Logical reversibility

by P. Zuliani

A method has been developed for transforming a program written in the probabilistic guarded-command language (*pGCL*) into an equivalent but reversible program. The method extends previous work 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 method should be useful in the compilation of a general-purpose programming language for quantum computation.

1. Introduction

Reversibility, when one is referring to a computing device, is essentially the carrying out of a computation so 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 conditions prior to execution.

In the context of logical reversibility, we are interested in the logical model (e.g., Turing machine, λ-calculus, Guarded Command Language [1]) of such a device, with the objective of developing a theoretical framework that allows reversibility of the computing process.

The first attempt to study reversibility in computing processes was described by Rolf Landauer in 1961 [2]. He was the first to use the expression *logically reversible* to denote a computation whose output uniquely defines its input. His main points were 1) that logical irreversibility is an intrinsic feature of useful computing processes, and 2) that the erasure of information has a nonzero thermodynamic cost; i.e., it always generates an increase of the entropy of the universe (Landauer's principle).

However, the first was proved to be false in 1963 by Lecerf [3] and in 1973 by Bennett [4], who, independently, were able to theoretically construct a logically reversible device based on a Turing machine, capable of calculating any computable function. Therefore, in a computation, in principle at least, it should be possible to 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 [5].

Landauer's principle was used by Bennett in 1981 to resolve one of the longstanding 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—a physically irreversible process [6]. 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, with important potential applications to real problems. The most famous of all quantum algorithms is Shor's algorithm for integer factorization [7]. This has raised the question whether it is possible to develop a suitable programming language for quantum computers, which we know are inherently reversible devices. For a traditional imperative programming language, 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.

0018-8646/01/$5.00 © 2001 IBM

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 present rules that transform *probabilistic Guarded-Command Language* (*pGCL*) [8] programs to equivalent but reversible *pGCL* programs. Furthermore, we extend Bennett's result to probabilistic computations, so that probabilistic classical algorithms can also be made reversible and run on a quantum computer. The work's significance arises as a result of the desire to compile general-purpose programming languages (e.g., [9]) for quantum computation. Among other things, such a programming language must make it possible to simulate classical computations on a quantum computer. Additionally, our work supplies techniques for direct compilation of an irreversible program into a reversible one.

2. Applications

As mentioned above, logical reversibility is strictly connected to quantum computation, because 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.

A programming language for quantum computers must therefore incorporate reversibility. *qGCL* [9] 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, which converts a classical bit register to its quantum analog, a *qureg*.
- *Initialization*, which prepares a qureg for a quantum computation.
- *Evolution*, which consists of iteration of unitary operators on quregs.
- *Finalization* or observation, which reads the content of a qureg.

qGCL enjoys the same features as *pGCL*: It has a rigorous semantics and an associated refinement calculus (see for example [8, 10]), 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 presented here, 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, since initialization and evolution are themselves unitary transformations, while finalization is intrinsically irreversible.

It is assumed that a compiler for *qGCL* will be used which will produce code executable by some quantum hardware architecture, for example quantum gates [11]. Such a compiler should be multiplatform, since *qGCL* programs may contain classical as well as quantum code. Quantum processors are likely to be expensive resources, and their use should be restricted to genuine quantum computations, leaving all other tasks to classical processors. Also, the limited availability of quantum algorithms due to the difficulty of quantum programming is another reason for our multiplatform 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 must be run on the quantum architecture. The latter must be treated by the techniques of this paper in order to produce a reversible version of the code and its corresponding unitary transformation.[1] At this point, all of 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 [3] proposed the first model of logically reversible computing. He gave a formal definition of a 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 three-tape Turing machine whose behavior can be divided into 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

[1] Paolo Zuliani, "Quantum Programming," Ph.D. thesis, Oxford University Computing Laboratory, 2001; in preparation.

computation is traced back using the history tape and cleaning the first tape. Thus, 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, since without saving the output, any logically reversible computer would be of little practical use.

Another model of logical reversibility, the Fredkin gate [5], 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 a 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, since 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 do not consider it here, since it does not seem to pertain to the present work.

