A Virtual Machine Architecture for Constraint Based Programming

Bill Stoddart

October 25, 2000

Abstract

We present a Forth style virtual machine architecture designed to provide for constriant based programming. We add $\boxed{\textbf{?CONTINUE}}$ and $\boxed{\textbf{CHOICE}}$ commands to allow for checking constraints and making tentative choice. A choice which is later seen to be incompatible with a constraint provokes backtracking, which is implemented by reversible execution of the virtual machine.

keywords: Forth, Virtual Machines, Constraint Based Programming, Reversible Computation.

1 Introduction

In Forth we can use an $\boxed{\textbf{IF}}$ construct to choose which of two sequences of operations to execute. The choice made depends on the value of a flag taken from the stack, and the programmer has to arrange the value of this flag so that the program will make the correct choice. In some cases however, the information needed to make the right choice might not be readily available, and it might be useful to leave the program free to make a tentative choice "for itself" (so to speak), with the possibility of later revising this choice if it turns out to be inappropriate. We call this form of choice "non-deterministic".

Consider how this technique could be used to provide predictive character input for a WAP phone, where each key represents up to three letters. To enter the word "love" we use the keys *jkl*, *mno vwx def*. When *jkl* is pressed at the start of a new word, the input routine makes a tentative choice of 'j' as the character entered. This is perfectly feasible (for the moment) since there are dictionay entries that begin with 'j'. When the second key is pressed it tries 'm' as the letter, then tests whether the dictionary contains any words that begin "jm". This choice is infeasible, since there are no such entries. The program backtracks and makes a different choice: "jn". Still infeasible. Another backtrack produces "jo" which is feasible. Carrying on in this way we might arrive at "jove". Now the user must press a button to provoke further backtracking to obtain "love".

The problems that can usefully be tackled with constraint based programming include timetabling, route planning and resource management.

The backtracking technique we propose is based on reversible execution. We provide three modes of execution. Normal, Conservative, in which any lost information is saved on a history stack, and Reverse, which runs back through a previous conservative execution and undoes its effect.

We extend Forth's repertoire of basic commands with a new guarded continuation command $\boxed{\textbf{?CONTINUE}}$. In conservative mode it removes a value from the stack, and if this is non zero allows execution to continue. If the value is zero it reverses the direction of execution.

This is used in conjunction with a new choice construct, which could have the form:

```
CHOICE <word1> <word2> ...  <wordn> ENDCHOICE
```

At run time the possible continuation addresses are pushed onto the history stack, and then the top address is selected for execution. If execution returns to this point in reverse, we choose another continuation and run forwards again. If there are no more continuation addresses we continue execution backwards to the previous CHOICE construct.

If execution is returned to the beginning of the program the constraints imposed were such that no feasible execution path could be found. The whole program is infeasible, and results in a "ko" message rather than "ok".

Execution becomes a search for a feasible path through the program. Tests that provide the flags used by the $\boxed{\textbf{?CONTINUE}}$ commands in out program express the constraints a potential solution must satisfy.

2 Virtual Machine Architecture, Outline

As a program runs, certain commands discard information whilst others conserve it. $\boxed{+}$ loses information since we cannot recover the values of its arguments x and y (say) from its result. To run it in information conserving mode we must save one of the values, say y, on the history stack.

Then, when we re-encounter the same instance of $\boxed{+}$ when running in reverse mode we have $x + y$ on the parameter stack and y on the history stack. To perform the reverse computation we copy y from the history stack (so the parameter stack contains $x + y$ and y) perform a subtraction (the stack now contains x) and finally move y from the history stack to the parameter stack.

Each primitive compiled command of our virtual machine may invoke one of three fragments of machine code of the underlying physical machine, depending on the execution mode in which it is invoked. These are normal forward mode, N, in which information is not conserved, conservation forward mode C which saves discarded information on the history stack, and reverse mode R which runs backward through the compiled virtual machine code reversing the effect of each command.

3 Execution Modes for Choice Choice

There are three execution modes for choice: first feasible, simultaneous and random.

