Compiling quantum programs

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

Abstract

In this paper we study a possible compiler for a high-level imperative programming language for quantum computation, the quantum Guarded-Command Language (qGCL). It is important because it liberates us from thinking of quantum algorithms at the data-flow level, in the same way as happened for standard computation a few decades ago.

We make use of the normal-form approach to compiler design, introduced by Hoare, Jifeng and Sampaio. In this approach a source program is transformed, by means of algebraic manipulations, into a particular form which can be directly executed by a target machine. This entails the definition of a simple quantum hardware architecture, derived from Hoare *et al.*'s computing model.

Our work provides a general framework for the construction of a compiler for qGCL, focusing mainly on the correctness of the design. Here we do not deal with other topics such as efficiency of compiled code, factorisation of unitary transformations and compilation of quantum data structures.

1 Introduction

In this paper we study a possible compiler for qGCL [21], a general-purpose programming language for quantum computation. qGCL has been successfully used to describe and reason about all known quantum algorithms and also to derive one of them (the Deutsch-Jozsa algorithm) from its specification [21]. Furthermore, qGCL has been used to describe and reason about peculiar quantum features, such as nonlocality and counterfactual computation [14, 26, 28].

Our work on qGCL has shown how to raise the level of abstraction from data-flow reasoning about quantum computation to that used in normal "software engineering". Indeed the benefit of that work is its use of abstract specification and of permitting data refinements to be used in implementing a specification using quantum procedures. But these benefits can be realised only if there is an accompanying method of compilation from qGCL into a data-flow model. That is what justify the use of high-level languages in Computer Science in general. In this paper such a method is provided.

qGCL was developed as a superset of the *probabilistic* Guarded-Command Language (pGCL), a high-level imperative programming language which can describe both classical and probabilistic computation [17, 15]. 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 [17, 15, 12]), which include program refinement, data refinement and combination of specifications with code.

In the design of the compiler we make use of the normal-form approach introduced by Hoare *et al.* [10]. Their idea is to compile a Guarded-Command Language program by transforming it, by means of algebraic manipulations, into a particular form which can be directly executed by a target machine. At each step high-level constructs are refined into low-level constructs. The algebraic laws used in the process are given by the refinement calculus associated with pGCL and therefore the correctness of the compiler follows from the soundness of each of the algebraic laws. Thus we design a compiler which is correct by construction.

The biggest advantage of this approach is its modularity: at each refining step we may postpone implementation decisions to a later, more appropriate, time. In an extreme case we may even decide not to provide any implementation for a particular stage, and this will not undermine the correctness of the compiler. That observation will prove very useful in our case as the compiler for qGCL we are going to describe will not provide any module for factorising unitary transformations in terms of "basic" quantum operators, though we pose the requirements for such a module. Therefore we do not need to specify the implementation of that module and we are not bound to a particular one, so that if a more efficient factorising method is developed we can use it immediately in our compiler.

Another advantage of the normal-form approach is that it provides a single framework, the refinement calculus, for developing programs, reasoning about programs and designing compilers, thereby providing a unique bridge which connects specifications, high-level (source) programs and low-level implementations.

With respect to quantum architecture we devise a simple generalisation of standard architecture which fits into Hoare *et al.*'s hardware model [10]. That architecture imposes loose requirements for the quantum hardware, since at the time of writing it is not clear on which architecture quantum computers will eventually be built. These requirements are equivalent to those found in other quantum architectures (for example the quantum circuit model [5, 2, 18]) and it seems that they represent what can currently be assumed about quantum architectures. However, the technique presented in this paper will be useful whatever quantum architecture is decided upon.

We start exposition by giving a short introduction to quantum programming with qGCL. Next we apply Hoare *et al.*'s normal-form approach [10] for developing a compiler for qGCL. Their work considered Dijkstra's plain Guarded-Command Language, thus omitting probabilism, so we expand the normal-form approach to cover pGCL and in turn qGCL. We follow the order of exposition of Hoare *et al.*'s paper and make our changes when needed. The compilation process is split into three main steps: simplification of expressions, reduction to normal form (control structure elimination) and machine state introduction.

2 Quantum programming

We give here a short presentation of the features of qGCL (a full introduction can be found in [21, 28]). qGCL is an extension of pGCL [17], which in turn extends Dijkstra's Guarded-Command Language GCL [7] with probabilism.

We start the exposition with the data types required by quantum computation, then we introduce pGCL and successively qGCL's quantum-related constructs. We advise the readers that all the relevant quantum-mechanical definitions and concepts are collected and summarised in Appendix A.

2.1 Quantum types

We present a transformation q which maps a classical data type to its quantum analogue. All our later examples require the application of q only to registers, so we restrict ourselves to that case.

