Logical reversibility

Paolo Zuliani
Oxford University Computing Laboratory,
Oxford, OX1 3QD,
U.K.
pz@comlab.ox.ac.uk

Abstract

A technique has been developed for transforming a program written in the probabilistic guarded command language (*pGCL*) into an equivalent but reversible program. The approach extends previous works on logical reversibility to that language and pertains to "demonic" nondeterminism and probability. Use is made of a formal definition of logical reversibility and the expectation-transformer semantics for *pGCL*. The technique should be useful in the compilation of a general purpose programming language for quantum computation.

1 Introduction

Reversibility, when speaking of computing devices, is essentially the property of carrying out a computation in such a way that at each step it is possible to choose whether to execute that step or "undo" it, thus forcing the device and its environment to return to the same conditions as before execution.

In the context of logical reversibility we are interested in the logical model (*e.g.* Turing machine, λ-calculus, Guarded Command Language [7], *etc*) of such a device. Therefore one aims to develop a theoretical framework that allows reversibility of the computing process.

The first attempt at studying reversibility in computing processes is due to Rolf Landauer in 1961. He was the first to use the expression *logically reversible* to denote a computation whose output uniquely defines the input. The two main points of his paper [11] were that logical irreversibility is an intrinsic feature of useful computing processes, and that the erasure of information has a non-zero thermodynamic cost, *i.e.* it always generates an increase of the entropy of the universe (Landauer's principle).

The former argument was proved to be false by Lecerf in 1963 [12] and Bennett in 1973 [2] who independently developed a logically reversible device based on a Turing machine, capable of calculating any computable function. Therefore in a computation one can in principle avoid information erasure by using a logically reversible device. In subsequent years several physical models of reversible computing devices were developed; see for example the billiard-ball computer of Fredkin and Toffoli [9].

Landauer's principle has been used by Bennett in 1981 to resolve one of the long-standing problems of physics: the paradox of Maxwell's demon. What prevents the demon from breaking the second law of thermodynamics is the fact that it must erase the record of one measurement to make room for the next, and we know that such a process is physically irreversible [3]. In particular, the reversible techniques of this paper do not apply to the demon's calculation because it is permitted only one bit of scratchpad memory.

The physics of computation has gained interest as efforts directed to apply quantum theory to computation have proved successful and with important potential applications to real problems. The most famous of all quantum algorithms is Shor's algorithm for integer factorization [19]. This has of course raised the question whether it is possible to develop a suitable programming method for quantum computers, which we know are inherently reversible devices. For a traditional imperative programming method, one of the problems is represented by the assignment statement, which is logically irreversible by its own nature. For higher-level languages it is represented by nondeterminism and probability.

The purpose of this paper is to provide a modern extension of Bennett's work on reversible Turing machines to include nondeterminism and probability. In particular, we shall give rules that transform *probabilistic Guarded Command Language (pGCL)* [15] programs to equivalent but reversible *pGCL* programs. Furthermore, we extend Bennett's result to probabilistic computations, so that also probabilistic classical algorithms can be made reversible and run on a quantum computer. The work's importance arises as a result of the desire to compile general-purpose programming languages (*e.g.* [18]) for quantum computation. Among other things, such a programming language must give the possibility of simulating classical computations on a quantum computer, and our work supplies the technique for a direct compilation of an irreversible program into a reversible one.

2 Applications

As mentioned before, logical reversibility is strictly connected to quantum computation. The reason is that the evolution of a quantum system is governed by operators which are *unitary*. Unitary operators have, among other properties, that of being invertible: given a quantum mechanical operator U there always exists an inverse operator U^{-1} such that $U \circ U^{-1} = \mathbb{I}$, where $\mathbb{I}$ is the identity operator and $\circ$ denotes composition of operators (if operators are represented by matrices, $\circ$ becomes matrix multiplication). This means that in principle any quantum computation can be reversed. On the other hand, classical (conventional) computations were not designed to be reversible, since it was thought that in order to be able to compute any computable function a certain degree of irreversibility was necessary. This view was reflected in the basics of programming languages; take the assignment $x := 0$ for example: the previous value of variable x is lost.

If we want to develop a programming language for quantum computers it must therefore incorporate reversibility. $qGCL$ [18] is a general-purpose programming language for quantum computation, developed as a superset of $pGCL$ considered here. In particular, $qGCL$ extends $pGCL$ with four constructs:

- transformation q, that converts a classical bit register to its quantum analogue, a *qureg*;
- *initialisation*, which prepares a qureg for a quantum computation;
- *evolution*, which consists of iteration of unitary operators on quregs;
- *finalisation* or observation, which reads the content of a qureg.

$qGCL$ enjoys the same features of $pGCL$: it has a rigorous semantics and an associated refinement calculus (see for example [15, 13]), which include program refinement, data refinement and combination of specifications with code. These properties make $qGCL$ suitable for quantum program development and correctness proof, not just for expressing quantum algorithms.

With the techniques we are going to expose it is possible to transform any $pGCL$ program into a reversible equivalent one, thus making it suitable to run on a quantum computer. The result is readily extended to $qGCL$ programs, as initialisation and evolution are themselves unitary transformations, whilst finalisation is intrinsically irreversible.

The idea is to have a compiler for $qGCL$, which will produce code executable by some quantum hardware architecture, for example quantum

gates [1]. Such a compiler will be multi-platform, as $qGCL$ programs may contain classical code, along with quantum code. When they become feasible, quantum processors are likely to be expensive resources and their use should be restricted to genuine quantum computations, leaving all the other tasks to classical processors. Also, the limited availability of quantum algorithms due to the difficulty of quantum programming, is another reason of our multi-platform choice.

Classical code in $qGCL$ programs will be treated with the standard compiler techniques. Quantum code must be distinguished in two parts: transformations already unitary and classical code that needs to be run on the quantum architecture. The latter needs to be treated by the techniques of this paper, in order to produce a reversible version of the code and its corresponding unitary transformation [22]. At this point all the unitary transformations can be passed to that part of the compiler which will output the code for the chosen quantum architecture.

