Programming Research Group

LOGICAL REVERSIBILITY

Paolo Zuliani

PRG-TR-11-00

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD

Abstract

A technique is developed that transforms any program in the probabilistic Guarded Command Language (*pGCL*) into an equivalent but reversible program. The result extends previous works firstly by considering a general purpose programming language (*pGCL*), and secondly by dealing with "demonic" nondeterminism and probability. A formal definition of logical reversibility is given and the expectation-transformer semantics for *pGCL* is used to prove the result. The technique presented has a direct application 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 before execution.

In the context of logical reversibility we are interested in the logical model (*e.g.* Turing machine, λ-calculus, Guarded Command Language [5], *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 [8] 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 [9] 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 [6].

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 [15]. 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, which in particular includes nondeterminism and probability. In particular, we shall give rules that transform *probabilistic Guarded Command Language (pGCL)* [12] programs to equivalent but reversible $pGCL$ ones. 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. Its importance arises as a result of the desire to compile general-purpose programming languages (*e.g.* [14]) 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: therefore given a quantum mechanical operator U there always exists the 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 (quantum mechanical operators are represented by matrices, so $\circ$ is in fact the standard matrix multiplication). This means that in principle any quantum computation can be reversed. On the other hand, classical computations are not reversible, 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. $qGCL$ [14] 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 [12], [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 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 one. When they will be ready, quantum processors will likely 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 [16]. At this point all the unitary transformations can be passed to the part of the compiler which will output the code for the chosen quantum architecture.

3 Previous work

Lecerf [9] 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 [6], is a 3-bit logic gate which is both reversible and conservative: that is input and output have the same number of bits at 1. Reversibility and conservativity are two independent properties: however we are not interested in conservativity, as it does not seem to play a role for our purposes.

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 [4], or the billiard-ball computer [6]. 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

A GCL program is a sequence of assignments, **skip**, and **abort** manipulated by the standard constructors of sequential composition, conditional selection, repetition and nondeterministic choice [5]. 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, $R \,{}^{\circ}_{9}\, S$, 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, $R \sqcap S$, 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]. 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

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

The cartesian product $T_1 \times T_2 \times \dots \times T_n$ 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 [7] or in terms of expectation transformers [13]. 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):

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

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 $R \lhd \mathit{cond} \rhd S$ is a special case of probabilistic choice: it is just $R\ {}_{[\mathit{cond}]}{\oplus}\ S$. 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:

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

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

$$\boxed{\begin{array}{rcl} wp.(P(\textbf{value } f{:}T\backslash E)).q & := & (wp.\mathit{body}.q)[f\backslash E] \\[1em] wp.(P(\textbf{result } f{:}T\backslash v)).q & := & [(\forall f \bullet wp.\mathit{body}.q[v\backslash f])] \\[1em] wp.(P(\textbf{value result } f{:}T\backslash v)).q & := & (wp.\mathit{body}.q)[f\backslash E] \end{array}}$$

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 $wp.P.\mathit{true} = \mathit{true}$, 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:*

$$(\textbf{skip} \operatorname{⨟} A) \simeq (A \operatorname{⨟} \textbf{skip}) \simeq A$$
$$(A \sqcap B) \operatorname{⨟} C \simeq (A \operatorname{⨟} C) \sqcap (B \operatorname{⨟} C)$$
$$(A\ {}_p{\oplus}\ B) \operatorname{⨟} C \simeq (A \operatorname{⨟} C)\ {}_p{\oplus}\ (B \operatorname{⨟} C)$$

4.3 Reversible programs

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

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

$$(R \operatorname{⨟} S) \simeq \textbf{skip}.$$

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:*

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

Proof. Consider a program R whose form is, without loss of generality:

$$R = R_0 \fatsemi R_1 \fatsemi \ldots \fatsemi R_{n-1}$$

where $n{:}\mathbb{N}$ and the R_i's are statements. Now we reason:

