 3$ i.e. $x = 2$. The whole semantics can be thought of as an extension of the idea of substitution, which gives us the "Generalised Substitution Language" and "B-GSL".

Non-deterministic choice between operations S and T is written $S \mathbin{[]} T$. Using $\wedge$ for logical and it has the rule:

$$[S \mathbin{[]} T]Q = [S]Q \wedge [T]Q$$

Meaning: if a non deterministic choice must establish some condition, both branches of the choice must be sure to do so. We are protecting ourselves against demonic choice, also known as sods law.

Another primitive construct is the guard. $g \longrightarrow S$ (" g guards S") will mean, in our reversible world, do S if g is true, otherwise reverse. Its logical rule is:

$$[g \longrightarrow S]Q \Leftrightarrow g \Rightarrow [S]Q$$

Choice is governed by guards. Using $\neg$ for logical not, consider:

$$g \longrightarrow S \mathbin{[]} \neg\, g \longrightarrow T$$

this construct must do S if g is true and T otherwise. It is equivalent to:

if g then S else T end

We name the operation that does nothing *skip*. We need it for the logical analysis of:

if g then S end

which is analysed as:

$g \longrightarrow S \,[]\, \neg\, g \longrightarrow \mathit{skip}$

An interesting operation is $\mathit{false} \longrightarrow \mathit{skip}$. Using the given rules we have:

$[\mathit{false} \longrightarrow \mathit{skip}]Q \Leftrightarrow$
$\mathit{false} \Rightarrow Q \Leftrightarrow$
true (since false implies anything)

So this operation will establish any condition we may wish for. Of course it is not a real operation, except in reversible computation. In out language it will either be a choice which is never taken, or, if it is presented as the only choice, it causes execution to reverse. For its logical properties it has long been known as *MAGIC*.

The rule for sequential composition $S;\ T$ is:

$[S;\ T]Q \Leftrightarrow [S][T]Q$

We now have all the rules required for a semantics of reversibility. Consider the "program":

$S \mathrel{\widehat{=}} (x := 1 \,[]\, x := 2);\ x = 2 \longrightarrow \mathit{skip}$

According to our rules this will assign $x := 2$. The operational interpretation is that if the choice $x := 1$ is made the guard will be false and execution will reverse to the choice construct and take the remaining choice $x := 2$. Now the guard is true and the program terminates with $x = 2$. The formal proof has two parts. The first is to show $[S]x = 2$. The second is to show the result is not due to "magic". For the second part we must show there is something S cannot establish: $\neg\, [S]\mathit{false}$.

5 Intel Pentium Architecture

Many Forth systems for the Pentium have opted for a native code style of implementation with subroutine threading and in-lining of short definitions. E.g. Chuck Moore's ColorForth, Bernd Paysan's BigForth, and commercial Forths from Forth Inc and MPE. In this section we briefly review the current generation Pentium architecture[4] [5] [6] and attempt to justify having taken the same implementation decision.

The Pentium family inherits the i386 instruction set, itself a lightly modified 32 bit form of the 8086 instruction set dating from 1976. The instruction set is compact, with many 8 bit instructions, but unlike RISC instruction sets it was not designed with the idea that opcode bits would drive processor logic in a fairly direct way. For this reason Pentium machine code instructions are processed via a pipeline which first translates them into micro operations: "μops". For example a call is converted int 4 μops. The μops are executed by a dispatch unit capable of handling up to 5 μops in parallel. Where logical dependencies allow it the dispatch unit may permit out of order execution. Results are then written back to memory or registers by a "retirement unit". In total this constitutes a 20 stage pipeline. The processor is connected to separate data and code caches, and assuming code being executed is available in the level 0 cache the pipeline is fed by fetches which bring 32 bytes at a time from the cache[2]. Whether this is all usable machine code will depend on branches. If branching to code at the last byte of a 32 byte cache line, the fetch obtain only one byte of executable code. Correspondingly if executable code from the cache line contains a branch, the code beyond the branch will not be used unless it follows a conditional branch which is not taken. Thus branches disrupt the flow of code through the pipeline and the first rule of Pentium optimisation is to avoid them[4]. Least disruptive is a return from a matching call. This is because the processor maintains a 16 entry shadow return stack. When a call is executed one of the actions is to push its return address on to this internal stack. When a return occurs the internal return stack is used to predict the return address.

