Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 253 (2010) 33–56
www.elsevier.com/locate/entcs

A Virtual Machine for Supporting Reversible Probabilistic Guarded Command Languages

Bill Stoddart, Robert Lynas

School of Computing, University of Teesside, Middlesbrough, UK

Frank Zeyda

High Integrity Systems Engineering Group, Department of Computer Science, University of York, UK

Abstract

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 an operational interpretation of the interaction between guards and choice provided by reversibility. Other constructs supported by the machine provide for the collection of all results of a search, a semantically clean "cut" which terminates a search when the accumulated results satisfy some given criteria, and forms of probabilistic choice, which we distinguish from non-deterministic choice. The paper includes a number of example programs.

Keywords: Reversible Computing, Virtual Machine, Formal Semantics, Probability, Backtracking, Forth

1 Introduction

Much interest in reversible computing has centred on its potential to reduce the minimum power requirements of a computation through an understanding of the importance of Landauer erasure [10]. Through the contributions of Bennett [5,6], Feynman [8] and others it has rekindled "mechanical" intuitions which most of us had abandoned (despite much talk of computing machinery) in the face of devices which appear to mimic purely mental functions; perhaps we could say it has helped to make us proper computer scientists, rather than just computing scientists.

At the same time there has been relatively little interest in exploiting reversibility to enhance programming, though a notable exception has been H. Baker [3,4] who

[1] We acknowledge the interesting and helpful remarks of the referees and of participants at the RC2009 workshop.

doi:10.1016/j.entcs.2010.02.005

has insisted that the intuitions provided by a reversible view of computing should inform our approach to programming, and has explored the use of reversibility in garbage collection.

For some time the authors have been investigating the effect on the formal development of programs of considering computation as an essentially reversible process. We have found that reversibility can simplify program semantics and suggest new programming structures and ways of thinking about problems. In particular we are interested in using reversibility to provide backtracking. These ideas have been discussed in our previous papers: in *The refinement of reversible computations* [18] we describe the development by refinement and proof of a Knight's Tour program using the B Method [1]. In *A Prospective-value semantics* [17] we give a new form of semantics for describing reversible computations. In the thesis [16] we investigate extending the B method of formal software development to develop reversible programs. In *A design based model of reversible computation* [13] we formulate our semantics of reversible computations within the framework of Hoare and He's approach to unifying theories of programming [9], and in *A unification of probabilistic choice* [12] we add probabilistic choice to our programs and link the probabilistic and non-probabilistic versions of our theory with a Galois connection.

The contribution of the present paper is to describe the reversible virtual machine (RVM) which provides execution support for our programming theories, with reverse execution being implemented in an efficient and novel way by "return threading". In addition to direct support for reversibility, the RVM provides other high level language support within a reversible context, including scoped local variables and stack frames, novel programming structures to express backtracking, lambda abstraction and closures, a general set package, and floating point (as well as integer) arithmetic. Its implementation is based on compilation of a stack based intermediate language (Forth) to native code, and includes the more straightforward forms of optimisation, such as code in-lining and peephole optimisation.

Our general approach is to take a non-reversible language and make its operations reversible by the addition of additional state, in the form of a "history stack". This is also the approach followed in the analysis of logical reversibility given by P Zuliani in [19]. The additional state which provides reversibility need not be seen during semantic analysis of individual commands. We can contrast this with investigations of reversible languages based exclusively on explicitly reversible instructions, as illustrated by the Janus language of C Lutz (1986) and more recently developed by T Yokoyama, H B Axelson and R Glück [2,15,14].

The rest of the paper is organised as follows. In Section 2 we consider how backtracking can be added to a guarded command language and describe new programming structures we require the RVM to support. In Section 3 we consider design perspectives for the RVM, discussing whether to aim for full logical reversibility or merely "semantic reversibility". We consider carefully how reversible assignment could be constructed in a stepwise reversible manner. In Section 4 we give a sufficient introduction to Forth and describe the internal organisation of the reversible machine and the basic mechanisms for implementing guards and choice. In Sec-

tion 5 we describe the virtual machine representation of programming structures designed to capture the results of reversible computations. In Section 6 we describe an example program that uses the structures we have introduced, in section 7 we discuss probabilistic programming, and in section 8 we conclude.

Source program listings for the RVM, example programs and a manual (110 pages) are available from `http://www.scm.tees.ac.uk/formalmethods`

2 Backtracking and guarded commands

A small change in a guarded command language is sufficient to provide a basic semantics of backtracking: one has only to liberalise the syntax of such languages to allow "stand alone" guarded command and choice constructs.

A selection statement in such a language might take the form:

$$\text{if } g_1 \Longrightarrow S_1 \sqcap g_2 \Longrightarrow S_2 \sqcap \ldots \text{ fi}$$

where the g_i are condition tests or "guards", the S_i are program actions, and the symbol $\sqcap$ represents choice. The informal semantics of the selection construct is that it may perform any action S_i for which the corresponding guard g_i is true. If no guard is true, the whole construct acts as a *skip*, i.e. has no effect.

To formally analyse such a construct we can use a predicate transformer semantics. We write $wp(S, Q)$, where S is a program and Q a predicate on program states, for the condition required to hold before S is executed (the "weakest precondition") to guarantee that Q will hold as a post condition. We have the following rules, which express the effect of the programming connectives "guard" and "choice" in terms of the logical connectives "implies" and "logical and":

$$wp(g \Longrightarrow S, Q) \;=\; g \Rightarrow wp(S, Q)$$
$$wp(S \sqcap T, Q) \;=\; wp(S, Q) \wedge wp(T, Q)$$

To analyse the selection constructs we remove the if ... fi brackets, add a final choice to perform no action if the complement of the guards is true, and then apply the rules for choice and guard. A command $g_i \Longrightarrow S_i$ whose guard is false is not available for selection and must have no effect on the answer. Since the answer is produced by the conjunction of answers provided by each separate guarded command, a non-selectable command must provide a result of *true* in order to have no effect (we recall here that in logic $a \wedge true \Leftrightarrow a$).

