Using Forth in an Investigation into Reversible Computation.

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

October 23, 2003

Abstract

Previous work by Landauer, Bennett and others has established that (a) irreversibility in the computing process is associated with inevitable dissipation of energy, and (b) this fundamental limitation on the efficiency of computation can be overcome by formulating computations in a stepwise reversible manner. We report how Forth is being used in an investigation into the effects of reversibility on programming language semantics, formal software development and programming style.

1 Introduction

In 1825 the world's first steam powered passenger railway line was inaugurated on Teesside. Powered by Stevenson's Locomotion no. 1, it transported 600 passengers plus freight at about 10mph. The rapid advance in steam locomotive design during the rest of the century and beyond was supported by the emergence of the theory of thermodynamics, which placed ultimate limitations on what could be expected of a steam engine. The same laws place fundamental lower limits on the power requirements of computing devices, and as the construction of computers approaches a physical limit, these thermodynamic considerations will become increasingly relevant to considerations of computing efficiency.

An early thermodynamic analysis of computing was given in "Irreversibility and Heat Generated in the Computing Process" (1961) by Ralph Landauer of IBM Research. [4]. Landauer argues that computing typically uses irreversible steps, for example the assignment $x := 0$ cannot be reversed because it destroys the value of x. Such steps are inevitably associated with the generation of heat and the consumption of a certain minimum amount of energy. If a computing process could be contrived which used only reversible steps, then the laws of thermodynamics would not impose any minimum energy requirement for the computation. He notes that individual steps in a computing process can be made reversible by providing additional memory storage to preserve lost data, but rejects this as a general technique as the result would be an unpredictable requirement for additional memory which would need to be irreversibly initialised to a known value: "Our unwieldy machine has therefore avoided the irreversible

operations during the running of the program, only at the expense of added comparable irreversibility during the loading of the program."

This conclusion was incorrect, because we can organise the required additional memory efficiently as a stack, and regard its initialisation as a one off cost which, once paid, will allow us to run all subsequent programs in a reversible manner. Despite its erroneous conclusion Landauer's paper made an enormous contribution in setting the terms for a debate on reversibility and was recently republished in the IBM Journal of Research and Development (in Vol 44, 2000).

Soon afterwards, Y Lecerf[5] formulated a reversible Turing Machine which potentially indicated how reversible computations could be managed, but this work did not feed into the reversible computing debate. However in 1973 Charles Bennett, a colleague of Ralph Landauer at IBM Research, described[2] how an arbitrary (one tape) Turing Machine could be translated into a reversible 3 tape machine. The latter performs the calculation of the original machine, storing any overwritten data on the (originally blank) second tape. It then copies the result to the third tape. Finally it reverses its calculations so as to terminate with the first tape back in its original condition, the second tape once again blank and the result left on the third tape. Fundamental to Bennett's analysis is that writing to a blank tape is a reversible operation. The blank second tape plays the role of the pre-initialised memory mentioned above.

An interesting recent contribution is given in the paper "Logical Reversibility" by P Zuliani [12]. Zuliani provides a similar formulation of non-reversible computation in terms of reversible computation. However, rather than using Turing Machines, he formulates his translation in terms of a variant of Dijkstra's Guarded Command Language (GCL). His work extends reversibility to computations involving non-deterministic and probabilistic choice, and presents it in a form suitable for incorporation into a software development method.

In our own work we have previously investigated aspects of "semantic reversibility". We have formulated the semantics of a reversible language in terms of predicate transformers and expression transformers[7, 10]. This work has been presented withing the framework of the B method, a formal method for producing mathematically proven software[1]. We have configured a B development tool to accept our modified form of the B language and completed the formal development of an example program (the Knight's Tour) which makes use of reversibility to implement backtracking[11]. We have proposed the use of reversible Forth to implemented the required functionality for our reversible language[6, 9, 8].