Subroutine threading is the implementation technique of choice since it exploits return prediction and allows us to remove branches altogether: most Forth primitives are short enough to compile in line. We can also apply simple but effective peephole optimisations. To consider this in more detail we need to describe the organisation of the virtual machine.

6 Machine organisation during forward execution

The virtual machine architecture has return and parameter stacks, a frame pointer for local variables, and a history stack. The allocation of physical to logical registers is not fixed but there is a canonical form which is taken at any transfer or branch. The canonical allocation of virtual machine stack pointers to hardware components is as follows (note that %esp and %esi are i386 registers and hsp is simply a memory location labelled with that name)

Forth	i386
parameter stack	%esi
return stack	%esp
frame pointer	%edi
history stack	hsp (memory)

The code definition of $\boxed{+}$ is written as follows:[3].

```
CODE + ( n1 n2 -- n3 \n3 = n1 + n2 )
   xchg %esp,%esi
   pop %eax
```

[2]Specific figures refer to the P4

[3]Our meta-compiler level code definitions are written in Gnu i386 assembler, with the the meta-compiler handling headers and control structures. Unlike Intel's own assembler, the source operand of a two operand instruction is to the left.

```
   pop %edx
   add %edx,%eax
   push %eax
   xchg %esp,%esi
   ret
ENDCODE
```

This generates 9 bytes of code, including a 1 byte return.

It is very common for code definitions to begin and end with an exchange of stack pointers, as we see here. Long definitions are invoked via a call. Short definitions (the user specifies the meaning of short) are compiled in-line, and peephole optimisation is applied at the joins. If `+ +` were to be compiled, optimisation would remove the following operations at the join:

```
   push %eax
   xchg %esp,%esi
   xchg %esp,%esi
   pop %eax
```

This is a saving of 6 bytes, so in line compilation of `+ +` generates 10 bytes of code, exactly the same as two calls. Also note that it results in the top element of the virtual machine stack being transferred between the two operations via the `%eax` register.

The coding style we have adopted for primitives is not always optimal for individual operations, but optimises nicely for sequences of primitives. We expect such sequences will occur more often in the Forth generated by our compiler than they do in Forth written by a human expert.

7 Reverse execution: restoring changed memory

$\boxed{!\_}$ performs the normal Forth store function, but also records the address of the store, and the previous value at that address, on the history stack. Reverse execution will restore the overwritten memory value. Words such as $\boxed{!\_}$ whose effects are to be undone during reverse execution, achieve this effect, in part, by depositing on the history stack the information needed to restore state, and the execution address of the operation that is going to perform the restoration. A series of gnu assembler macros have been provided to assist the process. For example, the following transfers three values (which may be any immediate, register or memory operands) to the history stack:

```
.macro hpush3 rm1 rm2 rm3
# pushes rm1 rm2 rm3 to hstack, rm3 will be top
   xchg hsp,%esp
   push \rm1
   push \rm2
   push \rm3
   xchg hsp,%esp
.endm
```

This macro is used in $\boxed{\texttt{!\_}}$[4]

```
CODE !_ ( x addr -- "store_")
   xchg %esp,%esi
   pop %eax # address for store
   mov (%eax),%edx #get current contents
   hpush3 %eax %edx $STORE_r
   pop (%eax)
   xchg %esp,%esi
   ret
ENDCODE
```

The values pushed onto the history stack are the data required to restore the original machine state and the address of the operation which will perform the reverse execution.

Machine organisation during reverse execution takes the following form:

Forth	i386
parameter stack	%esi
return stack	hsp
history stack	%esp

The switch to reverse computation is exemplified by $\boxed{\textbf{MAGIC}}$ which always forces it to occur:

```
CODE MAGIC ( -- )
   xchg hsp,%esp #point %esp at hstack
   ret #enter the most recently deposited reverse operation
   noop
ENDCODE MUST-IN-LINE
```

All that is needed is to exchange stacks and return into the code of the most recently deposited reverse operation.[5] The usual way in which execution is reversed is at a guard. $\boxed{\texttt{-->}}$ removes and tests a parameter stack flag. If zero, it switches to reverse execution, otherwise allows execution to continue ahead.

```
CODE --> ( f --   "guards")
   lodsl # pop %eax from the parameter stack
         # i.e. %eax := f || %esi := %esi+4
   test %eax,%eax
   if zero; # reverse
     xchg hsp,%esp #point %esp at hstack
     ret #enter the most recently deposited reverse operation
   endif
   noop
ENDCODE MUST-IN-LINE
```

Reverse operations find their parameters on the i386 stack, and after consuming them they return into the following reverse operation (return threading!). Note

[4]gnu assembler detail, brackets denote indirection. `mov (%eax),%edx` moves the contents of the location pointed to by `%eax` into `%edx`.