If we allow a naked guarded command to appear in a program, its logical analysis will tell us it is capable of ensuring any post condition when the guard is false. This is too good to be true, so it is usually considered that such a command only becomes meaningful during program analysis, where its contribution is combined with that of other guarded commands. To emphasise the nature of stand alone guarded commands their behaviour is described as "miraculous". The most extreme case of a command that cannot be selected when present in a choice can be defined as $false \Longrightarrow skip$, and is referred to as *Magic*.

It is, however, possible to give an interpretation of choice and guarded commands in terms of reversible computation. Furthermore, this interpretation requires no change in program semantics, although some care is needed when interpreting a proof that S can establish post-condition Q, as we must check that this is not due to miraculous behaviour.

In fact we prefer an alternative semantic approach, "Prospective Value Semantics" [17], which is perfectly adapted to reversible computing and suggests some useful new programming structures. We use $S \diamond E$ to represent the value(s) expression E might take after executing program S. In operational terms it can be thought of as representing the execution of S, recording of the value of expression E, and the reversal of execution to leave any variables in their original state. Where S includes choice constructs which allow it to produce more than one value of E, we use $\{S \diamond E\}$ to give the set of values that might be obtained.

As an example consider the effect of the following:

$$x := 0;\ y := \{x := 1 \sqcap x := 5 \sqcap x := 8 \diamond x + 10\};\ \ldots$$

This leaves $y = \{11, 15, 18\}$ and $x = 0$. The curly brackets are set brackets. The operational interpretation of $\{x := 1 \sqcap x := 5 \sqcap x := 8 \diamond x + 10\}$ is that it opens a new set construct, chooses to assign either 1, 5 or 8 to x, and deposits the resulting value of $x + 10$ into the set. Execution then reverses back to the choice, restoring the value of x to zero, and another choice is made and corresponding value deposited. When no more choices are available the set construct is closed and execution continues ahead. [2]

Now recalling Zuliani's use of additional state to render a guarded command language reversible, [19], we ask whether there is any additional operation required to implement backtracking and that requires Landauer erasure. Reversing back to a choice does not exactly restore the internal state, since the machine must keep track of which will be its next choice. For n possible choices this can be done by counting modulo n, which is a reversible process. We therefore conclude that this form of backtracking may be implemented with reversible operations.

We can illustrate the use of the $S \diamond E$ form with the following example, which sets y to the set of all locations at which the array a, of size (cardinality) $card(a)$, contains a value equal to x:

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

Here $1..card(a)$ is the set of possible index values for a, and $i :\in 1..card(a)$ chooses one of the index values and moves it from the set $1..card(a)$ into i. If the choice gives a value such that the guard of the following command is true, the value of i is recorded in the set and execution reverses back to the choice command. Otherwise the failed guard will cause execution to reverse without adding i to the set. This is repeated until the set of possible indices is exhausted.

[2] An alternative operational interpretation of this semantics is that choices may be executed in parallel, but this does not reflect the current organisation of the RVM

Each time execution arrives back at the choice we have a situation where the system state has not been exactly restored, the difference being that the most recently chosen element has been removed from the set. However, this does not affect the reversibility of our construct, because removing an element from a set is a reversible operation. The reverse operation is to put the element back into the set. That is, in general, if $x \in X$ then $X \setminus \{x\};\ X := X \cup \{x\}$ is equal to *skip* (the operation which leaves everything unchanged).

3 Design perspectives for the reversible virtual machine

The RVM is an implementation of the postfix language Forth augmented with a history stack. During forward execution the history stack accepts data required to restore the state of the system during reverse execution.

Within this design framework one possible approach would be to construct the system entirely from reversible components. This would allow constructs to be shown to be fully reversible by implementation.

Let us see what would be needed to achieve this for an assignment statement. Zuliani's technique for making an irreversible language reversible is to add some extra state, in the form of a boolean variable b (not used in our particular illustration) and a history stack, and to transform each operation in the language into a reversible operation which uses the history stack to preserve any information that would otherwise be lost when the operation is executed. A proof that the transformation yields reversible operations is given by providing the inverse of each transformed operation. For each operation S Zuliani constructs a transformed operation S_r with an inverse S_i such that $S_r\ ;\ S_i = \mathit{skip}$ and such that the effects of S and S_r on the original state space are the same. For the assignment statement the technique is applied as follows:

S	Reversible Operation S_r	Inverse Operation S_i
$v := e$	$\mathit{push}\ v\ ;\ v := e$	$\mathit{pop}\ v$

Whilst this analysis does what is claimed, the description of S_r uses an irreversible step: $v := e$. We will also use such irreversible steps in our simulation of reversibility, but we give here an analysis of how purely reversible updates may be performed. Our analysis is in terms of assignment statements; a treatment of reversible updates in terms of injective functions is given by Yokoyama et al in [14].

To give a transformation consisting of the sequential composition of *reversible steps* we first note that we have some reversible assignment statements to call upon. Those of the form $x := x + e$, where x does not occur free in e have an inverse $x := x - e$.

We also note that the exchange of values in two locations is reversible. Such exchanges can be written using multiple assignments, which have the general form $x, y := E, F$.

The role of the history stack will be taken by an integer array, h, having a

large enough size, and with an array index i used as a stack pointer. h and i are fresh variables with respect to the original program. We assume the elements of h are initialised to zero and i is initialised to one (requiring an initial investment of energy). When $i > 1$, $h(i)$ is the top element of the history stack.