We define the type $\mathbb{B} \mathrel{\widehat{=}} \{0, 1\}$, which we will treat as booleans or bits, depending on convenience. A classical register of size $n{:}\mathbb{N}$ is a vector of n booleans. The type of all registers of size n is then defined to be the set of boolean-valued functions on $\{0, 1, \ldots, n-1\}$:

$$\mathbb{B}^n \mathrel{\widehat{=}} \{0, 1, \ldots, n-1\} \longrightarrow \mathbb{B} \,.$$

The quantum analogue of $\mathbb{B}^n$ is the set of complex-valued functions on $\mathbb{B}^n$ whose squared modulus sum to 1:

$$q(\mathbb{B}^n) \mathrel{\widehat{=}} \{\chi{:}\mathbb{B}^n \longrightarrow \mathbb{C} \quad | \sum_{x:\mathbb{B}^n} |\chi(x)|^2 = 1\}$$

where we denoted the modulus of a complex number z by $|z|$. An element of $q(\mathbb{B})$ is called a *qubit* [22] and that of $q(\mathbb{B}^n)$ a *qureg* [21].

Classical state is embedded in its quantum analogue by the Dirac delta function:

$$\begin{aligned} &\delta{:}\mathbb{B}^n \longrightarrow q(\mathbb{B}^n) \\ &\delta_x(y) \mathrel{\widehat{=}} (y = x) \,. \end{aligned}$$

The range of δ, $\{\delta_x \mid x{:}\mathbb{B}^n\}$, forms a *basis* (called the *standard basis*) for quantum states, that is:

$$\forall \chi{:}q(\mathbb{B}^n) \bullet \chi = \sum_{x:\mathbb{B}^n} \chi(x)\delta_x \,.$$

The Hilbert space $\mathbb{B}^n \longrightarrow \mathbb{C}$ (with the structure making it isomorphic to $\mathbb{C}^{2^n}$) is called the *enveloping space* of $q(\mathbb{B}^n)$.

2.2 Probabilistic language pGCL

A Guarded-Command Language program is a sequence of assignments, **skip** and **abort** manipulated by the standard constructors of sequential composition, conditional selection, repetition and nondeterministic choice [7]. Assignments 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. We list all the other pGCL basic statements and constructors, though we shall use only a few of them in this paper.

- Basic statements
  - **skip**, which always terminates doing nothing;
  - **abort**, which models divergence.
- Constructors
  - sequential composition, $R \fatsemi S$, which firstly executes R and then, if R has terminated, executes S;
  - iteration, **while** *cond* **do** S **od**, 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** $P(param) \mathrel{\widehat{=}} body$, where *body* is a valid pGCL statement 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 [16]. 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;
  - the local block enclosing S is denoted by **var** $\bullet\, S$ **rav**.

For the probabilistic combinator ${}_p\oplus$ we allow p to be an expression whose evaluation returns a real in $[0,1]$. Both nondeterministic and probabilistic choice may be written using a prefix notation, in case the branches are more than two. If $[\ P_j \bullet 0 \leqslant j < m\ ]$ denotes a finite indexed family of programs then

$$\sqcap [\ P_j \bullet 0 \leqslant j < m\ ]$$

chooses nondeterministically to execute one of the programs P_i. For probabilistic choice let $[\ (P_j, r_j) \bullet 0 \leqslant j < m\ ]$ be a finite indexed family of (program, number) pairs with

$\sum_{0 \leqslant j < m} r_j = 1$, then the probabilistic choice in which P_j is chosen with probability r_j is written in prefix form

$$\oplus[\ P_j \ @ \ r_j \bullet 0 \leqslant j < m\ ] \tag{1}$$

(whose advantage is to avoid the normalising factors required by nested infix form).

If E is a finite non empty set of expressions and x a program variable, then in GCL the assignment $x :\in E$ denotes the nondeterministic choice over all individual assignments of elements of E to x. In pGCL that choice is interpreted to occur with *uniform* probability.

Semantics for pGCL can be given either relationally [12] or in terms of expectation transformers [15]. A Galois connection embeds the relational model in the expectation-transformer model. The latter provides a superior simplicity in calculations and the interested reader can find a short exposition of its main definitions and concepts in Appendix B.

2.3 Reversible programs

In this section we shall give a formal definition of reversibility for pGCL programs, and establish some properties. It is needed by the fact that quantum computation is reversible by its own nature, that is, it always possible to "undo" the computation and return to the conditions prior to execution. We note that, in general, classical computation is not reversible.

Due to space constraints we will limit exposition to the basic concepts only and avoid proofs. A thorough discussion of the reversibility problem in pGCL and all technical details can be found in [27].