R reversible

$\Rightarrow$ def of reversibility

$\forall i{:}\{0, \ldots, n-1\}, \exists S_i \bullet (R_i \fatsemi S_i) \simeq \mathbf{skip}$

$\Rightarrow$ programming law

$\forall i{:}\{0, \ldots, n-1\}, \exists S_i \bullet (R_i \fatsemi \mathbf{skip} \fatsemi S_i) \simeq \mathbf{skip}$

$\Rightarrow$ hypothesis

$\forall i{:}\{0, \ldots, n-2\}, \exists S_i \bullet (R_i \fatsemi R_{i+1} \fatsemi S_{i+1} \fatsemi S_i) \simeq \mathbf{skip}$

$\Rightarrow$ transitivity of $\simeq$

$(R_0 \fatsemi R_1 \fatsemi \ldots \fatsemi R_{n-1} \fatsemi S_{n-1} \fatsemi \ldots \fatsemi S_1 \fatsemi S_0) \simeq \mathbf{skip}$

$\Rightarrow$ logic

$(\exists S := (S_{n-1} \fatsemi \ldots \fatsemi S_1 \fatsemi S_0) \bullet (R \fatsemi S) \simeq \mathbf{skip}$

□

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

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

Proof.

$$
\begin{array}{ll}
R \text{ reversible} & \\
\Rightarrow & \text{previous lemma} \\
\exists S \bullet (R \,;\, S) \simeq \textbf{skip} & \\
\Rightarrow & \text{def of equivalence} \\
\exists S \bullet \forall q{:}\mathcal{Q} \bullet wp.(R \,;\, S).q \equiv q & \\
\Rightarrow & wp\text{-semantics} \\
\exists S \bullet \forall q{:}\mathcal{Q} \bullet wp.R.(wp.S.q) \equiv q & \\
\Rightarrow & \text{special case } q \equiv \mathbf{1} \\
\exists S \bullet wp.R.(wp.S.\mathbf{1}) \equiv \mathbf{1} & \\
\Rightarrow & \text{logic} \\
\exists S \bullet wp.R.(wp.S.\mathbf{1}) \Leftarrow \mathbf{1} & \\
\Rightarrow & \text{take } q := wp.S.\mathbf{1} \\
\exists q{:}\mathcal{Q} \bullet wp.R.q \Leftarrow \mathbf{1} & \\
\Rightarrow & \text{monotonicity of } wp.R \\
wp.R.\mathbf{1} \Leftarrow \mathbf{1} & \\
\Rightarrow & \mathbf{1} \text{ greatest element of } (\mathcal{Q}, \Rrightarrow) \\
wp.R.\mathbf{1} \equiv \mathbf{1} &
\end{array}
$$

□

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):

$$
\begin{array}{l}
\textbf{module} \quad \textit{stack} \\
\quad \textbf{var } x{:}seq\ D \bullet \\
\quad \textbf{proc push } (\textbf{value } f{:}D) := \ x : [x = f{:}x_0] \\
\quad \textbf{proc pop } (\textbf{result } f{:}D) := \ x, f : [x_0 = f{:}x] \\
\textbf{end}
\end{array}
$$

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:*

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

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

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

$\equiv$ 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:*

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

Proof. We shall go straight into calculation:

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

$\equiv$ previous lemma

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

$\equiv$ assignment semantics

$q[v \backslash v]$

$\equiv$ syntactical substitution

q

$\equiv$ **skip** semantics

$wp.\mathbf{skip}.q$

□

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 $(P_r \,;\, P_i) \simeq$ **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:$pGCL \to pGCL$* 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:*

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

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
$R \sqcap S$	**push** b ; (R_r ; **push** T) $\sqcap$ (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:

$$\textbf{push}\ v_1 \,;\, \textbf{push}\ v_2 \,;\, \ldots\ldots \,;\, \textbf{push}\ v_{n-1} \,;\, \textbf{push}\ v_n$$

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:

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

We reason:

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

$\equiv$ 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:

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

We shall simplify the LHS:

$wp.(R_r \,;\, S_r \,;\, S_i \,;\, R_i).q$

$\equiv$ 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:

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

$$Q_i := \left(\begin{array}{l} \mathbf{pop}\ b \,; \\ (R_i \triangleleft b \triangleright S_i) \,; \\ \mathbf{pop}\ b \end{array} \right)$$

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

$$Q_r \fatsemi Q_i$$

$$\simeq \qquad \text{lemma 4.1}$$

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

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

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

$$\simeq \qquad \text{associativity}$$

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

$$\simeq \qquad \text{lemma 4.4}$$

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

$$\simeq \qquad \text{associativity}$$

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

$$\simeq \qquad \text{conditional selection}$$

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

$$\simeq \qquad \text{induction hypothesis}$$

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

$$\simeq \qquad \text{programming law}$$

$$\mathbf{pop}\ b$$

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

$$Q_r \fatsemi Q_i$$

$$\simeq$$

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

$$\simeq \qquad \text{programming law}$$

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

$$\simeq \qquad \text{corollary 4.5}$$

$$\mathbf{skip}$$

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, so is given in the appendix.

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

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

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

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

We reason:

$wp.(Q_r(a) \mathbin{;} Q_i(a)).q$

$\equiv$ sequential composition

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

$\equiv$ def of Q_i and substitution

$wp.Q_r(a).((wp.(\textit{body}_i).q[a\backslash f])[f\backslash a])$

$\equiv$ def of Q_r and substitution

$wp.(\textit{body}_r).((wp.(\textit{body}_i).q[a\backslash f])[f\backslash a])[a\backslash g])[g\backslash a]$

$\equiv$ logic

$(wp.(\textit{body}_r).(wp.(\textit{body}_i).q[a\backslash f]))[f\backslash g])$

$\equiv$ induction hyothesis

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

$\equiv$ logic

$q[a\backslash g]$

$\equiv$ 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:

$$\begin{array}{ll}
 & P_r \\
= & \hfill \text{def of } P_r \\
 & \left(\begin{array}{l} \textbf{var}\;\; c{:}\mathbb{B} \bullet \\ \quad c := T\,\fatsemi \\ \quad \textbf{while}\; c\; \textbf{do} \\ \quad\quad (\textbf{skip})\; {}_{\frac{1}{2}}\!\oplus\; (c := F) \\ \quad \textbf{od} \end{array}\right)_r \\
= & \hfill \text{seq composition and local block} \\
 & \begin{array}{l} \textbf{var}\;\; c{:}\mathbb{B} \bullet \\ \quad (c := T)_r\,\fatsemi \\ \quad \left(\begin{array}{l} \textbf{while}\; c\; \textbf{do} \\ \quad (\textbf{skip})\; {}_{\frac{1}{2}}\!\oplus\; (c := F) \\ \textbf{od} \end{array}\right)_r \end{array} \\
= & \hfill \text{assignment and loop} \\
 & \begin{array}{l} \textbf{var}\;\; c, b{:}\mathbb{B} \bullet \\ \quad \textbf{push}\; c\,\fatsemi \\ \quad c := T\,\fatsemi \\ \quad \textbf{push}\; b\,\fatsemi\, \textbf{push}\; F\,\fatsemi \\ \quad \textbf{while}\; c\; \textbf{do} \\ \quad\quad \Big((\textbf{skip})\; {}_{\frac{1}{2}}\!\oplus\; (c := F)\Big)_r\,\fatsemi \\ \quad\quad \textbf{push}\; T \\ \quad \textbf{od} \end{array} \\
= & \hfill \text{probabilistic choice}
\end{array}$$

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

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

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

and we see that $(P_r\,⨾\,P_i) \simeq$ **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 of 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.

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.

References

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

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

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

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

[7] He Jifeng, Annabelle McIver, and Karen Seidel. Probabilistic models for the guarded command language. *Science of Computer Programming*, 28:171–192, 1997.

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

[9] 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, 1973.

[10] Carroll Morgan. *Programming from Specifications*. Prentice-Hall International, 1994.