We can give a reversible transformation of $x := e$ as a sequence of reversible commands, as shown in the following trace:

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
$i := i + 1$	x_0	0	e

The reverse operation may be formed from the inverse operations of each of the above steps:

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

The above argument shows that memory updates can be done in a reversible manner, but at the cost of an increased complexity of operation. Having shown the possibility of reversible updates, we do not consider it necessary to pay the price of this added complexity when designing the RVM, and we aim for what we term "semantic reversibility" rather than logical reversibility. For semantic reversibility it is sufficient for a program to yield the same results as its logically reversible homologue. As far as the history stack is concerned that means we don't bother to initialise unused stack locations to zero (since the values held there have no influence on the results given by a program) and that we allow ourselves to use irreversible steps so long as their combined effect is equivalent to the combined effect of the reversible steps described above.

4 The RVM Forth Virtual Machine

The RVM is based on Forth [7]. Forth provides a stack based virtual machine, along with an interpreter which allows operations to be entered from a console or read from a file. A simple but extensible compiler allows new operations to be defined in terms of existing ones. An example of an interpreted interaction is:

```
3 4 < . <enter> -1 ok
```

Here, the Forth code `3 4 < .` is entered from a console. 3 and 4 are pushed onto the stack, the command `<` is a test which removes them and pushes a true flag (since $3 < 4$). The command `.` removes and prints the top stack value as an integer.

Where Forth commands occur in text we either display them as verbatim text, or, where this is not sufficiently clear, we display them within a box. This convention is used in Forth because the language has no syntactic limitations on what character strings can be used as names: all the punctuation symbols, for example, are standard Forth commands.

A new command to perform a greater than or equal test can be defined as:

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

This adds the command `>=` to the Forth "dictionary", where it has an equivalent status to pre-existing commands. The text within brackets is a comment which shows the stack effect of the new command. `:` and `;` are commands rather than syntax. `:` scans following text, adds the name "`>=`" to the Forth dictionary, and enters compile mode. `;` terminates the definition by compiling an exit command and switches back to interpret mode. Forth is a language of commands rather than syntax, though commands may impose some restrictions on what input Forth will accept. For example `:` could print an error if the system is not in interpret mode.

Forth commands have access both to the input stream, allowing them to parse following text, and the code space of the virtual machine, allowing them to perform compilation functions, illustrated here by the exit compiled by `;`.

Forth uses separate parameter and return stacks. The RVM adds a history stack to support reverse execution and includes reversible versions of all memory changing operations. As the above example shows, use of local variables is not obligatory, as parameters can be accessed directly from the stack. However, the RVM supports local variables and nested scopes, with stack frames held on the parameter stack.

4.1 Machine organisation during forward execution

RVM Forth is currently implemented on the Intel platform. 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 the following canonical form is taken at any transfer or branch during forward execution:

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

The virtual machine consists of a nucleus and a utilities layer. The nucleus definitions are written in either Forth or in our structured version of Gnu assembler. Compilation of the nucleus generates a monolithic Gnu assembler program, which can be linked with programs written in other languages, predominantly C. The nucleus provides an interactive Forth system which has few features (no control structures or variables even) but which can extend itself by incremental compilation, due to its ability to parse the input stream and compile code.

To illustrate the machine organisation at the lowest level we look at two examples of RVM-Forth "code" definitions. Both our examples work on integer data. + removes the top two elements of the stack, adds them, and pushes back the result. MIN removes two elements and replaces the smaller. We code such primitives in a structured adaptation of the GNU assembler. Glancing at the code below, the reader will notice a register exchange at the start and end of each of these operations. This is because Forth based virtual machines have separate return stack and parameter stack pointers, and the Intel platform provides only one fully featured stack pointer. By exchanging these registers, which the RVM uses as its parameter and return stack pointers, we can use the fully featured Intel stack pointer as the RVM parameter stack pointer within the body of the primitive definition.

```
CODE + ( n1 n2 -- n3)
   xchg %esp,%esi
   pop %eax
   pop %edx
   add %edx,%eax
   push %eax
   xchg %esp,%esi
   ret
ENDCODE
```

```
CODE MIN ( n1 n2 -- n3, n3 is min(n1,n2))
   xchg %esp,%esi
   pop %eax; pop %edx
   if %eax < %edx
     push %eax
   else
     push %edx
   endif
   xchg %esp,%esi
   ret
ENDCODE
```

When commands are composed in sequence, either a call to each command is compiled, or if the command is less that a certain length, say 16 bytes, it will be compiled in line with peephole optimisation. If in line code for + is followed by in-line code for MIN the optimiser will remove the following instructions at the join:

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

4.2 Reverse execution

The Standard Forth operation ! takes an address and a value from the stack and stores the value in the address. The RVM also has a reversible version of each memory changing command, named with an appended underscore. !_ performs the normal Forth store function, but also records on the history stack the address of the store, the previous value at that address, and the address of the reverse operation. We use the following macro when coding !_ :

```
.macro hpush3 rm1 rm2 rm3
# pushes rm1 rm2 rm3 to hstack, rm3 will be top
# note, rm signifies a register of memory location
   xchg hsp,%esp
   push \rm1
   push \rm2
   push \rm3
   xchg hsp,%esp
.endm
```

Now we give the code for the reversible memory store. Note that brackets in Gnu assembler denote indirection: `mov (%eax),%edx` moves the contents of the location pointed to by `%eax` into `%edx`.

```
CODE !_ ( x addr -- "store_")
   xchg %esp,%esi
   pop %eax # address for store
   mov (%eax),%edx #move current contents to %edx
   hpush3 %eax %edx $STORE_r #prime history stack
   pop (%eax) #store top of stack in the given location
   xchg %esp,%esi
   ret
ENDCODE
```

The values pushed onto the history stack are the data required to restore the original memory contents and the address of the operation which will perform the restoration during 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 the Forth guard command: `-->` which removes a flag from the stack, continues ahead if the flag is true, and switches to reverse execution if the flag is false.

Reverse execution normally continues until a choice is encountered which has an unexplored alternative. If no such choice exists it continues until the history stack is completely cleared and gives a "ko" rather than an "ok" response.