Our central thesis is that the property of reversibility can have a profound effect on what we think of as a programming language. It allows us to dispense with one of the "healthiness conditions" which Dijkstra proposed for sequential programming languages, namely the "Law of the Excluded Miracle". In addition, by supporting speculative computations which leave no trace other than to deposit a result, we can eliminate unwanted side effects and obtain an integration of state based and functional programming styles.[10]

In the present paper we attempt to clarify the relationship between thermodynamic and semantic aspects of reversibility and to indicate how the effect of reversibility on programming can be studied before we have access to physically reversible devices. The paper is organised as follows. In section two we introduce some abstract constructs for reversible computation. In section three we consider thermodynamic and semantic reversibility and relate these

to our Forth virtual machines. In section 4 we describe for the first time the Forth implementation of "speculative computation" and in section 5 we draw our conclusions.

2 Abstract Constructs for Reversible Computations

In designing languages for formal software development we look for language constructs with simple semantic rules that allow the effect of programs to be logically analysed. We work with Abrial's "Generalised Substitution Language" (GSL), which was designed for use in the construction of conventional (non-reversible) programs. GSL has a "predicate transformer" semantics which we now outline.

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 the post-condition $x = 3$ if and only if the pre-condition $x = 2$ holds. Mechanically we can calculate this by substituting $x + 1$ for x in $x = 3$ giving $x + 1 = 3$ i.e. $x = 2$. The substitution has transformed the post-condition predicate into the pre-condition predicate, hence the term "predicate transformer".

We can formulate the general rule for assignment using lambda abstraction on predicates. In $(\lambda\, x.P)E)$ let x be a variable, P a predicate and E an expression. Then $(\lambda\, x.P)E$ represents a new predicate obtained by substituting each occurrence of x in P by E. So for example:

$$(\lambda\, x.x = 3)(x + 1) \quad \equiv \quad (x + 1) = 3$$

With this notation we can express the GSL rule for assignment as:

$$[x := E]Q = (\lambda\, x.Q)E$$

In the following sections we introduce further constructs from GSL and show how they can provide a semantics of reversible computation.

2.1 Choice and Guard Constructs

GSL has a choice construct $S \,[]\, T$ which makes a non-deterministic choice between running either S or T but the semantics of the construct do not say which. The construct has the rule:

$$[S \,[]\, T]Q \quad \equiv \quad [S]Q \wedge [T]Q$$

Meaning: for a non-deterministic choice to be sure of establishing some condition, both branches of the choice must be sure to do so. We are protecting ourselves against "demonic" choice.

Choice is often used in combination with the guard construct $g \longrightarrow S$ ("g guards S"). The conditional choice

$$\text{if } g \text{ then } S \text{ else } T \text{ end}$$

can be expressed in GSL in terms of guards and choice as:

$$g \longrightarrow S \,[]\, \neg\, g \longrightarrow T$$

Here the guard is used to control which choice that is made. An operation in a choice construct can only be chosen if its guard is true.

The motivation for expressing an "if" construct in terms of choice and guards is to obtain the simplest possible rules for the analysis of programs *by other programs*. The rule for the guard construct is:

$$[g \longrightarrow S]Q \quad \equiv \quad g \Rightarrow [S]Q$$

.

We name the operation that does nothing *skip*. It has the rule:

$$[\mathit{skip}]Q \quad \equiv \quad Q$$

.

We need it to analyse the semantics of

$$\text{if } g \text{ then } S \text{ end}$$

which is now expressible as:

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

To see how all this works we now analyse the construct:

$$[\text{ if } x = 1 \text{ then } x := x + 1 \text{ end }]x = 2$$

i.e. we derive the pre-condition which ensures that the program:

$$\text{if } x = 1 \text{ then } x := x + 1 \text{ end}$$

will establish the post condition $x = 2$.