3 Previous work

Lecerf [12] proposed the first model of logically reversible computing. He gave a formal definition of reversible Turing machine and proved that an irreversible Turing machine can be simulated by a reversible one, at the expense of a linear space-time slowdown. However, he developed that result to prove a conjecture of theoretical computer science and his work was not immediately useful for reversible computing. Bennett's work was instead inspired by the previous studies of Landauer on the physics of computation and led to a key difference: Bennett's reversible Turing machine is a particular 3-tape Turing machine whose behaviour can be divided in three steps: during step one (forward computation) the machine carries out the required computation, saving the history of that in the second tape and using the first tape as workspace. In step two the output of the computation is copied into the third tape. In the last step the forward computation is traced back using the history tape and cleaning the first tape. So in the end the first and second tapes return to their initial configuration and the third contains the output.

In the second step lies the key difference between Lecerf's and Bennett's work, as without saving the output, any logically reversible computer would be of little practical use.

Another model of logical reversibility, the Fredkin gate [9], is a 3-bit logic

gate defined by the function $FG{:}\mathbb{B}^3 \to \mathbb{B}^3$

$$\forall c, x, y{:}\mathbb{B} \bullet FG(c, x, y) := (c, cx + \neg cy, \neg cx + cy)$$

that is, it swaps x and y if the control bit c is 0 and otherwise leaves them unchanged. The Fredkin gate is both reversible (it is its own inverse) and *conservative*: input and output have the same number of bits at 1. In conservative computing a computation is ultimately reduced to conditional exchanging of bits, since they are treated as unalterable objects which cannot be created or destroyed. The Fredkin gate is also universal for classical computation, as it is able to simulate the NAND gate for example.

Reversibility and conservativity are two independent properties: however, although conservativity is a property satisfied by many physical systems, we are not interested in it, as it does not seem to play a role for our purposes.

Fredkin and Toffoli [9] went further and described a physical model of computation, based on the reversible laws of classical mechanics, which could implement the Fredkin gate: the *billiard-ball* model of computation. The model involves planar elastic collisions between hard balls and fixed reflectors, governed by the laws of classical kinetic theory; a bit of information is given by the presence or absence of a ball at a certain time and position. The computer "hardware" is here represented by the spatial disposition of the reflectors, while the "software" and input data are given through the balls' initial conditions. Since the balls are rigid and the collisions are elastic the number of balls inside the "computer" does not vary, therefore the model is intrinsically conservative. The billiard-ball computer clearly requires a perfect isolation from all sources of thermal noise, both internal (the balls and the reflectors themselves) and external. A discussion of the thermodynamics of computation lies beyond the scope of this introduction but the interested reader can find an extensive treatment in Bennett's review [3].

Toffoli's work on reversible computing [20] considered the problem of realizing in a reversible way any function $\phi{:}\mathbb{B}^m \to \mathbb{B}^n$, for arbitrary $m, n > 0$. He solved it embedding ϕ in a bigger (*i.e.* augmented domain and codomain) invertible function, built upon a reversible primitive, the Toffoli gate. This gate is defined by the function $TG{:}\mathbb{B}^3 \to \mathbb{B}^3$

$$\forall x, c_1, c_2{:}\mathbb{B} \bullet TG(x, c_1, c_2) := (\neg x c_1 c_2 + \neg(c_1 c_2)x, c_1, c_2)$$

that is, it negates x if the control bits c_1, c_2 are set and otherwise leaves them unchanged. The Toffoli gate is universal but evidently not conservative.

When embedding a function ϕ into an invertible one, we have to provide specified values on certain input lines (*source*) and to disregard certain output lines (*sinks*). Toffoli distinguished also *garbage* lines, that is lines whose value depends on the input data and thus cannot be used as source lines for a new computation; *temporary storage* are instead output lines with constant values, thus useful for further computations. Toffoli discussed methods for reducing the use of source and garbage lines, culminating in the following theorem: using as primitive the Toffoli gate, any function ϕ can be realized reversibly, possibly with temporary storage, but with no garbage. The technique is essentially the same as that adopted by Bennett, *i.e.* uncomputing intermediate results, thus reusing temporary storage so that no garbage is left.

The very influential work of Feynman [8] considered the limitations of computers due to quantum mechanics, eventually finding out that no limitations hold. Feynman wanted to build a Hamiltonian (*i.e.* the state evolution operator for quantum systems) for a possible quantum system which could work as a computer. For this purpose he introduced two reversible primitives: the Controlled-NOT (CNOT) and the Controlled-Controlled-NOT (CCNOT) gates. The latter is the Toffoli gate, the former is again the Toffoli gate, but with just one control bit (it is a 2-bit gate). Feynman then described the "hardware" of the system: a collection of two-state quantum systems (*atoms*). Therefore one bit is represented by a single atom being in one of the two possible states. He then described the corresponding quantum operators for the CNOT and CCNOT gates using the method of creation and annihilation operators on single atoms.

The subsequent problem was to describe the operator for a general logic unit executing finite sequences of generic quantum operators: that is implementing by quantum hardware the high-level construct of iteration (though restricted to finite loops). Nowadays this problem belongs to the hardware compilation approaches (see [5, 17] for example), in which one can directly compile a high-level program into hardware, using reconfigurable hardware devices such as Field-Programmable Gate Arrays (FPGAs). Feynman solved the problem by augmenting the system with a supplementary array of atoms, to keep track of the operations performed. For a sequence of k operations to be executed we add $k + 1$ atoms (*program counter sites*). The Hamiltonian contains both the forward and backward computation (the Hamiltonian must be hermitian) and what drives the computation is the content of the program atoms. The Hamiltonian is built in a way such that if all program atoms are set to 0 then nothing happens; otherwise at any time there is only one program atom in state 1 (*cursor*), say i and this means that the

operations from 1 to i have been performed. Therefore when the cursor is in atom k all the sequence has been executed. To start the computer it is just necessary to set program atom 0 to state 1; the Hamiltonian then "passes" the cursor from atom site j to site $j+1$. When it reaches site k the Hamiltonian "kicks" the cursor back to site $k-1$ and so on, thus once we find the cursor at k we set it to 0 and stop the uncomputation.