Definition 2.1. A statement R is called ***reversible*** iff there exists a program S such that

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

S is called an *inverse* of R; clearly it is not unique. Equality is at the semantic level, as explained in Appendix B.

Definition 2.2. 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 Appendix C for a formal definition of stack, **push** and **pop**)

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

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

Lemma 2.1. *Let R be a reversible program. Then there exists a program S such that:*

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

Again, S is called an *inverse* of R and it is not unique. It can be shown that a reversible program must necessarily terminate for all inputs. The converse is false: the trivial program $x := 0$ does terminate but it is certainly not reversible.

A general technique for transforming any terminating pGCL program into an equivalent but reversible program is given in [27].

2.4 Quantum language qGCL

A *quantum program* is a pGCL program invoking quantum procedures and the resulting language is called qGCL. Quantum procedures can be of three different kinds: *Initialisation* (or state preparation) followed by *Evolution* and finally by *Finalisation* (or observation).

2.4.1 Initialisation

Initialisation is a procedure which simply assigns to its qureg state the uniform square-convex combination of all standard states

$$\forall \chi{:}q(\mathbb{B}^n) \bullet \mathbf{In}(\chi) \mathrel{\widehat{=}} \left(\chi := \frac{1}{\sqrt{2^n}} \sum_{x:\mathbb{B}^n} \delta_x\right).$$

There χ is a **result** parameter.

Initialisation so defined is *feasible* in the sense that it is achievable in practice [6] by initialising the qureg to the classical state $\delta_{\mathbf{0}}$ (where $\mathbf{0}$ denotes the register identically false) and then subjecting that to evolution by the (unitary) Hadamard transform, defined as a tensor power as follows:

$$\begin{aligned} H_n{:}q(\mathbb{B}^n) &\to q(\mathbb{B}^n) \\ H_1(\chi)(x) &\mathrel{\widehat{=}} \frac{1}{\sqrt{2}}(\chi(0) + (-1)^x\chi(1)) \\ H_{n+1} &\mathrel{\widehat{=}} H_n \otimes H_1 \end{aligned}$$

where exponentiation of bits is standard $(-1)^x = -1 \triangleleft x \triangleright 1$. In our language Initialisation could be defined as:

$$\mathbf{proc\ In}\,(\mathbf{result}\ \chi) \mathrel{\widehat{=}} (\chi := \delta_{\mathbf{0}} \mathbin{;} \chi := H(\chi))$$

where H is the Hadamard transform of appropriate size.

For example on $q(\mathbb{B})$, after initialisation, evolution by the Hadamard transformation H_1 results in $\chi = \delta_0$ (because H_1 is not only unitary but equal to its own adjoint and so self-inverse). Thus our definition of initialisation does not exclude setting state to equal δ_0 (or any other standard state for that matter).

2.4.2 Evolution

Quantum-mechanical systems evolve over time under the action of *unitary* transformations (see Appendix A for the definition). *Evolution* thus consists of iteration of unitary transformations on quantum state. (It is thought of, after initialisation, as achieving all superposed evolutions simultaneously, which provides much of the reason for quantum computation's efficiency.) In qGCL unitary evolution may be introduced in two forms: explicit (unitary) transformations on quantum state and procedures.

We have already given an example of explicit unitary transformation: the Hadamard transform defined in section 2.4.1. Evolution of qureg χ under unitary operator U is described in the form:

$$\chi := U(\chi).$$

Since unitary operators are homogenous, it follows that any assignment of the type $\chi := U(\psi)$ is invalid if (syntatically) $\chi \neq \psi$. In particular, when U is the identity operator we have the *no-cloning* theorem (see Appendix D).

The other type of unitary evolution is offered via a particular class of procedures defined by:

$$\textbf{qproc}\ Q\,(param) \mathrel{\widehat{=}} Qbody$$

where *param* is a list of parameters and *Qbody* is a standard pGCL program, constituted by a single local block; *Qbody* shall not contain either procedure calls or declarations. The difference between a standard procedure and a **qproc** is that in the latter code is meant to be executed on a quantum computer.

Calls to a **qproc** follows the custom as for standard procedures, except that standard parameters of the definition are transformed into the corresponding quantum types (we restrict parameters to boolean registers, as quregs are so far the only quantum data type available). Furthermore, all parameters are considered to be **value-result**, as the no-cloning theorem forbids the copy of a qureg. Consider the following **qproc**:

$$\textbf{qproc}\ Dummy\,(\textbf{value}\ a{:}\mathbb{B}^n) \mathrel{\widehat{=}} DummyBody$$

a call to *Dummy* would then be:

$$\begin{array}{l}\textbf{var}\ \chi{:}q(\mathbb{B}^n)\bullet\\ \quad Dummy(\chi)\\ \textbf{rav}.\end{array}$$

Since quantum transformations are reversible, it follows that code of a **qproc** must be reversible (thus terminating, see section 2.3). Again, evolution is feasible: it may be implemented using universal quantum gates [1, 5] and pGCL code in **qproc**'s can be made reversible using the technique illustrated in [27].