To reverse execution we copy the history stack pointer to the i386 stack pointer and return into the most recently deposited reverse operation, as shown in the following definition:

```
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
```

The command `MUST-IN-LINE` tells the compiler that this 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.

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 of location to be restored
   mov %edx,(%eax) #restore old contents
   ret # return into the next reverse computation
```

4.3 Choice

The specification of the guarded command language choice construct does not detail which choice should be taken first, and the RVM provides various possibilities, including making a random choice. The most simple possibility is provided by a virtual machine choice construct that always tries choices in order, with the first choice being tried first.

In RVM-Forth, to use this form of choice to select between programs `A`, `B` .. `C`, we write `<CHOICE A [] B [] .. [] C CHOICE>`. As each choice is compiled it is prefixed by code that primes the history stack to conditionally switch back to forward execution and transfer control to the following choice. The condition for making the switch is that the system is not in reverse execution mode as the result

of a cut. We simplify the code in the following explanation to ignore this possibility and also omit code to restore elements of the stack. The full code is available for inspection in the RVM listings and is well commented.

In the simplified code shown here, we 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 as 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 Forth 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
```

This simplified code provides efficient backtracking when no cut is used and no special action to restore parameter stack elements is required. This latter is quite often the case, since the stack elements required when forward execution commences are often local variables whose values are restored by reverse assignments. In this case our observations of a Knight's Tour case study using exhaustive search have indicated a performance for the virtual machine an order of magnitude faster than compiled Gnu Prolog.

Now let us turn our attention to the return stack, considering in detail how its elements are restored. Consider the execution of the following test routine, where `.S` is an operation to display the contents of the parameter stack:

```
: 1[]2 <CHOICE 1 [] 2 CHOICE> ;
: 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 .S is reached the stack will contain 1 1. MAGIC will force backtracking to be invoked. Forward execution will start again from the next choice within the operation 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 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 top of stack 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 the operation 1[]2, i.e the one which is invoked by a call from within the operation TEST1. When this occurs, the return address for 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 TEST1 as an operation 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
```

4.4 Abandoning a search

We sometimes want to abandon a search and try again with a change in search heuristic. This requires a stronger form of reverse execution, which cannot be revoked by normal requests to resume forward execution. The RVM provides a set of three commands which allow this form of reverse execution to be invoked and which control its scope.

The RVM source distribution includes a Knight's Tour program which makes use of these techniques. The program employs a heuristic that will usually find a circular tour from any starting square in under 100 moves. However, even when using the heuristic it is possible to enter a large region of the search space which contains no solutions and takes a long time to escape from. An effective way to deal with this is to give up after, say 1000 provisional moves and start the search again with a different heuristic.

Giving up will mean irrevocably reversing the execution of the current heuristic, but also controlling the scope of this irrevocable reverse execution so that the next choice of heuristic can be tried. This is achieved with the following operations, which are used in the form: `<TRY S TRY>` where `S` may conditionally execute the operation `GIVE-UP`.

The operation `<TRY` limits the scope of irrevocable reverse execution. It does not resume forward execution, but rather changes the nature of reverse execution so that forward execution can be resumed further back, e.g. at a choice with an unexplored alternative.

The operation `GIVE-UP`, if executed by code between `<TRY` and `TRY>`, will invoke irrevocable reverse execution.

The operation `TRY>` cancels the effect the previous `<TRY`, which we no longer require if we have successfully reached this point.

This can be combined with the choice construct as in the following example, in which we suppose `TACTIC1` can invoke `GIVE-UP` to escape from its search and pass control to `TACTIC2`.

```
<CHOICE <TRY TACTIC1 TRY> [] TACTIC2 CHOICE>
```

5 New Control Structures for Reversible Computation

To collect garbage after a calculation the calculation has to be organised in three stages: first, forward execution, second, evaluate and copy an expression giving the result, and third, reversal of computation which collects all garbage generated by forward computation, frees all history stack locations, and restores any variable values to their original settings. Formally we represent this as the program structure $S \diamond E$ where S is the program and E the expression to be evaluated to yield the result, and we call $S \diamond E$ the *prospective value* for E after S.

5.1 Deterministic form

The postfix translation of $S \diamond E$ where S is deterministic and E an integer expression, is:

```
<RUN S E INT>
```

Where `S` is the postfix translation of S and `E` the postfix translation of E. E may alternatively be an expression representing (as a reference) a set, and ordered pair, or a string. Then we must use `SET`, `PAIR>` or `STRING>` in place of `INT>`. These variations are concerned with protecting referenced data from erasure during reverse execution by cloning it; different forms of data are cloned in different ways, and our implementation does not have a sufficient awareness of type in the above context for the correct form of cloning to be selected automatically.

5.2 Collecting the possible results of a non-deterministic computation

Where S is non-deterministic the term $S \diamond E$ may have no possible values or more than one possible value. In that case it must appear within set brackets for its meaning to be properly defined, and will yield the set of possible values E could take after executing S. Type information is required for the set construction, and we make use of this to have a single command `RUN>` to match `<RUN`. Where E has an integer value, $\{S \diamond E\}$ would translate into RVM-Forth as:

```
INT { <RUN S E RUN> }
```

where `INT` provides type information for the set construction. Type information is given by a postfix expression representing the type of the elements in the set. This is based on the given sets[3] `INT` and `STRING` and set constructors `POW` (power set) and `PROD` (set product). If E was a set of string to integer maplets, the translation of $S \diamond E$ to postfix would be:

```
STRING INT PROD POW { <RUN S E RUN> }
```

[3] Our type theory is based on that of J. R. Abrial, in which types are maximal sets. Our concrete representation of types, however, is built from on empty sets: INT and STRING are the empty sets of integers and strings respectively.