There have been other models of computation using the laws of quantum mechanics, a good survey of which can be found in Bennett's paper on the history of reversible computing [4].

Our approach to logical reversibility differs from the previous models essentially in two points: it considers a high-level programming language instead of "low-level" models such as Turing machines and logic gates. The rigorous semantics of *pGCL* and its associated refinement calculus help us to carry out reasoning with clarity and formality. The second point is that our model includes (demonic) nondeterminism and probability, which have never been considered before. These two features are both present in quantum computation [18], and the need for designing a quantum programming language has guided our effort to include both forms of nondeterminism (demonic and probabilistic) in such a language.

4 Logical reversibility

4.1 Reversible devices

Before setting out the theory we shall need, it is worthwhile to discuss some points which will later motivate our choices.

A physically reversible device is a system whose behaviour is governed by the reversible law of physics: for example a quantum computer [6], or the billiard-ball computer [9]. If we look at such a system as a dynamical system, we may identify a state space X and a transition function (we suppose the behaviour to be time-independent) $f{:}X \to X$, possibly partial (input/output form part of state before/after as explained in next subsection). The reversibility hypothesis implies the injectivity of f, which in turn implies that any step of the evolution of the system can be traced back.

Classical irreversible computations can be carried out on a physically reversible computer, as Lecerf and Bennett discovered, but it is not trivial to prove it. The following discussion rules out one of the most obvious solutions: copying the input in the output. Let $g{:}A \to A$ be a deterministic

computation on some state space A: we may define $g_r{:}A \to A \times A$ by:

$$\forall a{:}A \bullet g_r.a := (a, g.a),$$

g_r is clearly injective and computable, so it seems that with very little effort we have given a positive answer to our question. This is not so, as the function g_r is not *homogeneous* whereas the transition function of a physical system always satisfies this property. One could recover homogeneousity by changing the domain of g_r in $A \times A$, but in this way injectivity is lost.

In conclusion we look for a logically reversible device for which it is possible to reverse any single step of the computation and which is homogeneous.

4.2 pGCL

In this section we present *pGCL*, the *probabilistic guarded-command language*, a programming language for describing probabilistic algorithms, of the type in the book by Motwani and Raghavan [16] for example. It has also found application in the description of quantum algorithms and in particular observation (or measurement) [18]. The main strengths of *pGCL* are its rigorous semantics [10, 15] and its associated refinement calculus, which make it possible to do formal reasoning and even derivation of code [13].

A guarded-command language program is a sequence of assignments, **skip**, and **abort** manipulated by the standard constructors of sequential composition, conditional selection, repetition and nondeterministic choice [7]. Assignment is in the form $x := E$, where x is a vector of program variables and E a vector of expressions whose evaluations always terminate with a single value. *pGCL* denotes the guarded-command language extended with the binary constructor ${}_p\oplus$ for $p{:}[0, 1]$, in order to deal with probabilism. The other *pGCL* basic statements and constructors are:

- **skip**, which always terminates doing nothing;
- **abort**, which models divergence;
- **var**, variable declaration;
- sequential composition, $R \fatsemi S$, which firstly executes R and then, if R has terminated, executes S;
- iteration, **while** *cond* **do** S, which executes S as long as predicate *cond* holds;

- binary conditional, $R \lhd cond \rhd S$, which executes R if predicate *cond* holds and executes S otherwise;
- nondeterministic choice, $R \,\square\, S$, which executes R or S, according to some rule inaccessible to the program at the current level of abstraction;
- probabilistic choice, $R \,{}_p\!\oplus\, S$, which executes R with probability p and S with probability $1 - p$;
- procedure declaration, **proc** $P(param) := body$, where *body* is a valid *pGCL* statement (including the specification statement, see below) and *param* is the parameter list, which may be empty. Parameters can be declared as **value**, **result** or **value result**, according to Morgan's notation [13]. As a quick explanation we will say that a **value** parameter is read-only, a **result** parameter is write-only and a **value result** parameter can be read and written. Procedure P is invoked by simply writing its name and filling the parameter list according to P's declaration.

Definition 4.1. The ***state*** x of a program P is the array of global variables used during the computation. That is

$$x := (v_1, \ldots, v_n) : T_1 \times T_2 \times \ldots \times T_n.$$

The cartesian product $T_1 \times T_2 \times \ldots \times T_n$ of all the data types used is called the ***state space*** of program P.

The only problem that might arise is when input and output have different types: this can easily solved by forming a new type from their discriminated union. Therefore there is no distinction among the type of initial, final and intermediate state of a computation, they all belong to the same state space.

For our purposes it is also useful to augment *pGCL* with the specification statement:

$$x : [pre, post].$$

It describes a computation which changes variable x in a such a way that, if predicate *pre* holds on the initial state, termination is ensured in a state satisfying predicate *post* over the initial and final states; if *pre* does not hold, the computation aborts.

Semantics for *pGCL* can be given either relationally [10] or in terms of expectation transformers [14]. We shall use the latter, due to its semplicity

in calculations. Expectation transformer semantics is an extension of the predicate transformer one. An *expectation* is a $[0,1]$-valued function on a state space X and may be thought of as a "probabilistic predicate". The set $\mathcal{Q}$ of all expectations is defined:

$$\mathcal{Q} := X \to [0,1].$$

Expectations can be ordered using the standard pointwise functional ordering and we shall use the symbol $\Rrightarrow$ to denote it. The pair $(\mathcal{Q}, \Rrightarrow)$ forms a complete lattice, with greatest element the constant expectation $\mathbf{1}$ and least element the constant expectation $\mathbf{0}$. For $i, j{:}\mathcal{Q}$ we shall write $i \equiv j$ iff $i \Rrightarrow j$ and $j \Rrightarrow i$.