Fredkin and Toffoli [5] 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 initial conditions of the balls. 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 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 paper, but the interested reader can find an extensive treatment in Bennett's review [6].

Toffoli's work on reversible computing [12] 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 by embedding ϕ in a bigger (i.e., augmented domain and codomain) invertible function, constructed upon a reversible primitive, the Toffoli gate. This gate is defined by the function $TG:\mathbb{B}^3 \to \mathbb{B}^3$:

$$\forall\, x, c_1, c_2:\mathbb{B} \bullet TG(x, c_1, c_2) := (\neg\, xc_1c_2 + \neg(c_1c_2)x, c_1, c_2);$$

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 must provide specified values on certain input lines (*source*) and disregard certain output lines (*sinks*). Toffoli also distinguished *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* consists instead of 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 a 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 and thus reusing temporary storage so that no garbage remains.

The very influential work of Feynman [13] considered the limitations of computers due to quantum mechanics, eventually discovering that no limitations apply. Feynman wanted to build a Hamiltonian (i.e., the state evolution operator for quantum systems) for a possible quantum system which could function 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 would be 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 [14, 15] 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 the backward computation (the Hamiltonian must be hermitian), and what drives the computation is the content of the program atoms. The Hamiltonian is formulated such that if all program atoms are set to 0, nothing occurs; otherwise, there is at any time only one program atom in state 1 (*cursor*), say i, indicating that the operations from 1 to i have been performed. Therefore, when the cursor is in atom k, all of the sequence has been executed. To start the computer it is necessary simply 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 $k - 1$, uncomputes operation k, and so on, until the cursor returns to site 0. At this point, the Hamiltonian restarts the formal computation. Therefore, to achieve the execution of k operations, the cursor is set to 0 when it is at site k (so that no further computation is performed).

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 [16].

Our approach to logical reversibility differs from the previous models in essentially 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 [9], and the need to design a quantum programming language has guided our effort to include both forms of nondeterminism (demonic and probabilistic) in such a language.

4. Logical reversibility

Reversible devices

Before describing the theory we need to use, we discuss some points which will later motivate our choices.

A physically reversible device is a system whose behavior is governed by the reversible law of physics: for example a quantum computer [17] or the billiard-ball computer [5]. If we look at such a system as a dynamical system, we can identify a state space X and a transition function (we suppose the behavior to be time-independent) $f{:}X \to X$, possibly partial (input/output form part of state before/after, as explained in the 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 that is not trivial to prove. 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

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

Since g_r is clearly injective and computable, 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 homogeneity by changing the domain of g_r in $A \times A$, but in this way injectivity is lost.

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

pGCL

In this section we describe *pGCL*, the *probabilistic guarded-command language*, a programming language for describing probabilistic algorithms of the type described for example in the book by Motwani and Raghavan [18]. The language has also found application in the description of quantum algorithms and in particular observation (or measurement) [9]. Its main strengths are its rigorous semantics [8, 19] and its associated refinement calculus, which make it possible to carry out formal reasoning and derivation of code [10].

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 [1]. 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 basic statements and constructors of *pGCL* are

- **Skip**, which always terminates doing nothing.
- **Abort**, which models divergence.
- **Var**, variable declaration.
- Sequential composition, $R\,{}_9^\circ\, S$, which first 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, $R \triangleleft cond \triangleright S$, which executes R if predicate *cond* holds and executes S otherwise.
- Nondeterministic choice, $R \sqcap S$, 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 [10]. We assume 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 1

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

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

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

The only problem that might arise is the case in which input and output have different types: This can easily be solved by forming a new type from their discriminated union. Therefore, there is no distinction among the types of initial, final, and intermediate states 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:

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

The statement describes a computation which changes variable x in 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 [19] or in terms of expectation transformers [20]. We use the latter approach because of its simplicity in calculations. Expectation transformer semantics is an extension of predicate transformer semantics. An *expectation* is a [0, 1]-valued function on a state space X and may be thought of as a "probabilistic predicate." The set $\mathfrak{Q}$ of all expectations is defined as

$$\mathfrak{Q} := X \rightarrow [0, 1].$$

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

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

Table 1 Expectation-transformer semantics for pGCL.