<RUN compiles code that will prime the history stack with an operation to resume forward execution beyond RUN>, RUN> adds the top stack element to the current set; if this element is a reference, the referenced data is cloned to protect it from garbage collection during reverse execution. } primes the history stack to perform garbage collection on the completed set on reverse execution.

We illustrate these structures with an assignment to y of the set of all positions at which a sequence s contains the integer value x.

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

This is a direct postfix translation of the assignment:
$y := \{i :\in 1..card(s);\ s(i) = x \Longrightarrow skip \diamond i\}$ discussed in Section 2.

6 A more Extensive Example: Pseudo Arithmetic

We will now illustrate use of the program constructs described above in solving a slightly larger example in which some design decisions will be needed to ensure efficiency. We consider the pseudo-arithmetic problem "two+two=four". Our aim is to find an assignment of digits to each of the letters t,w,o,f,u, and r such that when the letters in "two+two=four" are replaced by those digits, a valid addition results.

We will use global variables to hold the answer. This is not essential but it allows us to define an operation to express a result, left in the variables, as a sequence that associates each letter with its numeric value. In this definition we make use of the postfix maplet constructor |->. The postfix expression representing the ordered pair "t"$\mapsto t$ is " t" t |->. Sequence terms are written within square brackets. The comma, used in mathematics to separate sequence elements, is for us an operation which moves a value from the stack to become the next sequence element; that is why a comma follows the final sequence entry in the following definition:

```
: SOLN ( -- s, leave a sequence of string int pairs )
  STRING INT PROD
  [ " t" t |-> , " w" w |-> , " o" o |-> , " f" f |-> , " u" u |-> ,
  " r" r |-> , ] ;
```

A naive way of solving our problem would be to choose different values for each of the letters, with the added constraint that f and t should not be zero, then check if our guess provides a correct answer, i.e. apply the guard:

```
t 100 * w 10 * + o + 2 *  f 1000 * o 100 * + u 10 * + r +  =  -->
```

If the guard fails, reverse execution produces a revised choice until a solution is found. However, a more efficient approach, in terms of search time, is to construct a solution gradually, applying constraints as soon as possible.

In addition to the variable corresponding to each letter we require variables `C0, C1` which will hold the carries from column zero and column one of the addition. `DIGIT` is a variable holding the set of remaining digits, i.e. those which have not yet been allocated to any letter. Following its declaration we re-initialise the history stack, which at this point contains the potential to release the space allocated for the set should execution reverse back past all choices; the re-initialisation is to prevent this happening.

```
0 9 .. VALUE_ DIGIT ( declare set of digits ) HP! ( protect data )
```

Standard Forth variable declaration syntax is `<expression> VALUE <name>`, and in RVM-FORTH this can also be used to declare local variables. `VALUE_` is used to declare a variable which is to be restored on reverse execution.

The operation `PUZZLE` below leaves a solution in the variables `t,w,o,f,u,r`, and also prints the the solution. Before coding the problem we realise `f` must have the value 1, and also that `r`, which we will calculate after guessing `o`, must be even. For each column in the addition we must calculate the number to write down and the carry. We perform both calculations simultaneously with the `/MOD` operation. For example if `o` has the value 9, then

```
o o + 10 /MOD
```

would leave 8 and 1 on the stack.

We begin with the low order digits, guessing a value for `o` and then calculating `r` and `C0` using `/MOD`.

We must be sure `r` is different from `f`, but we do not check this since we know that `r` must be even and `f` is one. Our first guard is to check that `r` is different from `o`, and on passing this guard we perform a set-subtraction to remove `r` and `o` from `DIGIT`.

We then choose `w` from the remaining digits and calculate `u`, using a guard to check that this is equal to some remaining digit other than `w`.

When this guard is passed we only have to guess `t`, which, as a leading digit, cannot be zero, so we subtract `u`, `w`, and 0 (zero) from `DIGIT` before choosing a value for `t`. The final guard is to check that the resulting value to write down for column two is equal to `o` and the carry is equal to `f`.

```
: PUZZLE ( --, leave a solution to two+two=four in the given variables )
  1 to f  DIGIT INT { f , } \ to DIGIT
  DIGIT CHOICE to o
  o o + 10 /MOD to C0 to r
  r o != --> DIGIT INT { r , o , } \ to DIGIT
  DIGIT CHOICE to w
  w w + C0 + 10 /MOD to C1 to u
  u w !=  u DIGIT IN  AND -->
  DIGIT INT { u , w , 0 ( zero) , } \ to DIGIT
  DIGIT CHOICE to t
```

```
t t + C1 + 10 /MOD f = --> o = -->
CR ( output carriage return ) SOLN .SEQ ( print the solution) ;
```

6.1 Prompting for further solutions

PUZZLE finds us one solution, but this is a problem that has several. If we run PUZZLE from the console, we will obtain a solution, but several potential solutions remain that are not yet printed. The RVM can be prompted to produce these by using the command ? .

```
PUZZLE <enter>
[(t,9),(w,3),(o,8),(f,1),(u,7),(r,6)]ok
? <enter>
[(t,9),(w,2),(o,8),(f,1),(u,5),(r,6)]ok
...
? <enter>
[(t,7),(w,3),(o,4),(f,1),(u,6),(r,8)]ok
R ko
```

The last ? produces a ko response: there are no more solutions so execution is infeasible. This is the point at which the space allocated for the set of digits would have been garbage collected if we had not taken the precaution of clearing the history stack after the set was allocated.

This possibility for an operation to terminate whilst leaving in place the potential for further computation can give rise to confusing symptoms when debugging. if, after getting one answer from PUZZLE, we load and run another backtracking program which has no solutions. Then instead of producing a ko response, its effect will be to invoke further execution of the program which preceded it. This certainly confused us when we first experienced it, so we gave this bug a special name - "the ghost of programs past". Fully developed programs will not cause this symptom however, as the program will incorporate a wrapper that ensures clean termination and full garbage collection.

