A Reversible Virtual Machine

Bill Stoddart, Angel Robert Lynas
University of Teesside

Frank Zeyda, University of York

March 27, 2009

We describe a reversible stack based virtual machine designed as an execution platform for a sequential programming language used in a formal development environment. We revoke Dijkstra's "Law of the Excluded Miracle" to obtain a formal description of backtracking through the use of naked guarded commands and non-deterministic choice, with execution support provided by reversibility. Other constructs supported by the machine include..

Finite sets and relations of arbitrary complexity.

facilities for the collection of all results of a search

probabilistic choice

a "cut" facility

local variables and nested scopes

lambda expressions and closures

integer and floating point arithmetic

Structure of this Talk

Guarded Command Language, Reversibility and Backtracking.

"Semantic" vs "Logical" Reversibility.

Mechanisms of the Reversible Virtual Machine.

Guarded command language (GCL) is a form of programming language with a clean formal semantics.

pGCL (probabilistic GCL) was used by Paolo Zuliani in his investigation of logical reversibility (IBM Jnl of R&D 2001).

In our approach we extract some additional meaning from GCL that is specific to reversible computation.

Predicate transformer semantics of GCL

Let S be a program and Q a predicate over machine states.

Let $[S]Q$ be the condition that must hold before S is executed in order to guarantee Q holds after S is executed.

Example: $[x := x + 1]x = 5$ is the condition that must hold before $x := x + 1$ is executed to ensure $x = 5$ holds afterwards.

This condition is:

Predicate transformer semantics of GCL

Let S be a program operating and Q a predicate over machine states.

Let $[S]Q$ be the condition that must hold before S is executed in order to guarantee Q holds after S is executed.

Example: $[x := x + 1]x = 5$ is the condition that must hold before $x := x + 1$ is executed to ensure $x = 5$ holds afterwards.

This condition is: $x = 4$

The predicate transformer rule for the short conditional:

$$[\mathit{if}\ g\ \mathit{then}\ S\ \mathit{else}\ T\ \mathit{end}]Q$$

$$\equiv$$

$$(g \Rightarrow [S]Q)\ \wedge\ (\neg\, g \Rightarrow [T]Q)$$

Suggests a decomposition of the conditional into two more primitive constructs: "choice" and "guard"

Allowing us to write the short conditional as a "guarded choice":

$$g \Longrightarrow S \;[\!]\; \neg\, g \Longrightarrow T$$

Where the choice and guard constructs have these beautiful rules, which replace program connectives for choice and guard with the logical connectives for conjunction and implication:

$$[S \;[\!]\; T]Q \;\equiv\; [S]Q \wedge [T]Q \quad \text{(choice)}$$

$$[g \Longrightarrow S]Q \;\equiv\; g \Rightarrow [S]Q \quad \text{(guard)}$$

The constructs $g \Longrightarrow S$ and $\neg\, g \Longrightarrow T$ represents the "meaning" of each half of a conditional. They have no sensible individual operational meaning in classical programming. Indeed they are required to produce "miraculous" results. Suppose g is false, then

$$[g \Longrightarrow S]Q \quad \equiv \quad \mathit{false} \Rightarrow [S]Q \quad \equiv \quad \mathit{true}$$

i.e. when g is false, $g \Longrightarrow S$ will produce any result we care to propose.

This is exactly what is required from the part of the conditional which is not chosen, since analysis of the other part then gives the substantial result, which will be unchanged by being combined, via logical and, with the result *true* from the part not chosen.

Dijkstra excluded “naked” guarded commands from the world of programming constructs by proclaiming the “Law of the Excluded Miracle”.

In a reversible guarded command language however, we can revoke the Law of the Excluded Miracle, and give the following interpretation to the naked guarded command $g \Longrightarrow S$, based on the predicate transformer semantics given previously:

If g is true, execute S and continue ahead.

If g is false, engage reverse gear.

In our reversible guarded command language we also allow non-deterministic choice constructs. This needs some explaining since it could be argued that a reversible computation has to be deterministic in both directions. In fact the apparent non-determinism of our language stems from providing a semantics at a more abstract level than an execution mechanism.

Given a statement of the form

$S \,[\!]\, T$

there will be some supporting deterministic mechanism to decide which choice is made.

Backtracking

Consider the sequence

$x := 1 \,[]\, x := 2 \,;\;\; x = 2 \Longrightarrow \mathit{skip}$

Suppose $x := 1$ is chosen, the guard $x = 2$ will then be false and execution will reverse. The other choice, $x := 2$ is than made, and this time the guard is true and execution continues ahead. This mechanism does not require any supplementary data erasure due to the choice construct.