[11] Carroll Morgan. Proof rules for probabilistic loops. In *Proceedings of the BCS-FACS 7th Refinement Workshop*. Springer Verlag, July 1996.

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

[13] Carroll Morgan, Annabelle McIver, and Karen Seidel. Probabilistic predicate transformers. *ACM Transactions on Programming Languages and Systems*, 18(3):325–353, May 1996.

[14] J.W. Sanders and Paolo Zuliani. Quantum programming. *Mathematics of Program Construction 2000, Springer LNCS*, 1837:80–99, 2000.

[15] Peter W. Shor. Algorithms for quantum computation: Discrete log and factoring. In *Proceedings of the 35th Annual symposium on the Foundations of Computer Science*, pages 20–22, 1994.

[16] Paolo Zuliani. *DPhil Thesis.* Oxford University. In preparation.

A Appendix

Before going into the proof for the iteration construct we shall introduce relational semantics for GCL and extend it, according to our needs, to $pGCL$. A more general treatment of relational models for $pGCL$ is given in [7].

A.1 Relational semantics

A *nondeterministic* program is modelled relationally by a relation over its state space, and it is viewed as a *state transformer*, in which the input is coded in a particular *initial* state and the output, if the program halts, in a particular *final* state.

To address the problem of non termination the state space X of a program is usually augmented with the $\perp$ element, thus forming a new state space $X_\perp := X \cup \{\perp\}$.

Definition A.1. Let P be a GCL program and X its state space. The ***semantics*** of P is the total relation $[P]{:}X_\perp \longleftrightarrow X_\perp$ defined by:

$$x\,[P]\,y := \text{program } P \text{ when started in state } x \text{ may terminate in state } y,$$

where $x, y{:}X_\perp$.

In the above definition we have used *may* because of nondeterminism program P is allowed to terminate in more than one state or perhaps also not at all. The expression $x\,[P]\perp$ denotes the fact that P may not terminate from initial state x. Since $\perp$ never occurs as an initial state we may define $\perp[P]\,y$ as we wish: it is however chosen in such a way that $[P]$ satisfies some healthiness conditions which are required for a relation in order to represent a computation (see [7] for a full treatment).

In our case we only deal with terminating programs, therefore making not necessary the extra state $\perp$. The simplest terminating program is **skip**, whose semantics is thus the identity relation on X, ι_X.

We shall use the same model for $pGCL$ as well, thus multiple-valuedness of relations will make no distinction between nondeterminism and probabilism. For our purposes this is not an issue.

For sets A, B, relation $r{:}A \longleftrightarrow B$ and $a{:}A$ we define $r.(\!|a|\!) := \{b{:}B \bullet a\,r\,b\}$. We shall now model the execution of statement S with initial state x, that is S halts in one of its final states. This can be formalized by $[S].x := y{:}([S].(\!|x|\!))$. Furthermore we can substitute to any statement S the assignment statement $x := ([S].x)$. Again, we do not distinguish between probability and nondeterminism.

The following lemma will be useful later.

Lemma A.1. *Let R, S be statements over state space X. Then for $x{:}X$:*

$$[R \mathbin{;} S].(\!|x|\!) = \bigcup_{y:([R].(\!|x|\!))} [S].(\!|y|\!) \,.$$

Proof. We reason:

$$\begin{aligned}
& [R \mathbin{;} S].(\!|x|\!) \\
= & \qquad\qquad \text{def of } (\!|\cdot|\!) \\
& \{y{:}X \bullet x\,[R \mathbin{;} S]\,y\} \\
= & \qquad\qquad \text{seq composition} \\
& \{y{:}X \bullet \exists z{:}X \bullet x\,[R]\,z \wedge z\,[S]\,y\} \\
= & \qquad\qquad \text{def of } (\!|\cdot|\!) \\
& \{y{:}X \bullet z{:}([R].(\!|x|\!)) \wedge z\,[S]\,y\} \\
= & \qquad\qquad \text{logic} \\
& \textstyle\bigcup_{z:([R].(\!|x|\!))} [S].(\!|z|\!)
\end{aligned}$$