6.2 Generating multiple solutions to a search problem

One example of the kind of wrapper mentioned above collects all solutions to a search, leaves them on the stack as a set, collects garbage, and restores values of reversible variables. We code it as follows:

```
: ALL-SOLNS INT STRING INT PROD PROD POW { <RUN PUZZLE SOLN RUN> } ;
```

The postfix expression INT STRING INT PROD PROD POW, giving the type of the elements in the following set, may be read from right to left as follows. Each element in the set is: POW a set of PROD pairs, of which the second element is PROD a pair consisting of INT STRING an integer and a string, and the first element INT is an integer.

We recall that when code between `<RUN..RUN>` executes, it generates a value on the stack, provided in this case by `SOLN`, which is added to the set under construction. Code is then reflected back (reverse execution) which prompts the code to generate another solution. This terminates when all possible choices have been exhausted.

We can also collect solutions until the set of solutions found so far satisfies some property. The following operation leaves a set containing two solutions.

```
: TWO-SOLNS ( -- s, leaves a set containing two solutions of PUZZLE )
   INT STRING INT PROD PROD POW
  { <COLLECT PUZZLE SOLN TILL CARD 2 = SATISFIED> } ;
```

The code between `<COLLECT..TILL` generates values which are added to the set of solutions. `TILL` provides the set under construction as an argument to the following condition test, namely `CARD 2 =`. If the test returns a true flag or if there are no more unexplored choices, the structure terminates. `CARD` removes the set from the stack and leaves is cardinality (size), so termination occurs when two solutions have been found.

The successive solutions generated by `PUZZLE` are obtained by backtracking to the most recent choice with an unexplored alternative. Solutions are thus found in a fixed order, which is given by the order in which the search tree is traversed. Successive solutions will be tightly grouped as tightly as possible, being separated by the minimum amount of backtracking required to generate another solution rather than spread evenly across the sample space. Also, we will find the same two solutions each time we run `TWO-SOLNS`. Often we would prefer an evenly distributed sample of solutions, and the problem of obtaining such a sample can be solved by a form of probabilistic choice, the topic of the following section.

7 Probabilistic Choice

Probabilistic choice is important for breaking symmetry, probabilistic simulation (including finalisation of quantum computations), sampling of large search spaces, and variation of search heuristics.

The high level language structures generally used in discussions of probabilistic choice [12,11] are choice from a set (with every item having an equal chance of being selected) and a binary choice associated with probabilities p and $1 - p$. We write $x \ :\oplus \ E$ to represent the assignment of a random choice from the set E to x. We write $S \ {}_p\!\oplus \ T$ to represent a binary choice of operation S with probability p, or T with probability $1 - p$, $(0 < p < 1)$.

In terms of Landauer erasure the use of random choice requires a prepared field into which random or pseudo-random numbers data can be written, requiring an initial input of energy. One important question is whether the generation of random samples should be reversible. There is no thermodynamic significance in the decision (we can think of selecting numbers from a table, where incrementing a table pointer is a reversible operation) but it is a decision with considerable semantic impact.

Using a reversible pseudo-random number generator gives the desirable property that, even for a probabilistic program S, we have S [] S = S. That is, if backtracking returns to $S \sqcap S$ the same probabilistic choices will be made the second time S is executed. It has the disadvantage that separate random choices executed at equal history stack depths will not be "independent" in the probabilistic sense. The RVM provides both reversible and non reversible pseudo-random number generators. The reversible constructs have the same name as corresponding irreversible ones but with an appended underscore. Our discussion here utilises irreversible constructs, so that successive runs through the same code following backtracking may take different probabilistic choices.

7.1 Binary choice, expectation, and feasibility

A simple choice of assigning 1 to x with probability 2/3 and 2 to x with probability 1/3 is written in RVM-FORTH as

```
2 3 (+) IF 1 to x ELSE 2 to x THEN
```

Here, (+) will remove the two numbers supplied, and generate a true flag with probability two thirds or a false flag with probability one third.

Our semantics of probabilistic choice [12] is based in the expectations of random variables. We can think of the above code as an experiment which leaves a value in x. If the experiment is repeated many times (and each experiment is independent) we expect the long term average value of x to approach $2/3 * 1 + 1/3 * 2$

Once this form of choice is made it cannot be revised by backtracking. This can lead to a problem in interpreting the idea of expectation in conditions of "partial feasibility", as exemplified by the following code:

```
: TEST 2 3 (+) IF 1 to x ELSE 2 to x THEN  x 2 = --> ;
```

We have a probability of two thirds that TEST will leave the value 2 on the stack, and probability one third that is will prompt reverse execution, making the idea of the average value of the result meaningless, since in some cases there will not be a result.

This gives a situation which we cannot currently describe through any formal semantics. However, we can describe random choice which is subject to "feasibility". A program continuation is feasible if it will run to completion without backtracking past the current point of execution.

To make our choice subject to feasibility we insert an operator FIS~.

```
2 3 (+) FIS~ IF 1 to x ELSE 2 to x THEN  x 2 = -->
```

FIS~ expects a flag to be on the stack. Forward execution of FIS~ has no effect, but if execution reverses back to FIS~ due to infeasibility, it switches the value of the flag and recommences forward execution. Thus if 2 3 (+) has generated a true flag, 1 will be assigned to x and the condition x 2 = will be false, causing the following guard to reverse execution. When reverse execution arrives back at FIS~ the flag is switched to false. Forward execution begins and this time 2 will be

assigned to x and the guard will let execution continue ahead.

Random choice from a set, governed by feasibility, is given by the operation FIS-PCHOICE which expects a set on the stack and returns one of its elements. We also provide PCHOICE which makes an irrevocable random choice from a set. As would be expected from the above discussion, we have a formal semantics for FIS-PCHOICE but PCHOICE is only suitable for deployment in situations where possible infeasibility is not an issue (in which case its behaviour is equivalent to and more efficient than FIS-PCHOICE).