2.4.3 Finalisation

Finalisation corresponds to physical observation (or state reduction). Consider a qubit $\chi{:}q(\mathbb{B})$ initialised via quantum procedure **In**: we have that $\chi = \frac{1}{\sqrt{2}}(\delta_0 + \delta_1)$.

Observing χ will force it either to basis state δ_0 or to basis state δ_1. The choice of which state is entirely probabilistic and the probabilities are given by the squared moduli of the values of χ: in our case the probabilities are just $\frac{1}{2}$, since $\chi(0) = \chi(1) = \frac{1}{\sqrt{2}}$. An observation returns also a value which identifies the basis state on which χ has collapsed.

Using the probabilistic combinator of pGCL, quantum observation can be simply written as:

$$(i, \chi := 0, \delta_0)\ {}_{|\chi(0)|^2}\oplus\ (i, \chi := 1, \delta_1)$$

where i is the return value and we recall that $(|\chi(0)|^2 + |\chi(1)|^2) = 1$ since χ is a qubit.

In the general qureg case $\chi = \sum_{0 \leqslant i < n} \chi(j)\delta_j$, observation reduces χ to basis state δ_j with probability $|\chi(j)|^2$. By using the infix form of the probabilistic combinator we model quantum observation of qureg $\chi{:}q(\mathbb{B}^n)$ as:

$$\oplus[\ (i, \chi := j, \delta_j)\ @\ |\chi(j)|^2 \mid 0 \leqslant j < n\ ]\ .$$

In general, an observable is represented by a self-adjoint operator and the measurable values are exactly the eigenvalues of that operator. Equivalently, we can define an observable from a family of pairwise orthogonal subspaces which together span the enveloping space. The axioms of quantum mechanics assert that the measurement reduces the state vector to lie in one of those subspaces with different probabilities. For an exposition of basic quantum theory see [11].

Let $\mathcal{O}$ be an observable defined by the family of pairwise orthogonal subspaces $\{S_i \mid 0 \leqslant i < m\}$. In our notation we write $\mathbf{Fin}[\mathcal{O}](i, \chi)$ for the measurement of $\mathcal{O}$ on a quantum system described by state $\chi{:}q(\mathbb{B}^n)$, where i is a result parameter determining the subspace to which state is reduced and χ is a value-result parameter giving that state.

Finalisation is entirely defined using the probabilistic combinator of pGCL (see [21, 28] for an unabridged treatment); in our notation for procedures we write:

$$\begin{array}{l}\mathbf{proc\ Fin}\,(\mathcal{V},\ \mathbf{result}\ i{:}\{0,\ldots,m-1\},\ \mathbf{value\ result}\ \chi{:}q(\mathbb{B}^n)) \mathrel{\widehat{=}} \\ \qquad\qquad \oplus\,[\ (i, \chi := j, P_{S_j}(\chi))\ @\ \langle \chi, P_{S_j}(\chi)\rangle\ \mid 0 \leqslant j < m\ ]\ .\end{array}$$

where P_{S_j} is the projector onto subspace S_j. That definition of **Fin** remains valid when an observable $\mathcal{O}$ is defined by a self-adjoint operator O. In that case the projector for the j-th eigenspace of O is written $P^j_{\mathcal{O}}$.

We now introduce a simple form of finalisation important enough to deserve its own notation. For $\delta_x{:}q(\mathbb{B}^n)$ we define $\mathbb{C}\delta_x \mathrel{\widehat{=}} \{\alpha\delta_x \bullet \alpha{:}\mathbb{C}\}$ *i.e.* the one-dimensional complex vector space spanned by the basis vector δ_x. Let Δ be the indexed family of subspaces $[\mathbb{C}\,\delta_x \bullet x{:}\mathbb{B}^n]$: then finalisation with respect to Δ is called *diagonal* finalisation and abbreviated $\mathbf{Fin}[\Delta]\,(x, \chi)$. We might not decide to return x since $q(\mathbb{B}^n) \cap \mathbb{C}\,\delta_x$ is a singleton: in that case we would just write $\mathbf{Fin}[\Delta]\,(\chi)$.