□

For a function $f{:}X \to Y$ and $W \subseteq X$, $f{\restriction}W{:}W \to Y$ denotes the *restriction* of f to W defined by:

$$\forall w{:}W \bullet f{\restriction}W.w := f.w \,.$$

For a statement S and $i{:}\mathbb{N}$ we define the *iterated* statement S^i by:

$$\begin{aligned}
S^0 &:= \textbf{skip}, \\
S^i &:= S \mathbin{;} S^{i-1} \,.
\end{aligned}$$

A.2 Iteration

With respect to iteration we have to prove that:

$$(W_r \mathbin{;} W_i) \simeq \textbf{skip}$$

where:

$$W_r := \left(\begin{array}{l} \textbf{push } b \mathbin{;} \textbf{push } F \mathbin{;} \\ \textbf{while } c \textbf{ do} \\ \quad S_r \mathbin{;} \\ \quad \textbf{push } T \\ \textbf{od} \end{array}\right), \; W_i := \left(\begin{array}{l} \textbf{pop } b \mathbin{;} \\ \textbf{while } b \textbf{ do} \\ \quad S_i \mathbin{;} \\ \quad \textbf{pop } b \\ \textbf{od} \mathbin{;} \\ \textbf{pop } b \end{array}\right)$$

The proof will be split in two parts: first we shall prove that W_r always terminates then, using Hoare logic, we prove the correctness of the composition of the two loops.

We point out that the next lemma might be easily proved using the *probabilistic variant rule* [11], but that rule is complete only for finite state spaces.

Let X be the state space of statement S; program W_r's state space is then $X_r := (X \times \mathbb{B} \times H)$, where $H := stack.(X \cup \mathbb{B})$.

Lemma A.2. *Consider a terminating loop $L :=$ (**while** c **do** S **od**). Then loop L_r (reverse of L) is terminating as well.*

Proof. We have to prove that for expectation $\mathbf{1}$ over X_r, we have $wp.L.\mathbf{1} \equiv wp.L_r.\mathbf{1} \equiv \mathbf{1}$. Consider the two program transformers $l, l_r{:}pGCL \to pGCL$ defined by:

$$l.P := (S \fatsemi P) \triangleleft c \triangleright \mathbf{skip}$$
$$l_r.P := (S_r \fatsemi \mathbf{push}\ T \fatsemi P) \triangleleft c \triangleright \mathbf{skip}.$$

We shall show that:

$$\forall n{:}\mathbb{N} \bullet wp.(l^n.\mathbf{abort}).\mathbf{1} \equiv wp.(l_r^n.\mathbf{abort}).\mathbf{1}$$

therefore:

$$\mathbf{1} \equiv wp.L.\mathbf{1} \equiv (\sqcup\, n{:}\mathbb{N}\ wp.(l^n.\mathbf{abort}).\mathbf{1}) \equiv (\sqcup\, n{:}\mathbb{N}\ wp.(l_r^n.\mathbf{abort}).\mathbf{1}) \equiv wp.L_r.\mathbf{1}$$

We do not need to run the limits beyond the natural numbers, as we are dealing with continous statements only; we shall make use of the induction principle.

The base case $n = 0$ is trivially true:

$$l^0.\mathbf{abort} = \mathbf{abort} = l_r^0.\mathbf{abort}.$$

Consider now the successor case for $n{:}\mathbb{N}$:

$wp.(l_r^{n+1}.\mathbf{abort}).\mathbf{1}$

$\equiv$ def of l_r

$wp.((S_r \fatsemi \mathbf{push}\ T \fatsemi l_r^n.\mathbf{abort}) \triangleleft c \triangleright \mathbf{skip}).\mathbf{1}$

$\equiv$ semantics of conditional

$[\neg c] * wp.\mathbf{skip}.\mathbf{1} + [c] * wp.(S_r \fatsemi \mathbf{push}\ T \fatsemi l_r^n.\mathbf{abort}).\mathbf{1}$