7.2 A randomised version of a pseudo-arithmetic puzzle

We can use FIS-PCHOICE instead of CHOICE in the two+two=four problem in such a way that we obtain a random sample of possible solutions. Our code is exactly the same as for PUZZLE (above) except for the use of probabilistic choice. Each execution of RANPUZ chooses a possible solution at random, all solutions being equally likely to be chosen.

```
: RANPUZ ( -- leave a random solution to two+two=four in variables)
  1 to f  DIGIT INT { f , } \ to DIGIT
  DIGIT FIS-PCHOICE to o
  ...  ;
```

And we can provide a wrapper to generate two solutions and perform garbage collection.

```
: TWO-AT-RANDOM INT STRING INT PROD PROD POW
  { <RUN 1 100 .. CHOICE DROP RANPUZ SOLN RUN> } ;
```

The purpose of `1 100 .. CHOICE DROP` is to ensure that the following RANPUZ operation is executed sufficiently many times to be (almost) sure of getting two different answers. In fact, given that this problem has seven solutions, we have a probability of $(1/7)^{99}$ of only getting one result.

We should note that the reverse execution invoked to search for additional solutions, e.g. within a `{ <RUN .. RUN> }` structure, has to be distinguishable from reverse execution triggered by arriving at an impasse caused by a lack of feasible choices, so that FIS-PCHOICE can generate an alternative in response to the latter situation but not in response to the former. Indeed there are three different reverse execution modes withing the RVM, triggered by (i) infeasible continuation, (ii) a request for further solutions, and (iii) abandoning a search.

7.3 The random choice order implementation of non-deterministic choice

There is one further choice from a set that has a random aspect. The operation RANDOM-CHOICE expects a set on the stack, and makes a random choice from the set. It differs from FIS-PCHOICE in that it will yield further results on request. It resembles the operation CHOICE, in that, on request, it will eventually generate all elements of the set, and can therefore be considered as an implementation of non-deterministic choice. We can distinguish between CHOICE, RANDOM-CHOICE and

FIS-PCHOICE with the following test programs. Each of them chooses values from the set 1..3 within the context of a { <RUN..RUN> } structure, and also prints out each value as it is chosen.

```
: T1 ( -- s ) INT { <RUN 1 3 .. CHOICE DUP . RUN> } ;
: T2 ( -- s ) INT { <RUN 1 3 .. FIS-PCHOICE DUP . RUN> } ;
: T3 ( -- s ) INT { <RUN 1 3 .. RANDOM-CHOICE DUP . RUN> } ;
```

Operation T1 returns the set $\{1, 2, 3\}$ and always chooses these values in the same order. Operation T2 returns a singleton set, whose element is chosen at random from the set $\{1, 2, 3\}$. Operation T3 returns the set $\{1, 2, 3\}$ and chooses these elements in a random order.

RANDOM-CHOICE can be used to provide an implementation of the high level non-deterministic choice construct $S_1 \sqcap S_2 \,..\, \sqcap S_n$. The implementation technique is to use RANDOM-CHOICE to select a value from the set $1..n$ and then employ this value is a case statement to choose a particular operation. When using a search heuristic in which several alternatives score as equally likely to be the best choice, choosing them in a random order allows us to run the same program multiple times, either as sequential tactics (using the technique described earlier to abandon a search after a specified number of tentative steps) or in parallel, and to rely on the random element within the choice to provide a variation in performance that may avoid the occasional failure of the search heuristic to yield a rapid solution.

7.4 Generating sequences of independent trials: a simulation of the Monty Hall problem

Another use for the <RUN .. RUN> structure is to record the results of a sequence of independent random trials. We use an example in which each trial is a simulation of the Monty Hall Problem, a well known probabilistic brain teaser. For fuller information on this problem and the confusion it initially caused amongst purported experts, we refer the reader to the relevant Wikipedia article. We give just a short description. A game show host gives a contestant the chance to win either a goat or a car (supposed to be more valuable than a goat). The game is played on a stage, and the audience can see three closed doors. At the start of the game the host places a car behind one door and two (silent) goats behind the other doors. The contestant is than asked to choose a door. The host then chooses a different door, which is opened to reveal a goat, and asks the contestant if she would like to change hes choice of door. What should she do? Our simulation of the game and the operation to record the number of wins in n trials are as follows.

```
1 3 .. CONSTANT DOORS ( the set of doors )

: PLAY ( -- n, play the game, leave 1 if a car is won, 0 for a goat)
  (: :) ( declare an empty list of input variables )
    DOORS PCHOICE VALUE PRIZE-DOOR ( the door with the prize)
    DOORS  INT { PRIZE-DOOR , }  \  VALUE GOATS ( doors with goats)
    DOORS PCHOICE VALUE CHOSEN-DOOR ( the contestant's choice of door)
```

```
    GOATS INT { CHOSEN-DOOR , } \ PCHOICE
    VALUE OPENED-DOOR ( the door opened by the game-show host)
    ( now work out which is the new door for a contestant who changes)
    DOORS INT { OPENED-DOOR , CHOSEN-DOOR , } \ CHOICE
    PRIZE-DOOR =  ( check if it is equal to the prize door )
    NEGATE ( convert flag from -1 or 0 to 1 or 0 )
  1LEAVE ( leave top stack item as result ) ;

: GAMES ( n1 -- n2, n2 is the number of prizes won in n1 games)
  (: VALUE #GAMES :)
     INT INT PROD { <RUN 1 #GAMES .. CHOICE PLAY |-> RUN> }
     SUMSEQ ( sum the results, giving the number of cars won)
  1LEAVE ;
```

In the `GAMES` operation, the `{ <RUN .. RUN> }` structure is used to provide a sequence of results, that is a set of ordered pairs in which the first element is a number between 1 and the size of the sequence. The position of each result in the sequence is not an indication of when it occurred, but since we are recording independent trials that is of no interest to us in any case.