$wp.\mathbf{abort}.q :=$	$\mathbf{0}$
$wp.\mathbf{skip}.q :=$	q
$wp.(x := E).q :=$	$q[x \backslash E]$
$wp.(R; S).q :=$	$wp.R.(wp.S.q)$
$wp.(R \triangleleft cond \triangleright S).q :=$	$[cond] * (wp.R.q) + [\neg cond] * (wp.S.q)$
$wp.(R \,\Box\, S).q :=$	$(wp.R.q) \sqcap (wp.S.q)$
$wp.(R \;{}_p{\oplus}\; S).q :=$	$p * (wp.R.q) + (1 - p) * (wp.S.q)$
$wp.(z{:}[pre, post]).q :=$	$[pre] * ([\forall z \bullet [post] \Rrightarrow q])[x_0 \backslash x]$

The set $\mathfrak{T}$ of all expectation transformers is defined as

$$\mathfrak{T} := \mathfrak{Q} \rightarrow \mathfrak{Q}.$$

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

Not every expectation transformer corresponds to a computation: Only the *sublinear* ones do. Expectation transformer $j{:}\mathfrak{T}$ is said to be *sublinear* if

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

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.

Table 1 lists the expectation-transformer semantics for *pGCL* (we retain the *wp* prefix of predicate-transformer calculus for convenience): $q{:}\mathfrak{Q}$, $x{:}X$, $p \in [0, 1]$, and *cond*, *pre*, *post* are arbitrary Boolean predicates; $q[x \backslash E]$ denotes the expectation obtained after replacing all free occurrences of x in q with expression E; $\sqcap$ denotes the greatest lower bound; z is a subvector of state x and denotes the variables the specification statement is allowed to change; and $x_0{:}X$ 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 $\mathfrak{T}$.

Note that binary conditional $R \triangleleft cond \triangleright S$ is a special case of probabilistic choice: It is just $R \;{}_{[cond]}{\oplus}\; S$. This results in some simplification of the proof of our main theorem in the next section.

With regard to the procedures used, three cases must be distinguished, depending on the kind of parameter used; without loss of generality we assume the use of 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:

$$wp.(P(\mathbf{value}\, f{:}T\backslash E)).q := (wp.body.q)[f\backslash E],$$

$$wp.(P(\mathbf{result}\, f{:}T\backslash v)).q := [\,\forall f \bullet wp.body.q[v\backslash f]],$$

$$wp.(P(\mathbf{value\ result}\, f{:}T\backslash v)).q := (wp.body.q[v\backslash f])[f\backslash v],$$

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 occurs when $wp.P.true = true$, which translates directly to $wp.P.\mathbf{1} \equiv \mathbf{1}$ in expectation-transformer semantics.

Definition 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 is also useful later (we skip the proof, since it is a simple application of the above semantic rules).

Lemma 1
For *pGCL* programs A, B, and C we have

$$(\mathbf{skip} ⨾ A) \simeq (A ⨾ \mathbf{skip}) \simeq A,$$

$$(A \sqcap B) ⨾ C \simeq (A ⨾ C) \sqcap (B ⨾ C),$$

$$(A\ {}_p\oplus B) ⨾ C \simeq (A ⨾ C)\ {}_p\oplus (B ⨾ C).$$

Reversible programs

In this section we present a formal definition of reversibility for *pGCL* programs and establish some properties.

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

Definition 3
A statement R is called *reversible* iff there exists a statement S such that

$$(R ⨾ S) \simeq \mathbf{skip};$$

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

Definition 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 corresponds to the requirement that any step of the computation must be invertible. 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**) as

$$R := (\mathbf{push}\, x ⨾ x := -7 ⨾ x := x^2),$$

$$S := \mathbf{pop}\, x.$$

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

Lemma 2
Let R be a reversible program. Then there exists a program S such that

$$(R ⨾ S) \simeq \mathbf{skip}.$$

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 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.