[5]Additional details: $\boxed{\textbf{MUST-IN-LINE}}$ tells the compiler that the most recent definition must always be compiled as in line code. The noop placates the optimiser, which would otherwise remove the `ret` when compiling this code in line.

that they are never "called", but only returned to. Here is an example: the restore operation for store. Its coding relies on the way the history stack is primed during the execution of a matching $\boxed{\texttt{!\_}}$.

```
STORE_r:
   pop %edx #old contents
   pop %eax #address
   mov %edx,(%eax) #restore old contents
   ret # return into the next reverse computation
```

Unlike our previous model[9] we no longer record the effects of pure stack operations for subsequent reverse execution. This does impose some limitations, e.g. given:

```
: 1[]2 <CHOICE 1 [] 2 CHOICE> ;
```

`10 1[]2 +` cannot be reversed because the value 10 will not be re-established on the stack. Where we need to pass a value across a choice and that value is going to be consumed prior to possible backtracking, we must use a variable. We can however use `10 1[]2 OVER +` since that leaves the original stack values unchanged, and the stack pointer *is* re-established by reverse execution.

8 Choice

The specification of the choice construct does not detail which choice should be taken first, but we make the implementation choice that the first choice will be the first chosen. Consider the choice construct `<CHOICE A [] B [] C CHOICE>`. We begin our explanation by describing the corresponding assembler code generated by the meta compiler. In the following, A, B and C represent the assembler code corresponding to A, B and C respectively. The automatically generated labels would differ depending on the point in the application where the code was compiled, but would have the same inter-relationship. Bearing this in mind, the code would appear as follows:

```
# <CHOICE
   choice_prefix _L226
   A
   jmp _L228
# []
_L226
   choice_prefix _L227
   B
   jmp _L228
# []
_L227
#  final choice, no choice prefix needed
   C
# CHOICE>
_L228
```

For all choices except the last, we may, following the choice, at some point backtrack to the choice construct. In that case we need to restore the value of the parameter stack and frame pointers. The history stack is primed to achieve this by the `choice_prefix` macro. This takes one argument: the label of the following choice.

```
.macro choice_prefix label
# This code prefixes each of the choices in a choice
# construct (except the last). It primes the history
# stack so that reverse execution will hand control to
# the given label, which the meta-compiler will arrange to
# be the following choice in the choice construct.
   hpush4 %esi %edi $\label $choice_r
.endm
```

The values pushed onto the history stack by the choice prefix code are the parameter and frame stack pointers, the label of the following choice, and the address of the code fragment `choice_r`. This primes the return stack so that backtracking to this point will pass control to the code fragment `choice_r`. That code fragment must restore the parameter stack and frame pointer and re-enter forward execution at the following choice.

```
choice_r: #reverse execution code for bounded choice
# pre: continuation address and saved parameter stack
#pointer are on the history stack.
   pop %eax #next choice addr to %eax
   pop %edi #restore stack frame pointer
   pop hsp  #restore stack pointer
   xchg hsp,%esp #set stacks for forward execution
   jmp *%eax #jump to next choice
```

Note that we restore the parameter stack pointer but not the parameter stack elements. We will comment further on this in the conclusions. For the moment let us turn our attention to the return stack, which we will have to restore in full. Consider the execution of the following test routine:

```
: TEST1 1[]2 ; ( assume 1[]2 is called, not in-lined )
: TEST2 TEST1 1[]2 .S MAGIC ;
```

Assuming the first choice in a choice construct is the first chosen, the first time $\boxed{\textbf{.S}}$ is reached the stack will contain 1 1. $\boxed{\textbf{MAGIC}}$ will force backtracking to be invoked. Forward execution will start again from the next choice within $\boxed{\textbf{1[]2}}$ and will then return. We need a mechanism to ensure, among other things, that the return stack pointer and top return stack value are restored to the state they were in when the previous choice was made, so that the return from $\boxed{\textbf{1[]2}}$ will be correctly performed. This is achieved by the following code which the meta compiler appends to every compiled definition containing a choice construct.

```
  mov %esp,%eax #return stack pointer
  mov (%esp),%edx #top of return stack
  hpush3 %edx %eax $has_choice_r
  ret
```