The result of this experiment, we mention in passing, is to show that by changing doors, the contestant can double the chance of winning. The importance for reversible computation is that the structure used has advantages over a loop, for example a more declarative semantics, and since the code is reversed and garbage collected at each trial, the structure provides a transparent way to limit memory usage on long runs.

7.5 Random values and expression side effects

A random number generator contains "hidden state", in the sense that each invocation creates internal changes required to ensure the next invocation will produce an independent random value from the first.

This causes the usual semantic problems if multiple sub-expressions invoking the random number generator are allowed to co-exist.

An example of how the problem could manifest in a high level reversible language consider the two putative expressions below, which have the form $2 * E$ and $E + E$.

$$2 * (x := 1 \ {}_{0.5}\oplus \ \ x := 2 \diamond x)$$
$$(x := 1 \ {}_{0.5}\oplus \ \ x := 2 \diamond x) \ + \ (x := 1 \ {}_{0.5}\oplus \ \ x := 2 \diamond x)$$

These expressions risk having different values if the code for the second is compiled in such a way that two independent random values are used. RVM-FORTH does not attempt to provide any solutions, and it is up to any designer of an associated high level reversible language to ensure such problems cannot arise, e.g by providing a syntax that allows at most one random choice to occur within an expression.

8 Conclusions and Future Work

We have described a virtual machine that can serve as a target implementation platform for a reversible probabilistic guarded command language. The classical guard and choice constructs have been given a backtracking interpretation, and some novel programming structures, arising from reversibility, have been described. The work described here is complemented by our investigations into the semantics of probabilistic reversible guarded command languages, where these programming structures are given a tractable formal semantics.

Our long term aim is to produce a highly expressive reversible probabilistic guarded command language which can be integrated into a formal development method. In addition to adding reversibility, we want to add a functional sub-language and obtain optimal trade offs between proof effort and efficient data representation. The RVM, which will provide the execution platform for this language, is equipped with features that minimise the semantic distance between source and target language.

We have programmed a large number of classical problems on the RVM. Reversibility sometimes provides solution techniques which do not yet have a formal semantics, and such constructs stimulate further theoretical investigations.

References

[1] J-R Abrial. *The B Book*. Cambridge University Press, 1996.

[2] Holger Bock Axelsen, Robert Glück, and Tetsuo Yokoyama. Reversible Machine Code and its Abstract Processor Architecture. In Volker Diekert, Mikhail V. Volkov, and Andrei Voronkov, editors, *Computer Science – Theory and Applications. Proceedings*, volume 4649 of *Lecture Notes in Computer Science*, pages 43–54 Springer-Verlag, 2007.

[3] H G Baker. The Thermodynamics of Garbage Collection. In Y Bekkers and Cohen J, editors, *Memory Management: Proc IWMM'92*, volume 637 of *Lecture Notes in Computer Science*, pages 507–520. Springer-Verlag, 1992. See ftp://ftp.netcom.com/pub/hb/hbaker/ReverseGC.html.

[4] H G Baker. Linear Logic and Permutation Stacks—the Forth shall be First. *SIGARCH Comput. Archit. News*, 22(1), pages 34–43, 1994.

[5] C Bennett. The Logical Reversibility of Computation. *IBM Journal of Research and Development*, volume 17, pages 525–532, 1973.

[6] C Bennett. The Thermodynamics of Computation. *International Journal of Theoretical Physics*, 21, pages 905–940, 1982.

[7] ANSI J14 Technical Committee. *American National Standard for Information Systems: Programming Languages-Forth*. American National Standards Institute, 1994.

[8] R P Feynman. *Lectures on Computation.* Westview Press, 1996.

[9] C A R Hoare and He Jifeng. *Unifying Theories of Programming.* Prentice Hall, 1998.

[10] R Landauer. Irreversibility and Heat Generated in the Computing Process. *IBM J R&D*, 5, pages 183–191, 1961.

[11] Annabel McIver and Carroll Morgan. *Abstraction, Refinement And Proof For Probabilistic Systems.* Springer, 2004.

[12] W J Stoddart and F Zeyda. A Unification of Probabilistic Choice within a Design-based Model of Reversible Computation. *Formal Aspect of Computing*, 2009. Accepted for publication in the Special Issue on Unifying Theories of Programming, DOI 10.1007/s00165-007-0048-1.

[13] W J Stoddart, . Zeyda, and A R Lynas. A Design-based Model of Reversible Computation. In *UTP'06, First International Symposium on Unifying Theories of Programming*, volume 4010 of *Lecture Notes in Computer Science*, pages 63–83. Springer-Verlag, 2006.

[14] Tetsuo Yokoyama, Holger Bock Axelsen, and Robert Glück. Principles of a reversible Programming Language. In *CF '08: Proceedings of the 2008 conference on Computing frontiers*, pages 43–54. ACM Press, 2008.

[15] Tetsuo Yokoyama and Robert Glück. A Reversible Programming Language and its Invertible Self-Interpreter. In *PEPM '07: Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 144–153. ACM Press, 2007.

[16] F Zeyda. *Reversible Computations in B.* PhD thesis, University of Teesside, 2007.

[17] F Zeyda, W J Stoddart, and S E Dunne. A Prospective-value Semantics for the GSL. In M Henson, S King, S Schneider, and H Treharne, editors, *ZB2005*, volume 3455 of *Lecture Notes in Computer Science*, pages 107–202. Springer-Verlag, 2005.

[18] F Zeyda, W J Stoddart, and S E Dunne. The Refinement of Reversible Computations. In T Muntean and K Sere, editors, *2nd International Workshop on Refinement of Critical Systems*, 2003.

[19] P Zuliani. Logical reversibility. *IBM J R&D*, 45(6), pages 807–818, 2001.