Derivation

	$[\text{ if } x = 1 \text{ then } x := x + 1 \text{ end }]x = 2$	
$\equiv$	$[x = 1 \longrightarrow x := x + 1 \,[]\, \neg\, (x = 1) \longrightarrow \mathit{skip}]x = 2$	semantics of if
$\equiv$	$[x = 1 \longrightarrow x := x + 1]x = 2 \wedge [\neg\, (x = 1) \longrightarrow \mathit{skip}]x = 2$	choice rule
$\equiv$	$(x = 1 \Rightarrow [x := x + 1]x = 2) \wedge (\neg\, (x = 1) \Rightarrow [\mathit{skip}]x = 2)$	guard rule (twice)
$\equiv$	$(x = 1 \Rightarrow x + 1 = 2) \wedge (\neg\, (x = 1) \Rightarrow x = 2)$	substitution and skip rules
$\equiv$	$\mathit{true} \wedge (x = 1 \vee x = 2)$	logic
$\equiv$	$x = 1 \vee x = 2$	logic

this analysis tells us the pre-condition is $x = 1 \vee x = 2$, a result which complies with our intuition.

2.2 The Law of the Excluded Miracle

Abrial originally separated guards and choice because this separation provides the extremely simple semantic rules given above. There is not meant to be any meaningful interpretation of the construct $g \longrightarrow S$ taken in isolation. Rather we require *the disjunction of the guards in a choice construct to be true.* In the case of the construct: " if g then s else T end " construct for example, we have guards g and $\neg\ g$ for the if and else clauses respectively, and the disjunction of the guards is thus $g \vee \neg\ g$ which is true as required.

If we take an isolated construct $g \longrightarrow S$ we find it has an apparently unrealistic behavior. For example consider the operation defined as:

$$magic \mathrel{\widehat{=}} false \longrightarrow skip$$

Suppose we need to establish any post condition Q. According to our rules *magic* can do this for us:

	$[magic]Q$	
$\equiv$	$[false \longrightarrow skip]Q$	defn of *magic*
$\equiv$	$false \Rightarrow [skip]Q$	guard rule
$\equiv$	$true$	logic

This result is problematic. Since magic can establish any condition we could use it to establish $x = 1$ and also to establish $x \neq 1$. Evidently this "miraculous" behaviour cannot be attributed to any program which is implementable in the normal sense of the word, and to rule it out of consideration Dijkstra formulated a healthiness condition known as the "Law of the Excluded Miracle" which is exactly the requirement on the conjunction of the guards which we have given above[3].

Within this requirement the "miraculous" behaviour of a construct with a false guard describes exactly what we want from a *choice that is not taken.* Such a choice should always yield a result of *true* so that the total result from a choice construct can be obtained by conjoining this with the results from the other choices.

For example consider:

	$[(false \longrightarrow S) [] T]Q$	
$\equiv$	$[false \longrightarrow S]Q \wedge [T]Q$	choice rule
$\equiv$	$true \wedge [T]Q$shown above	
$\equiv$	$[T]Q$	logic

Since the choice of S cannot be made because if its false guard, we want the contribution of this choice to be true, so that the final result can be given by $true \wedge [T]Q$.

2.3 Reversibility and Backtracking: the Law of the Excluded Miracle Revoked.

We have now seen how our predicate transformer semantics can sometimes be *too powerful* in the sense that it can describe a construct that achieves more than we can expect any program can achieve, i.e. miraculous behaviour.

We have also seen that this miraculous behaviour can be tamed by the law of the excluded miracle, so that it makes a useful contribution to our semantics by describing the effect of a choice that is not taken.

Now we are ready to consider reversible computations, where we will discover another rôle that "miracles" can play. First we need a rule for sequential composition:

$$[S;\ T]Q \quad \equiv \quad [S][T]Q$$

We now have all the rules required to describe a computation involving reversibility. Consider the "program":

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

We will give the following operational interpretation of this code. The first statement is a choice which may assign either x:=1 or x:=2. In the case that x:=1 is chosen, the guard of the following statement will be false. This condition triggers a *reversal of computation.* Execution will reverse back to the choice statement and take the alternative assignment $x := 2$. The following statement can now execute as skip, and the program terminates with x=2. On the other hand if the initial choice is to assign $x := 2$ the following statement can execute immediately, and the program terminates with x=2 without any need for backtracking.