Recall that here we consider probabilistic termination (i.e., termination with probability 1) and not just deterministic (absolute) termination. In Section 6 we provide an example of this and apply our reversibility techniques to it.

Stacks

Before turning to the main theorem of this work, we 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 for initialization: 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 follows easily 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 a particular sequential composition of **push** and **pop**.

Lemma 4
For variable $v{:}D$ and expression $E{:}D$ we have

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

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

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

$\equiv$ 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

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

$\equiv$ 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

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

$\square$

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

Corollary 1
For variable $v{:}D$, we have

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

5. Reversibility

The significance of the following theorem is that an arbitrary terminating *pGCL* computation can be performed in a reversible manner. For any *pGCL* program P there is a corresponding reversible program P_r and an inverse P_i. Since $(P_r\,;\,P_i) \simeq$ **skip**, it would seem that we cannot access the output of P_r, thus obtaining nothing useful. However, as in Bennett's work [4], copying the final state of P_r before the execution of P_i solves the problem. In this way we would 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 rather to P preceded by a copy program that saves the initial state of P.

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

Theorem 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

$$(P_r\,;\,P_c\,;\,P_i) \simeq (P_c\,;\,P).$$

Proof The proof of the theorem relies on the following reversible equivalent, inverse of every atomic statement, and constructor of *pGCL*; they are listed in **Table 2**, where $v{:}D$ for some data type D, $b{:}\mathbb{B}$ ($\mathbb{B} := \{F, T\}$) is a Boolean variable, and c is a predicate. The variable declaration **var** is not included because it does not contain any code.

Program P_r can be constructed from program P by simply 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 program P by applying the inverse rules of the table to every statement of P.

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

$$\mathbf{push}\ v_1\,;\ \mathbf{push}\ v_2\,;\ \cdots\,;\ \mathbf{push}\ v_{n-1}\,;\ \mathbf{push}\ v_n\,.$$

By Corollary 1, 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

Table 2 Basis for proof of Theorem 1.

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

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 $(P_c \fatsemi P)$ therefore has the same effect as $(P_r \fatsemi P_c \fatsemi P_i)$, except that x and $head(S_C)$ are swapped. Adjustments can be made by executing $swap(head(S_C), x)$ after either $(P_r \fatsemi P_c \fatsemi P_i)$ or $(P_c \fatsemi P)$. 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

(**push** $v \fatsemi v := E \fatsemi$ **pop** v) $\simeq$ **skip**.

We reason as follows:

$wp.$(**push** $v \fatsemi v := E \fatsemi$ **pop** $v).q$

$\equiv$ sequential composition semantics

$wp.$(**push** $v).wp.(v := E).wp.$(**pop** $v).q$

$\equiv$ **pop** semantics

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

$\equiv$ assignment semantics

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

$\equiv$ logic

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

$\equiv$ see proof of Lemma 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

$(R_r \fatsemi S_r \fatsemi S_i \fatsemi R_i) \simeq$ **skip**.

We 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

Now consider the probabilistic combinator ${}_p\oplus$. Let Q_r, Q_i be the programs

$$Q_r := \begin{pmatrix} \textbf{push } b; \\ (R_r; \textbf{push } T) \oplus_p (S_r; \textbf{push } F) \end{pmatrix}$$

$$Q_i := \begin{pmatrix} \textbf{pop } b; \\ (R_i \triangleleft b \triangleright S_i); \\ \textbf{pop } b \end{pmatrix}$$

We show that $(Q_r; Q_i) \simeq$ **skip**:

$Q_r; Q_i$

$\simeq$ Lemma 1

push b; $(R_r;$ **push** $T; Q_i)\ {}_p\oplus\ (S_r;$ **push** $F; Q_i)$

Next, we develop the LHS of ${}_p\oplus$:

$R_r;$ **push** $T;$ **pop** $b; (R_i \triangleleft b \triangleright S_i);$ **pop** b

$\simeq$ associativity

$R_r;$ (**push** $T;$ **pop** b)$; (R_i \triangleleft b \triangleright S_i);$ **pop** b