$\equiv$ **skip** and seq. composition

$[\neg c] * \mathbf{1} + [c] * wp.S_r.wp.(\mathbf{push}\ T).wp.(l_r^n.\mathbf{abort}).\mathbf{1}$

$\equiv$ induction hypothesis

$[\neg c] * \mathbf{1} + [c] * wp.S_r.wp.(\mathbf{push}\ T).wp.(l^n.\mathbf{abort}).\mathbf{1}$

$\equiv$ l_n acts on X only and logic

$[\neg c] * \mathbf{1} + [c] * wp.S_r.wp.(\mathbf{push}\ T).(wp.(l^n.\mathbf{abort}).(\mathbf{1}\restriction X) * (\mathbf{1}\restriction \mathbb{B} \times H))$

$\equiv$ logic

$[\neg c] * \mathbf{1} + [c] * wp.S_r.(wp.(l^n.\mathbf{abort}).(\mathbf{1}\restriction X) * wp.(\mathbf{push}\ T).(\mathbf{1}\restriction \mathbb{B} \times H))$

$\equiv$ **push** terminating

$[\neg c] * \mathbf{1} + [c] * wp.S_r.(wp.(l^n.\mathbf{abort}).(\mathbf{1}\restriction X) * (\mathbf{1}\restriction \mathbb{B} \times H))$

$\equiv$ logic

$[\neg c] * \mathbf{1} + [c] * (\mathbf{1}\restriction \mathbb{B} \times H) * wp.S_r.(wp.(l^n.\mathbf{abort}).(\mathbf{1}\restriction X))$

$\equiv$ S_r behaves like S on X variables

$[\neg c] * \mathbf{1} + [c] * (\mathbf{1}\restriction \mathbb{B} \times H) * wp.S.(wp.(l^n.\mathbf{abort}).(\mathbf{1}\restriction X))$

$\equiv$ logic

$[\neg c] * \mathbf{1} + [c] * wp.S.wp.(l^n.\mathbf{abort}).\mathbf{1}$

$\equiv$ semantics of conditional

$wp.((S \mathbin{;} l^n.\mathbf{abort}) \triangleleft c \triangleright \mathbf{skip}).\mathbf{1}$

$\equiv$ def of l

$wp.(l^{n+1}.\mathbf{abort}).\mathbf{1}$

$\square$

In order to simplify the subsequent proofs we shall make use of a counter variable k, which will be incremented at every iteration of the reverse loop and decremented at every iteration of the inverse loop. Loops W_r', W_i' are :

$$W_r' := \left(\begin{array}{l} \mathbf{push}\ b \mathbin{;} \mathbf{push}\ F \mathbin{;} \\ \mathbf{while}\ c\ \mathbf{do} \\ \quad S_r \mathbin{;} \\ \quad \mathbf{push}\ T \mathbin{;} \\ \quad k := k+1 \\ \mathbf{od} \end{array}\right),\ W_i' := \left(\begin{array}{l} \mathbf{pop}\ b \mathbin{;} \\ \mathbf{while}\ b\ \mathbf{do} \\ \quad S_i \mathbin{;} \\ \quad \mathbf{pop}\ b \mathbin{;} \\ \quad k := k-1 \\ \mathbf{od} \mathbin{;} \\ \mathbf{pop}\ b \end{array}\right)$$

Since neither S_r nor S_i modify k, we can leave it out of the program's state without affecting our arguments; we shall also suppose for convenience that $k = 0$ before entering W_r'.

Lemma A.3. *Let $y = (x, b, h){:}X_r$ represents W_r's state. Then*

$$\mathbf{while}\ c\ \mathbf{do}\ (S_r \mathbin{;} \mathbf{push}\ T) \mathbin{;} k := k+1\ \mathbf{od}$$

has invariant I:

$$I := y{:}([(S_r \mathbin{;} \mathbf{push}\ T)^k].(\!|y_0|\!))$$

where $y_0 = (x_0, b_0, h_0)$.