Standard predicates are easily embedded in $\mathcal{Q}$ by identifying *true* with expectation $\mathbf{1}$ and *false* with $\mathbf{0}$. For standard predicate q we shall write $[q]$ for its embedding.

The set $\mathcal{J}$ of all expectation transformers is defined:

$$\mathcal{J} := \mathcal{Q} \to \mathcal{Q}.$$

In predicate transformer semantics a transformer maps post-conditions to their weakest pre-conditions. Analogously, expectation transformer $j{:}\mathcal{J}$ represent a computation by mapping post-expectations to their greatest pre-expectations.

Not every expectation transformers correspond to a computation: only the *sublinear* ones do. Expectation transformer $j{:}\mathcal{J}$ is said to be *sublinear* if

$$\forall a, b, c{:}\mathbb{R}^+, \forall A, B{:}\mathcal{Q} \bullet j.((aA + bB) \ominus \mathbf{c}) \Lleftarrow (a(j.A) + b(j.B)) \ominus \mathbf{c},$$

where $\ominus$ denotes truncated subtraction over expectations

$$x \ominus y := (x - y) \max \mathbf{0}.$$

Sublinearity implies, among other properties, monotonicity of an expectation transformer.

The following table gives the expectation-transformer semantics for $pGCL$ (we shall retain the wp prefix of predicate-transformer calculus for convenience):

$$
\begin{array}{rcl}
wp.\mathbf{abort}.q & := & \mathbf{0} \\
wp.\mathbf{skip}.q & := & q \\
wp.(x := E).q & := & q[x\backslash E] \\
wp.(R \fatsemi S).q & := & wp.R.(wp.S.q) \\
wp.(R \lhd cond \rhd S).q & := & [cond] * (wp.R.q) + [\neg cond] * (wp.S.q) \\
wp.(R \,\Box\, S).q & := & (wp.R.q) \;\sqcap\; (wp.S.q) \\
wp.(R \,{}_p\!\oplus\, S).q & := & p * (wp.R.q) + (1-p) * (wp.S.q) \\
wp.(z : [pre, post]).q & := & [pre] * ([\forall z \bullet [post] \Rrightarrow q])[x_0\backslash x]
\end{array}
$$

where q:$\mathcal{Q}$, x:X, $p \in [0,1]$ and *cond, pre, post* are arbitrary boolean predicates; $q[x\backslash E]$ denotes the expectation obtained after replacing all free occurences of x in q by espression E; $\sqcap$ denotes the greatest lower bound; z is a sub-vector of state x and denotes the variables the specification statement is allowed to change; x_0:X denotes the *initial* state. In the specification statement expectation q must not contain any variable in x_0. Recursion is treated in general using the existence of fixed points in $\mathcal{J}$.

Note that binary conditional $R \lhd cond \rhd S$ is a special case of probabilistic choice: it is just $R \,{}_{[cond]}\!\oplus\, S$. This will simplify a bit the proof of our main theorem in the next section.

For procedures we have to distinguish three cases, depending on the kind of parameter (without loss of generality we shall assume only one parameter). Consider a procedure P defined by:

$$\mathbf{proc}\ P(\{\mathbf{value}|\mathbf{result}|\mathbf{value\ result}\}\ f{:}T) := body$$

where T is some data type. Then a call to P has the following expectation-transformer semantics:

$$
\begin{array}{rcl}
wp.(P(\mathbf{value}\ f{:}T\backslash E)).q & := & (wp.body.q)[f\backslash E] \\
wp.(P(\mathbf{result}\ f{:}T\backslash v)).q & := & [(\forall f \bullet wp.body.q[v\backslash f])] \\
wp.(P(\mathbf{value\ result}\ f{:}T\backslash v)).q & := & (wp.body.q)[f\backslash E]
\end{array}
$$

where E is an expression of type T and $v{:}T$; f must not occur free in q.

In predicate-transformer semantics termination of program P is when $wp.P.true = true$, which directly translates to $wp.P.\mathbf{1} \equiv \mathbf{1}$ in expectation-transformer semantics.

Definition 4.2. Two $pGCL$ programs R, S are ***equivalent*** ($R \simeq S$) if and only if for any $q{:}\mathcal{Q}$, $wp.R.q \equiv wp.S.q$.

This definition induces an equivalence relation over the set of all programs. The following lemma will also be useful later (we skip the proof, as it is a simple application of the semantic rules just exposed).

Lemma 4.1. *For $pGCL$ programs A, B and C we have:*

$$(\mathbf{skip} \fatsemi A) \simeq (A \fatsemi \mathbf{skip}) \simeq A$$
$$(A \,\Box\, B) \fatsemi C \simeq (A \fatsemi C) \,\Box\, (B \fatsemi C)$$
$$(A \,{}_p\!\oplus\, B) \fatsemi C \simeq (A \fatsemi C) \,{}_p\!\oplus\, (B \fatsemi C)$$

4.3 Reversible programs

In this section we shall give a formal definition of reversibility for $pGCL$ programs, and establish some properties.

In order to simplify the exposition we will omit proofs which are not useful for our purposes; the interested reader can find them in [21].

Definition 4.3. A statement R is called ***reversible*** iff there exists a statement S such that

$$(R \fatsemi S) \simeq \mathbf{skip}.$$

S is called an *inverse* of R. Clearly it is not unique.

Definition 4.4. A program P is called ***reversible*** iff every statement of P is reversible.

The requirement that any statement of P and not just P must be reversible correspond to the need that any step of the computation can be inverted. The following example motivates this requirement: consider the programs R, S defined (see the next section for a formal definition of stack, **push** and **pop**)

$$R := (\mathbf{push}\ x \fatsemi\ x := -7 \fatsemi\ x := x^2)$$
$$S := \mathbf{pop}\ x$$

One can informally check that indeed $(R \fatsemi S) \simeq \mathbf{skip}$, while it is not true that any step of R can be inverted.