We will now give a formal proof that our semantics enable us to calculate this result, but before we do so we need to make a comment. We now have $x = 2 \longrightarrow \mathit{skip}$ as a stand alone program statement. This offends against the Law of the Excluded Miracle. Can this be justified?

In fact what is happening is that we are using the statement *to control an earlier choice.* The overall program remains non-miraculous. The penalty we have to pay in implementation terms is to provide a reversible computation platform to run the code. The penalty in proof terms is that any proof to establish a post condition must now have two parts, the second being to show that proof is not due to magic. For a more thorough treatment of the implications for formal software development the reader is referred to our paper "Refinement of Reversible Computations" [11].

Proof: As noted above, 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: i.e. $\neg\ [S]\mathit{false}$.

For the first part we must show $[S]x = 2$

$$\begin{array}{lll}
 & [S]x = 2 & \\
\equiv & [x := 1 \,[\!]\, x := 2;\ x = 2 \longrightarrow \mathit{skip}]x = 2 & \text{by defn of S} \\
\equiv & [x := 1 \,[\!]\, x := 2][x = 2 \longrightarrow \mathit{skip}]x = 2 & \text{by seq comp rule} \\
\equiv & [x := 1 \,[\!]\, x := 2](x = 2 \Rightarrow [\mathit{skip}]x = 2) & \text{by guard rule} \\
\equiv & [x := 1 \,[\!]\, x := 2](x = 2 \Rightarrow x = 2) & \text{by skip rule} \\
\equiv & [x := 1 \,[\!]\, x := 2]\mathit{true} & \text{by logic} \\
\equiv & [x := 1]\mathit{true} \wedge [x := 2]\mathit{true} \text{ by choice rule} & \\
\equiv & \mathit{true} \wedge \mathit{true} & \text{by substitution} \\
\equiv & \mathit{true} & \text{by logic.}
\end{array}$$

For the second part we must show $\neg\ [S]\mathit{false}$

$$
\begin{array}{lll}
 & \neg\,[S]false & \\
\equiv & \neg\,[x := 1 \,[]\, x := 2;\ x = 2 \longrightarrow skip]false & \text{by defn of S} \\
\equiv & \neg\,[[x := 1 \,[]\, x := 2][x = 2 \longrightarrow skip]false & \text{by seq comp rule} \\
\equiv & \neg\,[x := 1 \,[]\, x := 2](x = 2 \Rightarrow [skip]false) & \text{by guard rule} \\
\equiv & \neg\,[x := 1 \,[]\, x := 2](x = 2 \Rightarrow false) & \text{by skip rule} \\
\equiv & \neg\,[x := 1 \,[]\, x := 2]\neg\,(x = 2) & \text{by logic} \\
\equiv & \neg\,([x := 1]\neg\,(x = 2) \wedge [x := 2]\neg\,((x = 2)) & \text{by choice rule} \\
\equiv & \neg\,(\neg\,(1 = 2) \wedge \neg\,(2 = 2)) & \text{by assignment rule} \\
\equiv & \neg\,(true \wedge false) & \text{by logic.} \\
\equiv & true & \text{by logic.}
\end{array}
$$

In general, forward execution will reverse when it encounters a choice in which all alternatives have false guards, or if it meets a single guarded command with a false guard. Reverse execution will continue until a previous choice is found which has an unexplored option. If no such choice is found reverse execution will continue back to the start of the program, and the user will be told that execution of the program was infeasible.