$\simeq$ Lemma 4

$R_r; b := T; (R_i \triangleleft b \triangleright S_i);$ **pop** b

$\simeq$ associativity

$R_r; (b := T; (R_i \triangleleft b \triangleright S_i));$ **pop** b

$\simeq$ conditional selection

$R_r; R_i;$ **pop** b

$\simeq$ induction hypothesis

skip; **pop** b

$\simeq$ programming law

pop b

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

$Q_r; Q_i$

$\simeq$

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

$\simeq$ programming law

push b; **pop** b

$\simeq$ Corollary 1

skip

The proof for the nondeterministic combinator is almost identical to the previous one, so we omit it. The conditional selection is a special case of probabilistic choice, and it does not require 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 prove only the most general case of parameters, **value result**. Consider the procedure Q defined by

proc Q(**value result** $f{:}T$) := *body*.

We must show that, for variable $a{:}T$,

$Q_r(a); Q_i(a) \simeq$ **skip**.

We reason as follows:

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

$\equiv$ sequential composition

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

$\equiv$ definition of Q_i and substitution

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

$\equiv$ definition 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 hypothesis

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

$\equiv$ logic

$q[a\backslash g]$

$\equiv$ g is arbitrary

q

We can see from the table that the reversible constructors for conditional, probabilistic, and nondeterministic choice are very similar, whereas the inverse constructors are the same for all three. In fact, with respect to reversibility, it does not matter how the selection of two possible ways has been carried out: It only matters which way has been followed.

□

6. Example

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

$$P := \left(\begin{array}{l} \mathbf{var}\ c{:}\mathbb{B} \bullet \\ \quad c := T; \\ \quad \mathbf{while}\ c\ \mathbf{do} \\ \quad\quad (\mathbf{skip})\ {}_{1/2}{\oplus}\ (c := F) \\ \quad \mathbf{od} \end{array}\right).$$

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

$=$ definition of P_r

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

$=$ sequential composition and local block

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

$=$ assignment and loop

```
var $c, b{:}\mathbb{B}$ •
  push $c$;
  $c := T$;
  push $b$; push $F$;
  while $c$ do
    $((\mathbf{skip})\ {}_{1/2}{\oplus}\ (c := F))_r$;
    push $T$
  od
```

$=$ probabilistic choice

```
var $c, b{:}\mathbb{B}$ •
  push $c$;
  $c := T$;
  push $b$; push $F$;
  while $c$ do
    $((\mathbf{skip}; \mathbf{push}\ T)\ {}_{1/2}{\oplus}\ ((c := F)_r; \mathbf{push}\ F))$;
    push $T$
  od
```

$=$ assignment

```
var $c, b{:}\mathbb{B}$ •
  push $c$;
  $c := T$;
  push $b$; push $F$;
  while $c$ do
    $((\mathbf{skip}; \mathbf{push}\ T)\ {}_{1/2}{\oplus}\ (\mathbf{push}\ c; c := F; \mathbf{push}\ F))$;
    push $T$
  od
```

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

$$P_i = \left(\begin{array}{l} \mathbf{var}\ c, b{:}\mathbb{B} \bullet \\ \quad \mathbf{pop}\ b; \\ \quad \mathbf{while}\ b\ \mathbf{do} \\ \quad\quad \mathbf{pop}\ b; \\ \quad\quad (\mathbf{skip}) \triangleleft b \triangleright (\mathbf{pop}\ c); \\ \quad\quad \mathbf{pop}\ b; \mathbf{pop}\ b \\ \quad \mathbf{od} \\ \quad \mathbf{pop}\ b; \mathbf{pop}\ c \end{array}\right),$$

and we see that $(P_r; P_i) \simeq \mathbf{skip}$.

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 manner. For this purpose, P_r saves its "history" on a stack during a forward computation; the stack will be cleaned by a 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 primarily 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 not previously been considered.

For future work, consideration should be given to examining the uniqueness of the reversible and inverse statements S_r and S_i: Our argument just shows that it is possible to find one. Furthermore, consideration should be given to simplifying the proof for the reversible loop constructor by making use of further *wp* laws.