2.5 Valid qGCL programs

A qGCL program is a sequence of assignments, **skip**, initialisations, evolutions and finalisations manipulated by the following constructors: sequential composition, conditional selection, repetition, probabilistic choice and procedure declaration and call. Declaration of standard procedure is augmented with the **qproc** procedure defined is Section 2.4.2.

PUT SYNTAX HERE. NFA paper

In a **qproc** we allow a probabilistic assignment to a variable only if that variable is not evaluated in the guard of a loop. Such a restriction looks unreasonable if applied to probabilistic algorithms in general, but in our case is acceptable because of the finitary nature of all quantum algorithms known to date, even the cryptographic algorithms. In particular, we want to avoid the following situation:

$$\begin{array}{l}\textbf{while } \neg b \textbf{ do}\\ \quad (b := 0)\ {}_{\frac{1}{2}}\!\oplus (b := 1)\\ \textbf{od}.\end{array}$$

The loop above is indeed terminating, thus reversible and implementable through a suitable quantum operator. However, such an operator would require an infinite-dimensional Hilbert space, as the number of iterations of the loop is clearly unbounded. For simplicity we decide not to deal with infinite-sized operators and we therefore avoid probabilistically terminating computations.

As qGCL is entirely defined in terms of pGCL's commands and constructs it follows that pGCL semantics (see Appendix B) is an adequate semantic model for qGCL.

3 Simplifying expressions

We now start developing our compiler for qGCL by addressing the first step of the compilation process: the simplification of expressions.

In section 2 of their paper [10], Hoare *et al.* describe a specification space for the Guarded-Command Language. The predicate-transformer semantics for GCL can be embedded in the expectation-transformer semantics for pGCL (see [17, 15] for example) which makes sure the programming laws remain sound. The only difference is that whilst GCL programs are conjunctive, that is:

$$P \mathbin{;} (Q \sqcap R) = (P \mathbin{;} Q) \sqcap (P \mathbin{;} R)$$

for pGCL programs we only have a refinement:

$$P \mathbin{;} (Q \sqcap R) \sqsubseteq (P \mathbin{;} Q) \sqcap (P \mathbin{;} R)$$

and the equality holds, in the expectation-transformer model, if and only if P is not probabilistic (*i.e.* a standard GCL program). Fortunately conjunctivity is not required in Hoare *et al.*'s work, so all their laws are consistent with pGCL.

We now have to give some laws satisfied by the probabilistic choice constructor:

$$P\ {}_1\!\oplus Q = P,$$

$$P\ {}_r\!\oplus Q = Q\ {}_{1-r}\!\oplus P,$$

$$P\ {}_r\!\oplus P = P.$$

Next we have to describe how probabilistic choice interacts with other pGCL constructors and statements:

$$(P\ {}_r\!\oplus Q) \mathbin{;} R = (P \mathbin{;} R)\ {}_r\!\oplus (Q \mathbin{;} R)\,,$$

$$x := e \mathbin{;} (P\ {}_r\!\oplus Q) = (x := e \mathbin{;} P)\ {}_{r[x\backslash e]}\!\oplus (x := e \mathbin{;} Q)\,,$$

$$\mathbf{var}\, x \bullet (P \;{}_r\!\oplus Q) = (\mathbf{var}\, x \bullet P) \;{}_r\!\oplus (\mathbf{var}\, x \bullet Q)\,.$$

All these laws are also listed in Appendix C for quick reference; more laws are given in [12].

Next we deal with the proper simplification of expressions, which mainly involves the introduction of a register variable A of the target hardware. Here we add simplification rules for probabilistic choice, quantum finalisation and procedures; the first rule introduces register variable A into probabilistic choice:

Lemma 3.1. *If variable A does not appear in expression p then:*

$$Q \;{}_p\!\oplus R = \left[\begin{array}{l} \mathbf{var}\, A{:}[0,1] \bullet \\ \quad A := p\,; \\ \quad (Q \;{}_A\!\oplus R) \\ \mathbf{rav} \end{array}\right].$$

Proof. We reason:

$Q \;{}_p\!\oplus R$

$=$ **skip** identity and law D-3

$(\mathbf{var}\, A \bullet \mathbf{rav}\, ; Q) \;{}_p\!\oplus (\mathbf{var}\, A \bullet \mathbf{rav}\, ; R)$

$=$ law D-4

$(\mathbf{var}\, A \bullet A := p\; \mathbf{rav}\, ; Q) \;{}_p\!\oplus (\mathbf{var}\, A \bullet A := p\; \mathbf{rav}\, ; R)$

$=$ laws S-2, D-2

$\mathbf{var}\, A \bullet (A := p\, ; Q) \;{}_p\!\oplus (A := p\, ; R)\; \mathbf{rav}$

$=$ law A-1

$\mathbf{var}\, A \bullet A := p\, ; (Q \;{}_A\!\oplus R)\; \mathbf{rav}$

□

The next rule rewrites a probabilistic choice in more "standard" terms.