Semantics and Executable Constructs for Reversibility

$S \diamond E$

represents the execution of a program S, the recording of a result as given by the value of expression E, and a reversal of execution which restores the original state (apart from leaving the value of E on an evaluation stack and, if E is a reference, leaving the referenced structure in memory).

Where S has non-deterministic choices that can lead to more than a single result for E

$\{S \diamond E\}$ is the set of all such results

Example

$$\{x := 1 \,[]\, x := 2 \;\diamond\; 10 * x\} \;=\; \{10, 20\}$$

We will presently see a more elaborate example, together with a translation into the postfix language of the RVM.

Semantic Reversibility

Paolo Zuliani converted pGCL to a reversible language. To do this he added extra state, in the form of a history stack, to record information that would otherwise be lost in forward execution.

For each computational step S he proposes an augmented, reversible versions S_r and its inverse S_i.

Using this technique for assignment he proposes:

S	Reversible Op S_r	Inverse Op S_i
$v := e$	$push\ v;\ \ v := e$	$pop\ v$

This is similar to our approach, though we work at the level of virtual machine ops. However, we note that a short cut has been taken in that S_r *still includes the irreversible step* $v := e$.

We allow such short cuts for reasons of convenience, and because we can argue (as Yokoyama, Axelson and Glück have done in a study of "reversible updates") that updates *can in principle* be made in a reversible fashion. We give our analysis in terms of assignments.

We have some reversible assignment statements to call upon, for example those of the form $x := x + e$, where x does not occur free in e. Such a statement has an inverse $x := x - e$.

We note also that exchanging values in two locations is reversible.

The role of the history stack will be taken by an integer array, h, whose elements are assumed to be initialised to zero, and whose top element is $h(i-1)$

We write the exchange of values in x and $h(i)$ via the multiple assignment:

$x, h(i) := h(i), x.$

A reversible transformation of $x := e$ can be written as:

assignment	$h(i-1)$	$h(i)$	x
	$?$	0	x_0
$h(i) := h(i) + e$	$?$	e	x_0
$x, h(i) := h(i), x$	$?$	x_0	e
$i := i + 1$	0	x_0	e

Simulation of Logical Reversibility is fastidious due to the need to manage data which is of no semantic consequence, such as the contents of unused stack locations. We opt instead for a virtual machine which provides what we term “semantic reversibility”.

The Reversible Virtual Machine

The RVM provides a Forth stack based virtual machine. An interpreter allows operations to be entered from a console or read from a file. A simple extensible compiler allows new operations to be defined in terms of existing ones.

The machine is currently implemented on the i386 platform.

Examples of an interpreted interaction in Forth:

```
1 2 + . <enter> 3 ok

3 4 < . <enter> -1 ok

1 2 3 <enter> ok...

+ + . <enter> 6 ok

10 VALUE X <enter> ok
X 5 * to X <enter> ok
X . <enter> 50 ok
```

An new command to perform a greater than or equal test can be defined in terms of a less than test and an equality test as:

```
: >= ( n1 n2 -- f ) < NOT ;
```

Machine organisation during forward execution:

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

Memory changing commands use the history stack to preserve information during execution.

Code for a reversible memory store. The values pushed to the history stack are the address to be overwritten, its current contents, and the address of the code that will restore the original value on reverse execution.

```
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) #pop x into addr
   xchg %esp,%esi
   ret
ENDCODE
```

Machine organisation during reverse execution takes the following form:

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

A switch to reverse computation is performed by --> which removes a flag from the stack, continues forward if the flag is true and reverses execution if the flag is false. To reverse execution we copy the history stack pointer to the i386 stack pointer and return into the most recently deposited reverse op. The EXPLORE flag controls backtracking.

```
CODE --> ( f --   "guards")
  lodsl # %eax = f
  if %eax = $0; # reverse
    movl $-1,_EXPLORE(%ebp)
    mov hsp,%esp #point %esp at hstack
    ret #start reverse operations
  endif; noop
ENDCODE MUST-IN-LINE
```

Reverse operations find their parameters on the stack, and after consuming them they return into the following reverse operation. Note that they are never "called", but only returned to, a method of organising code which we refer to as "return threading". 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 !_.

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

An example abstract command language program and its equivalent RVM code

The following assigns to y the set of all positions at which the sequence s contains the value x.

$$y := \{i :\in 1..card(s);\ s(i) = x \Longrightarrow skip \diamond i\}$$

The code, with an RVM translation:

$$y := \{i :\in 1..card(s);\ s(i) = x \Longrightarrow skip \diamond i\}$$

```
INT { <RUN
  1 s CARD .. ( push set of possible indices )
  CHOICE ( make a choice from the set )
  to i ( assign it to i )
  s i APPLY ( push value s(i) )
  x = --> ( reverse unless the value = x )
  i ( otherwise add i to the set of results )
RUN> } to y
```

Thankyou