Proof. Using Hoare's logic we prove:

$$(I \wedge c)\, S_r \,{}_9^{\circ}\, \textbf{push } T\, (I).$$

Since $[\textbf{skip}] = \iota_{X_r}$, I holds before entering the loop. Now we reason:

I

$\Leftrightarrow$ def of I

$y{:}([(S_r \,{}_9^{\circ}\, \textbf{push } T)^k].(\!|y_0|\!))$

$\Leftarrow$ backward substitution $k := k+1$

$y{:}([(S_r \,{}_9^{\circ}\, \textbf{push } T)^{k+1}].(\!|y_0|\!))$

$\Leftarrow$ backward sub $y := ([S_r \,{}_9^{\circ}\, \textbf{push } T].y)$

$([S_r \,{}_9^{\circ}\, \textbf{push } T].y){:}([(S_r \,{}_9^{\circ}\, \textbf{push } T)^{k+1}].(\!|y_0|\!))$

$\Leftrightarrow$ associativity of seq composition

$([S_r \,{}_9^{\circ}\, \textbf{push } T].y){:}([(S_r \,{}_9^{\circ}\, \textbf{push } T)^k \,{}_9^{\circ}\, (S_r \,{}_9^{\circ}\, \textbf{push } T)].(\!|y_0|\!))$

$\Leftarrow$ lemma A.1

$([S_r \,{}_9^{\circ}\, \textbf{push } T].y){:}\left(\bigcup_{z:([(S_r {}_9^{\circ} \textbf{push } T)^k].(\!|y_0|\!))}[(S_r \,{}_9^{\circ}\, \textbf{push } T)].(\!|z|\!)\right)$

$\Leftarrow$ logic

$y{:}([(S_r \,{}_9^{\circ}\, \textbf{push } T)^k].(\!|y_0|\!))$

$\Leftarrow$ def of I and logic

$(I \wedge c)$

□

The **push** F statement before the loop just changes the initial state of the program to $y_0 = (x_0, b_0, F{:}h_0)$ which we therefore assume in the subsequent proofs.

Before entering the inverse loop there is a **pop** b statement. We have to prove that:

$$(I)\ \textbf{pop } b\ (y{:}([(S_r \,{}_9^{\circ}\, \textbf{push } T)^k \,{}_9^{\circ}\, \textbf{pop } b].(\!|y_0|\!))).$$

We reason:

$(y{:}([(S_r \,{}_9^{\circ}\, \textbf{push } T)^k \,{}_9^{\circ}\, \textbf{pop } b].(\!|y_0|\!)))$

$\Leftarrow$ backward sub $y := ([\textbf{pop } b].y)$

$([\textbf{pop } b].y){:}([(S_r \,{}_9^{\circ}\, \textbf{push } T)^k \,{}_9^{\circ}\, \textbf{pop } b].(\!|y_0|\!)))$

$\Leftrightarrow$ lemma A.1

$([\textbf{pop } b].y){:}\left(\bigcup_{z:[(S_r {}_9^{\circ} \textbf{push } T)^k].(\!|y_0|\!)}[\textbf{pop } b].(\!|z|\!)\right)$

$\Leftrightarrow$ **pop** deterministic

$([\mathbf{pop}\ b].y) \colon \left(\bigcup_{z:[(S_r \,⨟\, \mathbf{push}\ T)^k].(\!|y_0|\!)} [\mathbf{pop}\ b].z \right)$

$\Leftrightarrow$ logic

$y \colon ([(S_r \,⨟\, \mathbf{push}\ T)^k].(\!|y_0|\!))$

$\Leftrightarrow$ def of I

I

We can now pass to the last lemma.