Lemma 3.2. *For programs Q, R and variable b not occurring free in both Q and R, we have:*

$$Q \;{}_p\!\oplus R = \left[\begin{array}{l} \mathbf{var}\; b{:}\mathbb{B} \bullet \\ \quad (b := 1) \;{}_p\!\oplus (b := 0)\,; \\ \quad (Q \triangleleft b \triangleright R) \\ \mathbf{rav} \end{array}\right].$$

Proof. We reason:

$Q \;{}_p\!\oplus R$

$=$ **skip** identity and law D-3

$(\mathbf{var}\, b \bullet \mathbf{rav}\, ; Q) \;{}_p\!\oplus (\mathbf{var}\, b \bullet \mathbf{rav}\, ; R)$

$=$ laws D-4, D-5

$$
\begin{array}{ll}
(\mathbf{var}\, b \bullet b := 1 \,;\, Q\ \mathbf{rav}) \;{}_p\!\oplus\; (\mathbf{var}\, b \bullet b := 0 \,;\, R\ \mathbf{rav}) & \\
= & \text{laws S-2, D-2} \\
\mathbf{var}\, b \bullet (b := 1 \,;\, Q) \;{}_p\!\oplus\; (b := 0 \,;\, R)\ \mathbf{rav} & \\
= & \text{programming laws} \\
\mathbf{var}\, b \bullet (b := 1 \,;\, Q \triangleleft b \triangleright R) \;{}_p\!\oplus\; (b := 0 \,;\, Q \triangleleft b \triangleright R)\ \mathbf{rav} & \\
= & \text{law S-2} \\
\mathbf{var}\, b \bullet (b := 1) \;{}_p\!\oplus\; (b := 0) \,;\, (Q \triangleleft b \triangleright R)\ \mathbf{rav} &
\end{array}
$$

□

Therefore we can compile any probabilistic choice into the generation of a random boolean followed by a conditional choice. Another solution to our compilation problem is:

$$
Q \;{}_p\!\oplus\; R = \left[\begin{array}{l} \mathbf{var}\ r{:}[0,1] \bullet \\ \quad r :\in [0,1] \,; \\ \quad (Q \triangleleft (r < p) \triangleright R) \\ \mathbf{rav} \end{array}\right],
$$

but it requires a pGCL semantics extended to continuous distributions. That work has been already carried out by McIver and Morgan [13], but since we do not need to use continuous distributions later, we opt for the former (and simpler) solution and we do not need to augment our semantics.

We may suppose that the generation of the probabilistic boolean is implemented by some special low-level instruction of the target machine, but it turns out that we can implement it by high-level qGCL commands, that is by means of an appropriate quantum computation. Therefore, later in this section we give that quantum implementation.

We now offer a quantum implementation for the probabilistic generation of booleans.

Lemma 3.3. *If variable χ is different from b and A, then:*

$$
(b := 1) \;{}_A\!\oplus\; (b := 0) = \left[\begin{array}{l} \mathbf{var}\ \chi{:}q(\mathbb{B}) \bullet \\ \quad \chi := H_{\arccos(\sqrt{1-A})}(\delta_0) \,; \\ \quad \mathbf{Fin}[\Delta](b,\chi) \\ \mathbf{rav} \end{array}\right]
$$

where H_θ is the (unitary) rotation operator defined:

$$
\begin{array}{l}
H_\theta{:}q(\mathbb{B}) \to q(\mathbb{B}) \\
H_\theta(\chi)(x) \mathrel{\widehat{=}} (1-x)(\chi(0)\cos\theta - \chi(1)\sin\theta) + x(\chi(0)\sin\theta + \chi(1)\cos\theta).
\end{array}
$$

Proof. We reason:

$$
\begin{array}{ll}
\mathbf{var}\ \chi \bullet \chi := H_{\arccos(\sqrt{1-A})}(\delta_0) \,;\, \mathbf{Fin}[\Delta](b,\chi)\ \mathbf{rav} & \\
= & \text{definition of } H_\theta \\
\mathbf{var}\ \chi \bullet \chi := (\delta_0\sqrt{1-A} + \delta_1\sqrt{A}) \,;\, \mathbf{Fin}[\Delta](b,\chi)\ \mathbf{rav} &
\end{array}
$$