Lemma 4.2. *Let R be a reversible program. Then there exists a program S such that:*

$$(R \fatsemi S) \simeq \mathbf{skip}.$$

Again, S is called an *inverse* of R and it is not unique. A reversible program must necessarily terminate for all inputs, as stated in the following lemma.

Lemma 4.3. *Let R be a reversible program. Then $wp.R.\mathbf{1} \equiv \mathbf{1}$.*

The converse of the previous lemma is false. Consider the trivial program $x := 0$: it does terminate but it is certainly not reversible.

It is worthwhile to recall that here we consider probabilistic termination (*i.e.* termination with probability 1) and not just deterministic (absolute) termination. In section 6 we shall give an example of this and apply our reversibility techniques to it.

4.4 Stacks

Before turning to the main theorem of this work we shall briefly introduce a well known data structure: the stack data structure. The specifications for state and operations are, for a data type D (in terms of state x_0 before and state x after):

module *stack*
  var x:*seq* D $\bullet$
  proc push (**value** f:D) := $x : [x = f{:}x_0]$
  proc pop (**result** f:D) := $x, f : [x_0 = f{:}x]$
end

where *seq* denotes the *sequence* data type. There is no need of initialisation: any sequence of type D will do.

The semantics is the usual: **push** just copies the content of f on the top of the stack, whereas **pop** saves the top of the stack in f and then clears it. The stack is of unlimited capacity, that is we may save as many values as we wish.

From the definitions it easily follows that the precondition for **push** is *true* and the precondition for **pop** is that x_0 must not be empty.

The next lemma shows that an assignment can be regarded as particular sequential composition of **push** and **pop**.

Lemma 4.4. *For variable v:D and expression E:D we have:*

$$(\mathbf{push}\ E \fatsemi \mathbf{pop}\ v) \simeq v := E.$$

Proof. We shall consider an arbitrary expectation q over variables $x{:}seq\ D$ and $v{:}D$.

$wp.(\mathbf{push}\ E \fatsemi \mathbf{pop}\ v).q$

$\equiv$ semantics of sequential composition

$wp.(\mathbf{push}\ E).wp.(\mathbf{pop}\ v).q$

$\equiv$ semantics of **pop**

$wp.(\mathbf{push}\ E).([\forall f \bullet [\forall x, v \bullet [x_0 = v{:}x] \Rrightarrow q[v\backslash f]\,]\,])[x_0\backslash x]$

$\equiv$ logic and $x{:}seq\ D$

$wp.(\mathbf{push}\ E).([\forall f \bullet (q[v\backslash f])[x, f\backslash tail(x_0), head(x_0)]\,])[x_0\backslash x]$

$\equiv$ syntactical substitution

$wp.(\mathbf{push}\ E).([\forall f \bullet q[x, v\backslash tail(x_0), head(x_0)]\,])[x_0\backslash x]$

$\equiv$ syntactical substitution and logic

$wp.(\mathbf{push}\ E).(q[x, v\backslash tail(x), head(x)])$

$\equiv$ semantics of **push**

$(wp.(x : [x = f{:}x_0]).q[x, v\backslash tail(x), head(x)])[f\backslash E]$

$\equiv$ semantics of specification

$((q[x, v\backslash tail(x), head(x)])[x\backslash f{:}x_0])[f\backslash E]$

$\equiv$ syntactical substitution

$(q[x, v\backslash tail(f{:}x), head(f{:}x)])[f\backslash E]$

$\equiv$ sequence properties

$(q[x, v\backslash x_0, f])[f\backslash E]$

$\equiv$ syntactical substitution

$q[x, v\backslash x_0, E]$

$\equiv$ x_0 is arbitrary

$wp.(v := E).q$

□

We immediately derive the corollary that, when applied to program variables, **push** is reversible and an inverse is **pop**.

Corollary 4.5. *For variable v:D, we have:*

$$(\mathbf{push}\ v \fatsemi \mathbf{pop}\ v) \simeq \mathbf{skip}.$$

5 Reversibility

The meaning of the following theorem is that an arbitrary terminating *pGCL* computation can be performed in a reversible way. For any *pGCL* program P there is a corresponding reversible program P_r and an inverse P_i. Since $(P_r \fatsemi P_i) \simeq \mathbf{skip}$ it would seem that we cannot access the output of P_r, thus having nothing useful. However, as in Bennett's work [2], copying the final state of P_r before the execution of P_i solves the problem. In this way we will end up with the final and the initial state of P_r (the latter because of the execution of P_i). This new three-step reversible program is therefore not exactly equivalent to P but to P preceeded by a copy program that saves the initial state of P.

A *program transformer t:pGCL* $\rightarrow$ *pGCL* is a finite set of (computable) syntactical substitution rules that applied to a program P uniquely defines another program P_t. Examples of program transformers are the various preprocessors for programming languages like *C* or *C++*.

Theorem 5.1. *There exist three program transformers r, c and i such that for any terminating program P, P_r is an inverse of P_i and:*

$$(P_r \fatsemi P_c \fatsemi P_i) \simeq (P_c \fatsemi P).$$

Proof. The proof of the theorem relies on the following reversible equivalent and inverse of every atomic statement and constructor of *pGCL*. They are listed in the following table:

$pGCL$ atomic statement S	reversible statement S_r	inverse statement S_i
$v := e$	**push** v ⨟ $v := e$	**pop** v
skip	**skip**	**skip**
$pGCL$ constructor C	reversible constructor C_r	inverse constructor C_i
R ⨟ S	R_r ⨟ S_r	S_i ⨟ R_i
while c **do** S **od**	**push** b ⨟ **push** F ⨟ **while** c **do** S_r ⨟ **push** T **od**	**pop** b ⨟ **while** b **do** S_i ⨟ **pop** b **od** ⨟ **pop** b
$R \triangleleft c \triangleright S$	**push** b ⨟ $(R_r$ ⨟ **push** T $) \triangleleft c \triangleright (S_r$ ⨟ **push** $F)$	**pop** b ⨟ $(R_i \triangleleft b \triangleright S_i)$ ⨟ **pop** b
$R \,\square\, S$	**push** b ⨟ $(R_r$ ⨟ **push** $T) \,\square\, (S_r$ ⨟ **push** $F)$	**pop** b ⨟ $(R_i \triangleleft b \triangleright S_i)$ ⨟ **pop** b
$R \;{}_p\!\oplus\; S$	**push** b ⨟ $(R_r$ ⨟ **push** $T) \;{}_p\!\oplus\; (S_r$ ⨟ **push** $F)$	**pop** b ⨟ $(R_i \triangleleft b \triangleright S_i)$ ⨟ **pop** b
proc $Q(param) := body$	**proc** $Q_r(param) := body_r$	**proc** $Q_i(param) := body_i$

where v:D for some data type D, b:$\mathbb{B}$ ($\mathbb{B} := \{F, T\}$) is a boolean variable and c is a predicate. Variable declaration **var** is not listed in the table, as it does not contain any code.

Program P_r can be built from program P simply by applying to every statement of P the reversible rules given in the previous table (of course the rules must be recursively applied until we arrive at an atomic $pGCL$ statement). Similarly, program P_i can be developed from P applying the inverse rules of the table to every statement of P.

P_c is just a 'copy' program that copies the state x:X of P into a stack S_C:$stack.X$. If $x = \{v_1, v_2, \ldots, v_n\}$ then P_c is:

$$\textbf{push } v_1 \text{ ⨟ } \textbf{push } v_2 \text{ ⨟ } \ldots\ldots \text{ ⨟ } \textbf{push } v_{n-1} \text{ ⨟ } \textbf{push } v_n$$

By corollary 4.5 P_c is reversible.

The strategy is the following: P_r behaves like P, except that it saves its history in the stack S:$stack.(X \cup \mathbb{B})$. The copy program P_c copies the final state x_f of P_r into stack S_C. Finally P_i 'undoes' the computation and takes variables x, b, S back to their original value (*i.e.* before the beginning of P_r); the output is encoded in the state x_f saved by P_c in the stack S_C.

The execution of $(P_c$ ⨟ $P)$ has therefore the same effect of $(P_r$ ⨟ P_c ⨟ $P_i)$, except that x and $head(S_C)$ are swapped. Things can then be adjusted by executing $swap(head(S_C), x)$ after either $(P_r$ ⨟ P_c ⨟ $P_i)$ or $(P_c$ ⨟ $P)$. Note that $swap$ is reversible and self-inverse.

The first step of the proof is to show that every reversible atomic statement and every reversible constructor is, with regard to the previous definition, really reversible.

For **skip** the verification is immediate. For the assignment $v := E$ we have to show that:

$$(\mathbf{push}\ v \fatsemi v := E \fatsemi \mathbf{pop}\ v) \simeq \mathbf{skip}.$$

We reason:

$wp.(\mathbf{push}\ v \fatsemi v := E \fatsemi \mathbf{pop}\ v).q$

$\equiv$ seq. composition semantics

$wp.(\mathbf{push}\ v).wp.(v := E).wp.(\mathbf{pop}\ v).q$

$\equiv$ **pop** semantics

$wp.(\mathbf{push}\ v).wp.(v := E).(q[x, v\backslash tail(x), head(x)])$

$\equiv$ assignment semantics

$wp.(\mathbf{push}\ v).(q[x, v\backslash tail(x), head(x)])[v\backslash E]$

$\equiv$ logic

$wp.(\mathbf{push}\ v).(q[x, v\backslash tail(x), head(x)])$

$\equiv$ see proof of lemma 4.4

q

The proof for the constructors is by induction: the hypothesis is to have two reversible statements R_r, S_r (and their inverse R_i, S_i) and we have to prove that the six reversible constructors will still generate reversible statements.

For sequential composition we have to show that:

$$(R_r \fatsemi S_r \fatsemi S_i \fatsemi R_i) \simeq \mathbf{skip}$$

We shall simplify the LHS:

$wp.(R_r \fatsemi S_r \fatsemi S_i \fatsemi R_i).q$

$\equiv$ semantics

$wp.(R_r).wp.(S_r).wp.(S_i).wp.(R_i).q$

$\equiv$ associativity

$wp.(R_r).(wp.(S_r).wp.(S_i)).wp.(R_i).q$

$\equiv$ induction hypothesis on S_r

$wp.(R_r).wp.(R_i).q$

$\equiv$ induction hypothesis on R_r

q

Consider now the probabilistic combinator ${}_p\oplus$. Let Q_r, Q_i be the programs:

$$Q_r := \left(\begin{array}{l} \mathbf{push}\ b \fatsemi \\ (R_r \fatsemi \mathbf{push}\ T) \oplus_p (S_r \fatsemi \mathbf{push}\ F) \end{array} \right)$$

$$Q_i := \left(\begin{array}{l} \mathbf{pop}\ b \fatsemi \\ (R_i \lhd b \rhd S_i) \fatsemi \\ \mathbf{pop}\ b \end{array} \right)$$

We show that $(Q_r \fatsemi Q_i) \simeq \mathbf{skip}$:

$Q_r \fatsemi Q_i$

$\simeq$ lemma 4.1

$\mathbf{push}\ b \fatsemi (R_r \fatsemi \mathbf{push}\ T \fatsemi Q_i)\ {}_p\oplus\ (S_r \fatsemi \mathbf{push}\ F \fatsemi Q_i)$

We shall work on the LHS of ${}_p\oplus$:

$R_r \fatsemi \mathbf{push}\ T \fatsemi \mathbf{pop}\ b \fatsemi (R_i \lhd b \rhd S_i) \fatsemi \mathbf{pop}\ b$