In first feasible mode, we attempt to execute each choice in turn, consuming the corresponding continuation address on the history stack as we do so. If a choice proves to be infeasible we try another. If no choices are feasible, the whole choice construct is infeasible, and sets the virtual machine into reverse execution mode so that it will backtrack to the previous non-deterministic choice. This provides a depth first search for a feasible path through a computation.

In simultaneous mode the computation is cloned into as many dopplegangers as there are choices. These run in parallel under a "termination pact". The first to terminate execution survives. The others die, either committing suicide if their continuation proves infeasible, or killed by a signal from the terminating clone. This provides a breadth first search for a feasible path through a computation.

In random mode a pseudo-random number generator is used to select a choice, and a programmer may weight the probability of each choice.

4 Virtual Machine Implementation Details

One requirement of our virtual machine is the ability to interpret the same sequence of virtual machine instructions in three different modes, and this prompted us to choose a modified form of "Indirect Threaded Code" which supports Normal, Conservative and Reverse modes of execution through the use of three "code fields" for each word. These point to the code for normal execution, conservative forward execution and backward execution of the operation.

For each primitive definition, these three code fields contain pointers to sections of machine code that directly implement the normal, forward and reverse forms of the operation. Each of these sections of machine code has its own version of *next*, with the reverse code version causing execution to thread backwards through the threaded code.

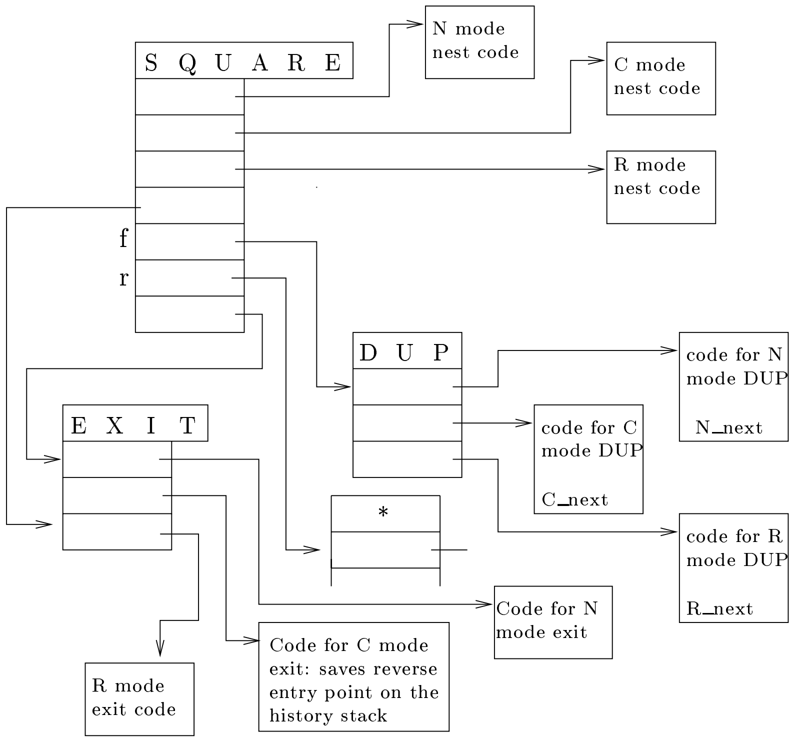
For a high level definition the three code fields contain pointers to sections of machine code that nest into the definition. The reverse version of *nest* must commence execution of the high level definition from the point that definition previously exited. The conservative code for $\boxed{\textbf{EXIT}}$ will have left this address on the history stack when the definition last executed in a forward direction.

When running backwards through a high level definition we reverse the effect of each virtual machine instruction until we reach the forward entry point of the definition, but what then? Running backwards into the definition's code field pointers has no meaning. We need to insert an extra virtual machine instruction, which will perform a reverse exit from the definition and go back to the calling location. But where was this location? It can be found on what is generally called the "return stack", but which in the case of reverse execution is more appropriately thought of as the "came from" stack.

Fig. 1 shows the threaded code organisation of the virtual machine.