Thus we give `has_choice_r` the task of restoring the return stack pointer and top of stack. When `has_choice_r` subsequently runs, it can restore the return tos location immediately, but the return stack pointer must, for the moment, be saved in `hsp`, recalling that it will ultimately be restored when the operation `xchg hsp,%esp` is executed in `choice_r`.

```
#has_choice_r restores the return stack pointer and
#top return stack element. It is deposited on hstack
#just before exit from any secondary with the
#"has_choice" attribute
#pre: top of hstack is old return stack pointer value
#     next of hstack is old top element
has_choice_r:
   pop %eax #the old rsp
   pop %edx #the old tos
   mov %edx,(%eax) #restore old tos location
   mov %eax,hsp #restore the old stack pointer
```

Finally we need to consider what happens when backtracking returns to the first occurrence of $\boxed{\textbf{1[]2}}$, i.e the one which is invoked by a call from within $\boxed{\textbf{TEST1}}$. When this occurs, the return address for $\boxed{\textbf{TEST1}}$ must be restored. The mechanism used is similar to the one just described, except that we must not alter the return stack pointer. That will be restored as described above. We designate $\boxed{\textbf{TEST1}}$ as a word which *inherits choice*. The set of such words is defined recursively as the words which invoke a word which has choice, together with the words which invoke a word which inherits choice. The following code is appended to words which inherit choice:

```
  mov %esp,%eax #return stack pointer
  mov (%esp),%edx #top of return stack
  hpush3 %edx %eax $inherits\_choice\_r
  ret
```

and the `inherits_choice_r` code fragment is:

```
#inherits_choice\_r restores the return addr slot of
#the operation that deposited it on the hstack. It is
#deposited on hstack just before leaving any word with
#inherited_choice" attribute.
#pre: top of hstack is the addr within the return stack
#     where the retn addr for the operation that
#     deposited inherits_choice_r was held.
#     next of hstack is the return address itself.
inherits_choice_r:
  pop %eax # return stack slot
  pop %edx # return address
  mov %edx,(%eax) # put it back
  ret
```

9 Conclusions and Future Work

By including some degree of reversibility in the execution mechanism of a virtual machine we can implement choice and guard constructs, which allow unobtrusive backtracking with a procedural rather than declarative style.

We are currently adding sets to the language. At the same time we are developing the formal semantics for a construct that provides the set all of possible answers that can be generated by a non-deterministic program.

In the current work we have sought to maximise efficiency by taking careful account of the underlying physical machine architecture. However the mechanisms described should be readily adaptable to other Forth implementations.

References

[1] Jean-Raymond Abrial. *The B Book.* Cambridge University Press, 1996.

[2] H G Baker. The Thermodynamics of Garbage Collection. In Y Bekkers and Cohen J, editors, *Memory Management: Proc IWMM'92*, number 637 in Lecture Notes in Computer Science, 1992. See ftp://ftp.netcom.com/pub/hb/hbaker/ReverseGC.html.

[3] G Charlton. FOSM a Forth string matcher. In *EuroFORML9, Marianbad*, 1991.

[4] Intel Corporation. *Intel Architecture Optimization Reference Manual.* Intel, 1999.

[5] Intel Corporation. *Desktop Performance and Optimization for Intel Pentium 4 Processor.* Available from www.intel.com, 2001.

[6] Intel Corporation. *Intel Pentium 4 and Intel Xeon Processor Optimization.* Intel developer.intel.com, 2002.

[7] M L Gassenenko. Formalisation of return address manipulations and control transfers. In *EuroForth, St Petersburg*, 1996.

[8] B R Rodrigues. Pattern forth. Master's thesis, Bradley Univ, USA, 1989. This dissertation and other articles by the author are available at www.zetetics.com/bj/papers.

[9] W J Stoddart. A Virtual Machine Architecture for Constraint Based Programming. In P J Knaggs, editor, *16th EuroForth Conference, ISBN 9525310 x x*, 2000.

[10] W J Stoddart. An Execution Architecture for B-GSL. In Bowen J and Dunne S E, editors, *ZB2000*, Lecture Notes in Computer Science, 2000.

[11] W J Stoddart and F Zeyda. Implementing sets for reversible computation. In A ERTL, editor, *18th EuroForth, Technical University of Vienna*, 2002.