Lemma A.4. *The inverse loop:*

$$\mathbf{while}\ b\ \mathbf{do}\ (S_i \,⨟\, \mathbf{pop}\ b) \,⨟\, k := k - 1\ \mathbf{od}$$

has invariant J:

$$J := y \colon ([(S_r \,⨟\, \mathbf{push}\ T)^k \,⨟\, \mathbf{pop}\ b].(\!|y_0|\!))$$

Proof. We reason:

J

$\Leftrightarrow$ def of J

$y \colon ([(S_r \,⨟\, \mathbf{push}\ T)^k \,⨟\, \mathbf{pop}\ b].(\!|y_0|\!))$

$\Leftarrow$ backward substitution $k := k - 1$

$y \colon ([(S_r \,⨟\, \mathbf{push}\ T)^{k-1} \,⨟\, \mathbf{pop}\ b].(\!|y_0|\!))$

$\Leftarrow$ backward sub of $y := ([S_i \,⨟\, \mathbf{pop}\ b].y)$

$([S_i \,⨟\, \mathbf{pop}\ b].y) \colon ([(S_r \,⨟\, \mathbf{push}\ T)^{k-1} \,⨟\, \mathbf{pop}\ b].(\!|y_0|\!))$

$\Leftarrow$ logic and induction hypothesis

$([S_i \,⨟\, \mathbf{pop}\ b].y) \colon ([(S_r \,⨟\, \mathbf{push}\ T)^k \,⨟\, \mathbf{pop}\ b \,⨟\, S_i \,⨟\, \mathbf{pop}\ b].(\!|y_0|\!))$

$\Leftrightarrow$ associativity of seq composition

$([S_i \,⨟\, \mathbf{pop}\ b].y) \colon ([((S_r \,⨟\, \mathbf{push}\ T)^k \,⨟\, \mathbf{pop}\ b) \,⨟\, (S_i \,⨟\, \mathbf{pop}\ b)].(\!|y_0|\!))$

$\Leftrightarrow$ lemma A.1

$([S_i \,⨟\, \mathbf{pop}\ b].y) \colon \left(\bigcup_{z:[(S_r \,⨟\, \mathbf{push}\ T)^k \,⨟\, \mathbf{pop}\ b].(\!|y_0|\!)} [S_i \,⨟\, \mathbf{pop}\ b].(\!|z|\!) \right)$

$\Leftarrow$ logic

$y \colon ([(S_r \,⨟\, \mathbf{push}\ T)^k \,⨟\, \mathbf{pop}\ b].(\!|y_0|\!))$

$\Leftrightarrow$ def of J and logic

$(J \wedge c)$

$\square$

It is easy to see that the loop terminates after exactly k iterations: as long as $k > 0$ we have $b = T$. When $k = 0$ the postcondition is therefore $y{:}([\mathbf{pop}\ b].(\!|y_0|\!))$, which is equivalent to $y := ([\mathbf{pop}\ b].y_0)$ as **pop** is deterministic. The postcondition can be further simplified:

$[\mathbf{pop}\ b].y_0$

$=$ def of y_0

$[\mathbf{pop}\ b].(x_0, b_0, F{:}h_0)$

$=$ semantics of **pop**

(x_0, F, h_0)

and the loop then terminates since $b = F$. Now, defining C to be the program:

$$C := \left(\begin{array}{l} \mathbf{push}\ F\,; \\ \mathbf{while}\ c\ \mathbf{do}\ S_r\ ;\ \mathbf{push}\ T\ \mathbf{od}; \\ \mathbf{pop}\ b; \\ \mathbf{while}\ b\ \mathbf{do}\ S_i\ ;\ \mathbf{pop}\ b\ \mathbf{od}; \end{array}\right)$$

from Lemma A.3, A.4 and remark above, we get that $C \simeq (b := F)$. Therefore:

$(W_r\ ;\ W_i)$

$\simeq$ def of W_r, W_i

$\mathbf{push}\ b\ ;\ C\ ;\ \mathbf{pop}\ b$

$\simeq$ remark above

$\mathbf{push}\ b\ ;\ b := F\ ;\ \mathbf{pop}\ b$

$\simeq$ see proof for assignment

skip

and this completes the proof for the main theorem.