The law of the Excluded Miracle has been revoked in the sense that we can now have programs with potentially miraculous behaviour as executable code. If we attempt to run such a program as part of a larger program it will cause execution to reverse and an alternative choice to be made. If we attempt to run it stand alone, it will report that it cannot run, and give the user a prompt of "ko" rather than "ok".

2.4 Reversibility and Speculative Computation

Bennett's scheme for a reversible computation was to perform the computation, saving any overwritten data on a second tape; leave the result of the calculation on a third tape; then reverse the calculation to restore the initial memory state and clean the second tape.

We would like to formalise an equivalent process in the GSL language. Also, we would like to allows many such reversible computations to be incorporated into some overall computation.

In our recent work on this topic [10] we introduce the notation $S \diamond E$, where S is a program and E is an expression on the state space. It will have the following operational interpretation. S is run as a reversible computation, saving any overwritten data on a history stack. The expression E is then evaluated in the resulting state. A copy is made of the result of the evaluation. Execution of the evaluation and the program S is then reversed to restore the original state, and the copy of the result becomes the result of the evaluation of $S \diamond E$..

We can treat $S \diamond E$ as *an expression on the current state space.* Its evaluation has no effect on the program state. Here is a simple example:

$x := 0;\ y := (x := 3 \diamond 2 * x)$

The expression $(x := 3 \diamond 2 * x)$ evaluates to 6. Its evaluation leaves x set to its previous value, so the overall effect is to set $x = 0$ and $y = 6$.

We have given a more substantial example, a mini-max algorithm, in a recent paper [10]. When coding the algorithm speculative computation is used to "look ahead" in the game without entailing any state change. This gives the feeling of a functional programming style, but one in which we can freely access global state, i.e. we do not have to carry state with us in the form of function parameters.

The formal definition of $S \diamond E$ is given fully in [10]. It uses the idea of substitution in expressions. For example $x := 3 \diamond 2 * x$ is defined as the substitution of 3 for x in the expression $2 * x$. As with predicate transformers, a rule is given for each connective in the language (guard, choice, sequential composition...). To deal with non-deterministic choice we use the form $S \diamond \{E\}$ to represent the set of all possible values of E that could be obtained from running S. For example:

$$(x := 1 \,[]\, x := 2) \diamond \{x\} = 1, 2$$

. If we run a miraculous program within the context of a speculative computation we obtain an empty set of results: e.g.

$$magic \diamond x = \{\}$$

3 Logical and Semantic Reversibility

We have seen that the notion of reversibility has profound implications for computing. The logical reversibility of Landaur and Bennett removes any known theoretical requirements for a minimum energy dissipation during the computing process. The semantic reversibility discussed in the previous section allows us to formulate language semantics in a more general way, revoking Dijkstra's Law of the Excluded Miracle and introducing a mechanism for speculative computation. We now relate these two notions of reversibility.

Logical reversibility is a stronger criterion than semantic reversibility, because all "information" is relevant to the former, whereas to implement the latter we need only concern ourselves with information which can influence the outcome of the computing process.

The difference can be illustrated by considering a stack. At the beginning of a computation the memory locations which are allocated to the stack contain no semantic information. Their value cannot influence the computation. In thermodynamic terms, however, the loss of any information from these locations will have exactly the same implications for energy consumption as the loss of information from any other locations.

It would seem then, that prior to pushing a value to the stack we must save the previous contents of the next free location. For that we could use another stack but that in turn will need to have its data saved... We seem in danger of an infinite regression!

To avoid this we must invest an initial outlay of energy in irreversibly initialising the stack locations to zero (or any other agreed value). The return we get for this initial investment is that reversing a write to the stack cell now only entails resetting that location to its agreed initial value, rather than restoring its old contents. The stack can be used repeatedly on this basis with no further thermodynamically unavoidable energy consumption overheads.

If we are mainly interested in the effect of reversibility on programming, we can ignore such considerations and concentrate on semantic reversibility. Also, since logical reversibility implies semantic reversibility, the semantic structures we are researching will be readily implementable on a logically reversible computing device designed to support sequential programming. Indeed the MIT Pendulum project has already prototyped such a device, though work on reversibility has also explored alternative paradigms, such as cellular automata.