$\simeq$ associativity

$R_r \fatsemi (\mathbf{push}\ T \fatsemi \mathbf{pop}\ b) \fatsemi (R_i \lhd b \rhd S_i) \fatsemi \mathbf{pop}\ b$

$\simeq$ lemma 4.4

$R_r \fatsemi b := T \fatsemi (R_i \lhd b \rhd S_i) \fatsemi \mathbf{pop}\ b$

$\simeq$ associativity

$R_r \fatsemi (b := T \fatsemi (R_i \lhd b \rhd S_i)) \fatsemi \mathbf{pop}\ b$

$\simeq$ conditional selection

$R_r \fatsemi R_i \fatsemi \mathbf{pop}\ b$

$\simeq$ induction hypothesis

$\mathbf{skip} \fatsemi \mathbf{pop}\ b$

$\simeq$ programming law

$\mathbf{pop}\ b$

A similar calculation of the RHS of ${}_p\oplus$ gives the same result, therefore:

$Q_r \fatsemi Q_i$

$\simeq$

push $b \fatsemi$ (**pop** b $\ {}_p\!\oplus$ **pop** b)

$\simeq$ programming law

push $b \fatsemi$ **pop** b

$\simeq$ corollary 4.5

skip

The proof for the nondeterministic combinator is almost identical to the previous, so we omit it. The conditional selection is a special case of probabilistic choice and it does not need any further attention. The proof for the iteration construct is rather long and technical, but it can be found in [21].

For the procedure definition we shall only prove the most general case of parameters, *value result*. Consider the procedure Q defined by:

$$\textbf{proc}\ Q(\textbf{value result}\ f{:}T) := body$$

we have to show that, for variable $a{:}T$:

$$Q_r(a) \fatsemi Q_i(a) \simeq \textbf{skip}.$$

We reason:

$wp.(Q_r(a) \fatsemi Q_i(a)).q$

$\equiv$ sequential composition

$wp.Q_r(a).(wp.Q_i(a).q)$

$\equiv$ def of Q_i and substitution

$wp.Q_r(a).((wp.(body_i).q[a\backslash f])[f\backslash a])$

$\equiv$ def of Q_r and substitution

$wp.(body_r).((wp.(body_i).q[a\backslash f])[f\backslash a])[a\backslash g])[g\backslash a]$

$\equiv$ logic

$(wp.(body_r).(wp.(body_i).q[a\backslash f]))[f\backslash g])$

$\equiv$ induction hyothesis

$(q[a\backslash f])[f\backslash g]$

$\equiv$ logic

$q[a\backslash g]$

$\equiv$ $\quad$ g is arbitrary

q

We can see from the table that the reversible constructor for conditional, probabilistic and nondeterministic choice are very similar, whereas the inverse constructor is the same for the three of them. In fact for the issue of reversibility it does not matter in what manner a selection of two possible ways has been carried out: it only matters which way has been followed.

□

6 Example

In this section we shall illustrate the application of our reversible and inverse techniques on a program which terminates only with probability 1 (not absolutely). Consider the following program P:

$$P := \left(\begin{array}{l} \textbf{var}\ \ c{:}\mathbb{B} \bullet \\ \quad c := T \fatsemi \\ \quad \textbf{while}\ c\ \textbf{do} \\ \quad\quad (\textbf{skip}) \ {}_{\frac{1}{2}}\!\oplus\ (c := F) \\ \quad \textbf{od} \end{array} \right)$$

elementary probabilistic reasoning shows that $wp.P.\mathbf{1} \equiv \mathbf{1}$. Using the reversible rules of the table we develop program P_r:

P_r

$=$ $\quad$ def of P_r

$$\left(\begin{array}{l} \textbf{var}\ \ c{:}\mathbb{B} \bullet \\ \quad c := T \fatsemi \\ \quad \textbf{while}\ c\ \textbf{do} \\ \quad\quad (\textbf{skip}) \ {}_{\frac{1}{2}}\!\oplus\ (c := F) \\ \quad \textbf{od} \end{array} \right)_r$$

$=$ $\quad$ seq composition and local block

$$\begin{array}{l} \textbf{var}\ \ c{:}\mathbb{B} \bullet \\ \quad (c := T)_r \fatsemi \\ \quad \left(\begin{array}{l} \textbf{while}\ c\ \textbf{do} \\ \quad\quad (\textbf{skip}) \ {}_{\frac{1}{2}}\!\oplus\ (c := F) \\ \quad \textbf{od} \end{array} \right)_r \end{array}$$

$=$ assignment and loop

$$
\begin{array}{l}
\textbf{var}\ \ c, b{:}\mathbb{B} \bullet \\
\quad \textbf{push}\ c \fatsemi \\
\quad c := T \fatsemi \\
\quad \textbf{push}\ b \fatsemi \textbf{push}\ F \fatsemi \\
\quad \textbf{while}\ c\ \textbf{do} \\
\qquad \Big((\textbf{skip}) \ {}_{\frac{1}{2}}\!\oplus\ (c := F) \Big)_r \fatsemi \\
\qquad \textbf{push}\ T \\
\quad \textbf{od}
\end{array}
$$

$=$ probabilistic choice

$$
\begin{array}{l}
\textbf{var}\ \ c, b{:}\mathbb{B} \bullet \\
\quad \textbf{push}\ c \fatsemi \\
\quad c := T \fatsemi \\
\quad \textbf{push}\ b \fatsemi \textbf{push}\ F \fatsemi \\
\quad \textbf{while}\ c\ \textbf{do} \\
\qquad \Big((\textbf{skip} \fatsemi \textbf{push}\ T) \ {}_{\frac{1}{2}}\!\oplus\ ((c := F)_r \fatsemi \textbf{push}\ F) \Big) \fatsemi \\
\qquad \textbf{push}\ T \\
\quad \textbf{od}
\end{array}
$$