$$= \qquad \text{definition of } \mathbf{Fin}$$

$$\mathbf{var}\ \chi \bullet \chi := (\delta_0\sqrt{1-A} + \delta_1\sqrt{A}) \,;\, (b, \chi := 0, \delta_0) \ {}_{\langle \chi, P^0_\Delta(\chi) \rangle}\!\oplus (b, \chi := 1, \delta_1)\ \mathbf{rav}$$

$$= \qquad \text{law A-1}$$

$$\begin{array}{l} \mathbf{var}\ \chi \bullet \\ \quad \begin{pmatrix} \chi := (\delta_0\sqrt{1-A} + \delta_1\sqrt{A}); \\ b, \chi := 0, \delta_0 \end{pmatrix} {}_{\langle \chi, P^0_\Delta(\chi)\rangle[\chi \backslash (\delta_0\sqrt{1-A}+\delta_1\sqrt{A})]}\!\oplus \\ \qquad \begin{pmatrix} \chi := (\delta_0\sqrt{1-A} + \delta_1\sqrt{A}); \\ b, \chi := 1, \delta_1 \end{pmatrix} \\ \mathbf{rav} \end{array}$$

$$= \qquad \text{definition of } P^0_\Delta \text{ and logic}$$

$$\begin{array}{l} \mathbf{var}\ \chi \bullet \\ \begin{pmatrix} \chi := (\delta_0\sqrt{1-A} + \delta_1\sqrt{A}); \\ b, \chi := 0, \delta_0 \end{pmatrix} {}_{1-A}\!\oplus \begin{pmatrix} \chi := (\delta_0\sqrt{1-A} + \delta_1\sqrt{A}); \\ b, \chi := 1, \delta_1 \end{pmatrix} \\ \mathbf{rav} \end{array}$$

$$= \qquad \text{law A-2}$$

$$\mathbf{var}\ \chi \bullet (b, \chi := 0, \delta_0) \ {}_{1-A}\!\oplus (b, \chi := 1, \delta_1)\ \mathbf{rav}$$

$$= \qquad \text{law P-2}$$

$$\mathbf{var}\ \chi \bullet (b, \chi := 1, \delta_1) \ {}_{A}\!\oplus (b, \chi := 0, \delta_0)\ \mathbf{rav}$$

$$= \qquad \text{laws S-2, D-2}$$

$$(\mathbf{var}\ \chi \bullet b, \chi := 1, \delta_1\ \mathbf{rav}) \ {}_{A}\!\oplus (\mathbf{var}\ \chi \bullet b, \chi := 0, \delta_0\ \mathbf{rav})$$

$$= \qquad \text{laws D-5, D-4}$$

$$(b := 1 \,;\, \mathbf{var}\ \chi \bullet \mathbf{rav}) \ {}_{A}\!\oplus (b := 0 \,;\, \mathbf{var}\ \chi \bullet \mathbf{rav})$$

$$= \qquad \text{laws D-3 and } \mathbf{skip} \text{ identity}$$

$$(b := 1) \ {}_{A}\!\oplus (b := 0)$$

□

At this point we are ready to give a quantum implementation for probabilistic choice.

Lemma 3.4. *If variables A and χ do not appear in programs Q and R then:*

$$Q \ {}_{p}\!\oplus R = \left[\begin{array}{l} \mathbf{var}\ A{:}[0,1], \chi{:}q(\mathbb{B}) \bullet \\ \quad A := p; \\ \quad \chi := H_{\arccos(\sqrt{1-A})}(\delta_0); \\ \quad \mathbf{Fin}[\Delta](\chi); \\ \quad (Q \triangleleft \chi = \delta_1 \triangleright R) \\ \mathbf{rav} \end{array} \right]$$

Proof. We start from LHS:

$Q\ {}_p\!\oplus R$

$=$ lemma 3.1

$\mathbf{var}\ A{:}[0,1] \bullet A := p\ ⨾\ (Q\ {}_A\!\oplus R)\ \mathbf{rav}$

$=$ lemma 3.2

$\mathbf{var}\ A{:}[0,1], b{:}\mathbb{B} \bullet A := p\ ⨾\ (b := 1)\ {}_A\!\oplus (b := 0)\ ⨾\ (Q \triangleleft b \triangleright R)\ \mathbf{rav}$

$=$ lemma 3.3

$\mathbf{var}\ A{:}[0,1], b{:}\mathbb{B}, \chi{:}q(\mathbb{B}) \bullet$
$A := p\ ⨾\ \chi := H_{\arccos(\sqrt{1-A})}(\delta_0)\ ⨾\ \mathbf{Fin}[\Delta](b,\chi)\ ⨾\ (Q \triangleleft b \triangleright R)$
$\mathbf{rav}$