We can extend the concept of logical and semantic reversibility to apply to virtual machines. Forth can be made reversible by providing it with a history stack, modifying its primitives to record an audit trail, and providing inverse operations for all primitives. It also has a remarkable ability to be incrementally extended, e.g. with new control structures. These characteristics make it the ideal compiler target language for these investigations.

Two variants of our reversible Forth virtual machine have been described in previous EuroForth conferences [6, 8]. Both use a history stack to keep an audit trail of changes, but otherwise their organisations are very different.

That described in [6] uses a multiple code field threaded code approach with separate code fields to support forward and reverse threading of instructions. Reverse execution is interpreted literally, with even control transfers being reversed. The design has three code fields per operation, which serve for normal (non-reversible) execution, conservative (reversible execution) and reverse execution. This allows the same compiled code to have three possible interpretations.

This machine had some pretensions to being efficient in so much as it is implemented in assembler rather than C or Java, but the high cost of frequent control jumps associated with threaded code led us to develop a second reversible virtual machine which uses native compiled Forth. In this machine the history stack is used to hold the addresses of the reverse execution routines as well as any overwritten data. At the start of reverse execution the history stack becomes the system stack. The entry addresses for the reverse execution routines can then be thought of as return addresses and reverse execution just returns into the top such routine. This finds any data it needs on the stack and then returns into the next routine.

We originally intended the second machine to supplant the first, but we now see they both have a role to play in our investigation.

In our current work the design of the first machine has been simplified. Each threaded code operation now has just two code fields for forward and reverse execution. The machine no longer supports a non-reversible computation mode. Our aim is to develop this design to provide a rigorous simulation of logical reversibility. Composed uniquely of stepwise reversible components and their inverses, it will serve as a test bed which can demonstrate how the language structures we propose can be implemented in logically reversible components.

The second machine will be used to investigate how efficiently semantic reversibility can be implemented on the ubiquitous i386 architecture, and to serve as the main execution platform to experiment with programming in a reversible language.

4 Implementing Speculative Computation

In this section we consider the Forth implementation of speculative computation. Implementation of the other abstract constructs for reversible computation (choice and guard) has been described in a previous paper [8].

Speculative computation should cope with nested instantiations such as:

$$S \diamond ((T \diamond \{E\}) \cup (U \diamond \{F\}))$$

in such a way that the only effect is to generate the resulting value and leave a

reference to it on the parameter stack. The intermediate values generated e.g. by $T \diamond \{E\}$ will have been uncomputed by reverse execution, and a deep copy of them has to be made for incorporation into the result.

We will attempt to explain the implementation by giving the equivalent Forth syntax, then tabulating the forward and reverse code laid down when this syntax is compiled. Our description is for the first of our virtual machines described above, where the same compiled code has a different interpretation in forward and reverse execution.

The compiler translates the phrase $S \diamond E$ into the Forth code:

```
<AFTER S E C AFTER>
```

Here `S` is the Forth translation of the GSL code S, `E` the Forth translation of the GSL code E and `C` is code to insert the current value of E into the result set.

The code `C` depends on the type of E. If E is an integer value the code will just insert the current result into the result set. If the result is a more complex data structure such as a set, a reference to the structure will be left as the result. However we cannot just insert the reference in the result set as the structure it refers to will be uncomputed during reverse execution. In this case C must make a deep copy of the result and insert a reference to this copy in the result set.

In the following tabulation of code laid down by `<AFTER S E C AFTER>`, numbering is used to indicate the order of execution and the destination of branches. The mnemonics **branch&f** and **branch&r** indicate "branch and reverse execute" and "branch and forward execute" respectively. The table is followed by a list of notes, which again follow the execution order numbering of the table.