Acknowledgments

The author thanks his supervisor, Dr. Jeff Sanders, for suggestions and for reading various drafts of this paper. This work was 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. E. W. Dijkstra, "Guarded Commands, Nondeterminacy and the Formal Derivation of Programs," *Commun. ACM* **18,** 453–457 (1975).
2. R. Landauer, "Irreversibility and Heat Generation in the Computing Process," *IBM J. Res. & Dev.* **5,** 183–191 (1961).
3. 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).
4. Charles H. Bennett, "Logical Reversibility of Computation," *IBM J. Res. & Dev.* **17,** 525–532 (1973).
5. Edward Fredkin and Tommaso Toffoli, "Conservative Logic," *Int. J. Theor. Phys.* **21,** 219–253 (1981).
6. Charles H. Bennett, "The Thermodynamics of Computation—A Review," *IBM J. Res. & Dev.* **21,** 905–940 (1981).
7. Peter W. Shor, "Algorithms for Quantum Computation: Discrete Log and Factoring," *Proceedings of the 35th Annual Symposium on the Foundations of Computer Science*, 1994, pp. 20–22.
8. Carroll Morgan and Annabelle McIver, "*pGCL*: Formal Reasoning for Random Algorithms," *South African Computer J.* **22,** 14–27 (1999).
9. Jeff W. Sanders and Paolo Zuliani, "Quantum Programming," *Math. Program Constr.* **1837,** 80–99 (2000).
10. Carroll Morgan, *Programming from Specifications*, Prentice-Hall International, New York, 1994.
11. A. Barenco, C. H. Bennett, C. H. Cleve, N. Margolus, P. Shor, T. Sleator, J. A. Smolin, and H. Weinfurter, "Elementary Gates for Quantum Computation," *Phys. Rev. A* **52,** 3457–3467 (1995).
12. Tommaso Toffoli, "Reversible Computing," *Automata, Languages, and Programming*, J. de Bakker, Ed., Springer-Verlag, New York, 1980.
13. Richard P. Feynman, "Quantum Mechanical Computers," *Found. Phys.* **16,** 507–531 (1986).
14. J. Bowen, H. Jifeng, and I. Page, "Hardware Compilation," *Towards Verified Systems*, J. Bowen, Ed., Elsevier, New York, 1994, pp. 193–207.
15. Ian Page, "Constructing Hardware–Software Systems from a Single Description," *J. VLSI Signal Proc.* **12,** 87–107 (1996).
16. Charles H. Bennett, "Notes on the History of Reversible Computation," *IBM J. Res. & Dev.* **32,** 16–23 (1988).
17. David Deutsch, "Quantum Theory, the Church–Turing Principle and the Universal Quantum Computer," *Proc. Roy. Soc. Lond. A* **400,** 97–117 (1985).
18. Rajeev Motwani and Prabhakar Raghavan, *Randomized Algorithms*, Cambridge University Press, Cambridge, England, 1995.
19. H. Jifeng, A. McIver, and K. Seidel, "Probabilistic Models for the Guarded Command Language," *Sci. Computer Program.* **28,** 171–192 (1997).
20. C. Morgan, A. McIver, and K. Seidel, "Probabilistic Predicate Transformers," *ACM Trans. Program. Lang. & Syst.* **18,** 325–353 (1996).
21. Paolo Zuliani, "Logical Reversibility," *Technical Report TR-11-00*, Oxford University Computing Laboratory, Oxford, England, 2000; available at *http://www.comlab.ox.ac.uk*.

Received November 3, 2000; accepted for publication May 1, 2001

Paolo Zuliani *Oxford University Computing Laboratory, OX1 3QD, Oxford, United Kingdom (pz@comlab.ox.ac.uk).* After receiving a Laurea Degree in computer science in 1997 from Università degli Studi di Milano, Italy, Mr. Zuliani began graduate studies at Oxford University. He is now completing his work toward a Ph.D. degree as a member of the Programming Research Group in the Computing Laboratory under the supervision of Dr. Jeff Sanders. His current research interests are quantum computation and formal methods.