4.1 Reversing Branch Instructions

Fig. 2 illustrates the use of a forward branch instruction in the case of a "normal" threaded code machine (on the left) with our virtual machine version which supports reversible execution on the right. First consider forward execution of the normal code. After executing OP1 execution encounters a branch instruction. We assume this to be a conditional branch with some unspecified test on machine state deciding whether the branch will be taken. If taken, the branch is said to be "active", and execution continues at the location indicated by the



N mode nest code
S Q U A R E
C mode nest code
R mode nest code
f
r
D U P
code for N mode DUP N_next
E X I T
code for C mode DUP C_next
*
code for R mode DUP R_next
Code for N mode exit
Code for C mode exit: saves reverse entry point on the history stack
R mode exit code

Figure 1: Threaded Code Organisation for the Abstract Command Language Architecture

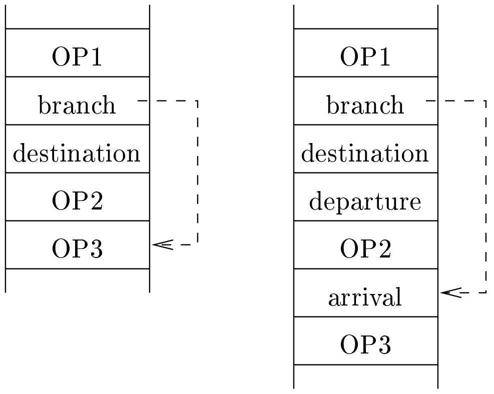


Figure 2: Normal (left) and Reversible Instruction Sequences Containing a Forward Branch.

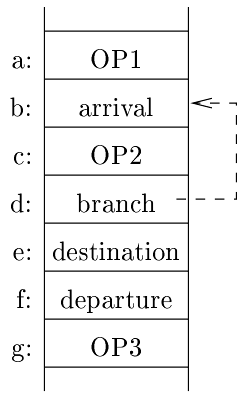


Figure 3: A Reversible Backward Branch

destination field of the branch; that is with OP3. If the branch is not taken, OP2 is executed followed by OP3.

Now consider an attempt at reverse execution of this code. There are basically two problems. Firstly, after reverse execution has undone the effect of OP3 it will move on to OP2. This is incorrect because we do not know that OP2 was ever executed. And secondly, after OP2, reverse execution will encounter the destination field of the branch, and try to interpret this as an instruction.

We deal with the first problem by inserting at the destination point of all jumps an additional instruction called *arrival*. It works in conjunction with a "branch active" flag which forms part of the virtual machine state. This flag is set whenever a branch occurs, and reset by *arrival*. Its function is to tell *arrival* whether it received control by a local or remote arrival. For example if *arrival* in fig. 2 is executed immediately after OP2 this constitutes a local arrival. When an active branch occurs, the executing branch instruction pushes the "from" address of the branch onto the history stack.[1] If the branch is inactive (execution of branch instruction but no branch) no history is recorded. Thus when *arrival* executes there are two possible conditions: either the branch active flag is set and the "from address" of the branch is on the history stack, or, the branch active flag is reset and there is no from address on the history stack. In the second case *arrival* saves the address of the previous instruction on the history stack, so that reverse execution of *arrival* can always find its backwards continuation address on the history stack:

The second problem has a simple solution. We insert a virtual machine instruction *departure* after the destination field of any branch instruction. This allows reverse execution to step over this field. The *departure* instruction is never executed in forward mode, as the branch instruction just steps over it.

In fig. 3 we see these ideas applied to a backward branch. The letters a, b, c.. are used to label the locations of each virtual machine instruction so we can represent some forward and reverse traces.

Consider an instance of execution in which the branch is first active and then inactive, so that OP1, OP2, OP3 are performed in the sequence:

[1]More specifically it pushes the address of the last instruction executed before the branch, which is OP1 in this case.

OP1, OP2, OP2, OP3

Assuming an empty history stack at the start of execution, we have the following trace of forward execution in C mode, where x is the history recorded by OP1, y_1 and y_2 the histories recorded by the two executions of OP2, and z is the history recorded by OP3.