$=$ assignment

$$
\begin{array}{l}
\textbf{var}\ \ c, b{:}\mathbb{B} \bullet \\
\quad \textbf{push}\ c \fatsemi \\
\quad c := T \fatsemi \\
\quad \textbf{push}\ b \fatsemi \textbf{push}\ F \fatsemi \\
\quad \textbf{while}\ c\ \textbf{do} \\
\qquad \Big((\textbf{skip} \fatsemi \textbf{push}\ T) \ {}_{\frac{1}{2}}\!\oplus\ (\textbf{push}\ c \fatsemi c := F \fatsemi \textbf{push}\ F) \Big) \fatsemi \\
\qquad \textbf{push}\ T \\
\quad \textbf{od}
\end{array}
$$

Analogously, program P_i, an inverse of P_r, is developed applying the inverse rules of the table:

$$P_i = \left(\begin{array}{l} \textbf{var}\ \ c, b{:}\mathbb{B} \bullet \\ \quad \textbf{pop}\ b \fatsemi \\ \quad \textbf{while}\ b\ \textbf{do} \\ \quad\quad \textbf{pop}\ b \fatsemi \\ \quad\quad (\textbf{skip}) \lhd b \rhd (\textbf{pop}\ c) \fatsemi \\ \quad\quad \textbf{pop}\ b \fatsemi \textbf{pop}\ b \\ \quad \textbf{od} \\ \quad \textbf{pop}\ b \fatsemi \textbf{pop}\ c \end{array} \right)$$

and we see that $(P_r \fatsemi P_i) \simeq \textbf{skip}$.

7 Conclusions

We have developed a set of rules that, given a *pGCL* program P, enables us to write another program P_r that computes the same output as P, but in a logically reversible way. For this purpose P_r saves its 'history' on a stack during the forward computation; the stack will be cleaned by the backward computation that takes P_r to its initial state. The output of the forward computation is copied onto another stack, in order to be available at the end of the process.

The contributions of this work are mainly two: with respect to previous works in logical reversibility we consider a high-level programming language, *pGCL*, instead of "low-level" models such as Turing machines and logic gates. *pGCL* enjoys a rigorous semantics and an associated refinement calculus, which facilitate reasoning about programs. The second contribution is the presence in our model of (demonic) nondeterminism and probability, which have never been considered before.

For future work it would worth discussing the uniqueness of the reversible and inverse statement S_r and S_i: our argument just shows that it is possible to find one. The proof for the reversible loop constructor might be able to be simplified by making use of further *wp* laws.

8 Acknowledgements

The author would like to thank his supervisor Jeff Sanders for suggestions and for reading various drafts of this paper.

This work has been supported by a scholarship from the Engineering and Physical Sciences Research Council (UK) and by a scholarship from Università degli Studi di Perugia (Italy).

References

[1] Adriano Barenco et al. Elementary gates for quantum computation. *Physical Review A*, 52(5):3457–3467, 1995.

[2] C. H. Bennett. Logical reversibility of computation. *IBM Journal of Research and Development*, 17:525–532, 1973.

[3] Charles H. Bennett. The thermodynamics of computation - a review. *IBM Journal of Research and Development*, 21:905–940, 1981.

[4] Charles H. Bennett. Notes on the history of reversible computation. *IBM Journal of Research and Development*, 32(1):16–23, 1988.

[5] J. Bowen, J. He, and I. Page. Hardware compilation. In J. Bowen, editor, *Towards Verified Systems*, chapter 10, pages 193–207. Elsevier, 1994.

[6] David Deutsch. Quantum theory, the Church-Turing principle and the universal quantum computer. *Proceedings of the Royal Society of London*, A400:97–117, 1985.

[7] E. W. Dijkstra. Guarded commands, nondeterminacy and the formal derivation of programs. *CACM*, 18:453–457, 1975.

[8] Richard P. Feynman. Quantum mechanical computers. *Foundations of Physics*, 16(6):507–531, 1986.

[9] Edward Fredkin and Tommaso Toffoli. Conservative logic. *International Journal of Theoretical Physics*, 21:219–253, 1981.

[10] J. He, A. McIver, and K. Seidel. Probabilistic models for the guarded command language. *Science of Computer Programming*, 28:171–192, 1997.

[11] Rolf Landauer. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development*, 3:183–191, 1961.

[12] Yves Lecerf. Machines de Turing réversibles. Récursive insolubilité en $n \in \mathbb{N}$ de l'équation $u = \theta^n u$, où θ est un isomorphisme de codes. *Comptes rendus de l'Académie française des sciences*, 257:2597–2600, 1963.

[13] C. C. Morgan. *Programming from Specifications*. Prentice-Hall International, 1994.

[14] C. C. Morgan, A. McIver, and K. Seidel. Probabilistic predicate transformers. *ACM Transactions on Programming Languages and Systems*, 18(3):325–353, May 1996.

[15] Charles Carroll Morgan and Annabelle McIver. *pGCL*: formal reasoning for random algorithms. *South African Computer Journal*, 22:14–27, 1999.

[16] Rajeev Motwani and Prabhakar Raghavan. *Randomised algorithms*. Cambridge University Press, 1995.

[17] Ian Page. Constructing hardware-software systems from a single description. *Journal of VLSI Signal Processing*, 12(1):87–107, 1996.

[18] J. W. Sanders and P. Zuliani. Quantum programming. In *MPC '00: Mathematics of Program Construction, Springer LNCS*, volume 1837, pages 80–99, 2000.

[19] P. W. Shor. Algorithms for quantum computation: Discrete log and factoring. In *FOCS '94: Proceedings of the 35th Annual Symposium on the Foundations of Computer Science*, pages 20–22, 1994.

[20] T. Toffoli. Reversible computing. In de Bakker, editor, *Automata, Languages, and Programming*, pages 632–644. Springer-Verlag, 1980.

[21] P. Zuliani. Logical reversibility. *IBM Journal of Research and Development*, 45(6):807–818, 2001.

[22] P. Zuliani. *Quantum Programming*. PhD thesis, Oxford University Computing Laboratory, 2001. Available at http://www.comlab.ox.ac.uk.