Forward	Reverse
	14. Continue reverse execution
1. Reserve space for result set.	9. branch&f to 10.
2. Push obtained address to speculative computation stack	
3. S	8. S'
4. E	7. E'
5. Deep copy current result to result set	
6. Magic	
10. Resize result set.	13. branch&r to 14
11. Move result reference from speculative computation stack to parameter stack	12. Deep release result

Notes

1. The size of the result set is unknown at this point, so a generous amount of space is reserved. The case in which this is insufficient is not covered in our description.

2. This stack is needed for nested invocations of speculative execution.

3. Forward execution code for S.

4. Forward execution code for E.

5. The comment "deep copy" is only relevant where the current result is a reference. In that case the values referenced (at all levels) must be copied, as they will be uncomputed during reverse execution.

6. The effect of *magic* is to reverse execution.

7. Reverse execution code for E.

8. Reverse execution code for S. If S is non-deterministic, execution will reverse through S to a point of choice with an unexplored alternative. At that point forward execution will be resumed. If no unexplored choices remain reverse execution will reach the command before S'.

9. Reverse execution only reaches this point after all forward execution paths through S have been exhausted. We now branch past the code for S and E and continue forward execution from that point.

10. Generally, too much space will have been reserved for the result. The excess is recovered at this point.

11. We are now ready to place a reference to the result on the parameter stack. This completes the speculative computation of $S \diamond E$.

12. This stage happens later, when execution reverses back over the speculative computation. Since the only effect of the speculative compilation was to leave a set of results, the only thing to be done during reverse execution is to release the space allocated for this result. For logical reversibility it would be more accurate to say deep erasure of the result, as the memory space allocated would need to be returned to its initialised value.

13. This branch takes us over the reverse code for the speculative computation, which must not be executed at this point.

5 Conclusions

Computing can be made "logically reversible" by preserving information at each step. This removes any lower thermodynamic bound on theoretical energy requirements. Semantic reversibility is a weaker requirement which can be ensured by preserving at each step all information relevant to the outcome of the computation if reversed and repeated from that step.

The original aspect of our research is to interpret semantic reversibility in terms of a reformulation of the predicate transformer semantics of B GSL. The reformulation admits a wider family of possible programs by revoking Dijkstra's "Law of the Excluded Miracle". The result is a language with inherent backtracking which can incorporate speculative computations. Some of the required control structures for such a language appear to be complex to implement. Reversible Forth provides an ideal interactive environment in which to experiment with such implementations. By compiling from GSL to Forth, we keep the compilation process simple and can tackle many of the implementation complexities of reversible computing within an interactive Forth development environment.

For our experiments we use two different virtual machines. One simulates logical reversibility. It will be used to demonstrates that our proposed programming structures can be implemented entirely in terms of reversible components (we hope!). The other attempts an efficient implementation of semantic reversibility with the aim of making the techniques we investigate usable for practical programming on current architectures.

References

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

[2] C Bennett. Logical Reversibility of Computation. *IBM Journal of Research and Development*, 17, 1973.

[3] E W Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

[4] R Landauer. Irreversibility and Heat Generated in the Computing Process. *IBM Journal of Research and Development*, 5, 1961.

[5] Y Lecerf. Machines de Turing Réversibles. *Comptes Rendus de l'Academie Française des Sciences*, 257, 1963.

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

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

[8] W J Stoddart. Efficient "reversibility" with guards and choice. In A Ertl, editor, *18th EuroForth, Technical University of Vienna*, 2002.

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

[10] W J Stoddart and F Zeyda. Expression transformers in B-GSL. In D Bert, J Bowen, S King, and M Walden, editors, *ZB03*, Lecture Notes in Computer Science, no 2651, 2003.

[11] 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. Available from www.esil.univ-mrs.fr/ spc/rcs03/rcs03.

[12] P Zuliani. Logical reversibility. *IBM Journal of Research and Development*, 45(6), 2001.