Location	Operation	BA flag	History Stack
		false	$\langle\rangle$
a	OP1	false	x
b	arrive	false	$x ^\frown \langle a \rangle$
c	OP2	false	$x ^\frown \langle a \rangle ^\frown y_1$
d	branch	true	$x ^\frown \langle a \rangle ^\frown y_1 ^\frown \langle c \rangle$
b	arrive	false	$x ^\frown \langle a \rangle ^\frown y_1 ^\frown \langle c \rangle$
c	OP2	false	$x ^\frown \langle a \rangle ^\frown y_1 ^\frown \langle c \rangle ^\frown y_2$
d	branch	false	$x ^\frown \langle a \rangle ^\frown y_1 ^\frown \langle c \rangle ^\frown y_2$
g	OP3	false	$x ^\frown \langle a \rangle ^\frown y_1 ^\frown \langle c \rangle ^\frown y_2 ^\frown z$

For the reverse trace the branch active flag is not needed. All jumps in the reverse execution sequence are handled by the reverse execution of *arrival* which finds its continuation location on the history stack:

Location	Operation	History stack
		$x ^\frown \langle a \rangle ^\frown y_1 ^\frown \langle c \rangle ^\frown y_2 ^\frown z$
g:	OP3	$x ^\frown \langle a \rangle ^\frown y_1 ^\frown \langle c \rangle ^\frown y_2$
f:	depart	$x ^\frown \langle a \rangle ^\frown y_1 ^\frown \langle c \rangle ^\frown y_2$
c:	OP2	$x ^\frown \langle a \rangle ^\frown y_1 ^\frown \langle c \rangle$
b:	arrival	$x ^\frown \langle a \rangle ^\frown y_1$
c:	OP2	$x ^\frown \langle a \rangle$
b:	arrival	x
a:	OP1	empty

We finish our discussion of branch instructions with a note on normal (N mode) forward execution. The data in the branch destination field compiled immediately following each branch instruction is interpreted slightly differently in N mode and C mode. N mode branch execution jumps to the instruction following the instruction jumped to by C mode execution of the same instruction. This means *arrival* is never executed in N mode and the mechanism for reverse execution of branches imposes little or no run time penalty on N mode code.

4.2 The Multi-Tasker

We have said that for the execution of certain constructs, execution must be cloned. The virtual machine provides multi-tasking with a classical Forth syle non-preemptive round robin scheduler. Creating a clone requires the allocation of memory for its state space, the copying of the existing state space into that of the clone, and linking the clone into the round robin as a task. Killing a task requires unlinking it from the round robin and de-allocating its memory space. Duplicating the compiled program code is not required as this is re-entrant.

5 Optimisation

Ways of optimising reversible code are not discussed here, but are supported by some special virtual machine instructions. A simple example is where we have a sequnce of operations which use the stack to calculate a value, and then we store this value in a variable. To undo the effect of this we only need to restore the old value of the variable; there is no point in reverse executing every instruction of the sequence. Possible solutions to such problems are left to the readers imagination.

6 Implementation

Our implementation environment is Linux running on the i386 architecture, and our nucleus is coded in i386 gnu assembler. We have written a pre-processor which adds macros to this assembler. These deal with word headers, control structures and high level definitions. These macros together with the gnu assembler form a crude and limited meta compiler sufficient to create the nucleus of a reversible Forth.

7 Other Work

An extensive bibliography of work on reversible computing is available at:
ftp://ftp.netcom.com/pub/hb/hbaker/ReverseGC.html

In our own work we use an associated high level language, B-GSL,[1] which has a simple formal semantics which includes non-deterministic choice and backtracking. Direct use of a reversible Forth will let us explore problem solving techniques that are beyond the current scope of GSL and suggest directions for its extension.

Acknowledgements

Thanks to Andrew Haley of Cygnus Solutions, Cambridge, and Gordon Charlton of the Forth Interest Group for sharing their knowledge of reversible computation.

References

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