$=$ definition of **Fin** and logic

$\mathbf{var}\ A{:}[0,1], b{:}\mathbb{B}, \chi{:}q(\mathbb{B}) \bullet$
$A := p\ ⨾\ \chi := H_{\arccos(\sqrt{1-A})}(\delta_0)\ ⨾\ \mathbf{Fin}[\Delta](b,\chi)\ ⨾\ (Q \triangleleft \chi = \delta_1 \triangleright R)$
$\mathbf{rav}$

$=$ suppress and undeclare variable b in **Fin**

$\mathbf{var}\ A{:}[0,1], \chi{:}q(\mathbb{B}) \bullet$
$A := p\ ⨾\ \chi := H_{\arccos(\sqrt{1-A})}(\delta_0)\ ⨾\ \mathbf{Fin}[\Delta](\chi)\ ⨾\ (Q \triangleleft \chi = \delta_1 \triangleright R)$
$\mathbf{rav}$

□

Next we give two rules for "simplifying" finalisation. The first one allows us to write a general observation in terms of a diagonal (thus simpler) observation (the proof is a replay, in our formalism, of standard results of linear algebra).

Lemma 3.5. *If $\mathcal{O}$ is an observable for qureg $\chi{:}q(\mathbb{B}^n)$ then:*

$$\mathbf{Fin}[\mathcal{O}](r,\chi) = \left[\begin{array}{l} \chi := C_{\mathcal{O}}(\chi) ⨾ \\ \mathbf{Fin}[\Delta](r,\chi) ⨾ \\ r,\chi := a_r, C_{\mathcal{O}}^{-1}(\chi) \end{array}\right]$$

where the a_r's are the eigenvalues of O (the self-adjoint operator corresponding to $\mathcal{O}$) and $C_{\mathcal{O}}$ is the unitary operator which change from O's eigenvector basis to the standard basis.

Proof. By the spectral theorem the eigenvectors γ_i of O form an orthonormal basis for the enveloping space. Furthermore, the action of O on χ can be written:

$$O(\chi) = \sum_{i=0}^{n-1} \gamma_i \cdot a_i \langle \gamma_i, \chi \rangle$$

where the a_i's are the (possibly degenerate) eigenvalues of O. Let U be a unitary operator. From the rules of quantum theory we infer that the quantum system described by state χ and observable O is equivalent to the system described by state $U(\chi)$ and observable

UOU^{-1}, that is they return the same physical predictions (for a more detailed treatment see [11] for example). It is now a trivial matter of algebra to show that:

$$O(\chi) = U^{-1}(UOU^{-1})U(\chi)\,. \tag{2}$$

Now, let $C_{\mathcal{O}}$ be the transformation defined as:

$$C_{\mathcal{O}}(\chi) \mathrel{\widehat{=}} \sum_{i=0}^{n-1} \delta_i \cdot \langle \gamma_i, \chi \rangle\ .$$

It is easy to see that $C_{\mathcal{O}}$ represents the basis change from basis γ_i to standard basis δ_i; moreover we have that $C_{\mathcal{O}}$ is unitary, thus a legitimate quantum computation.

Now we reason:

$$\begin{bmatrix} \chi := C_{\mathcal{O}}(\chi) ⨟ \\ \mathbf{Fin}[\Delta](r,\chi) ⨟ \\ r,\chi := a_r, C_{\mathcal{O}}^{-1}(\chi) \end{bmatrix}$$

$=$ definition of **Fin**

$$\begin{bmatrix} \chi := C_{\mathcal{O}}(\chi) ⨟ \\ \oplus\ [\ (r,\chi := j, P_{\Delta}^{j}(\chi))\ @\ \langle \chi, P_{\Delta}^{j}(\chi)\rangle\ \bullet\ 0 \leqslant j < n\ ] ⨟ \\ r,\chi := a_r, C_{\mathcal{O}}^{-1}(\chi) \end{bmatrix}$$

$=$ law A-1

$$\begin{bmatrix} \oplus \left[\begin{pmatrix} \chi := C_{\mathcal{O}}(\chi) ⨟ \\ r,\chi := j, P_{\Delta}^{j}(\chi) \end{pmatrix} @\ \langle C_{\mathcal{O}}(\chi), P_{\Delta}^{j} C_{\mathcal{O}}(\chi)\rangle\ \bullet\ 0 \leqslant j < n \right] ⨟ \\ r,\chi := a_r, C_{\mathcal{O}}^{-1}(\chi) \end{bmatrix}$$

$=$ laws A-2, S-2

$$\oplus \left[\begin{pmatrix} r,\chi := j, P_{\Delta}^{j} C_{\mathcal{O}}(\chi)) ⨟ \\ r,\chi := a_r, C_{\mathcal{O}}^{-1}(\chi) \end{pmatrix} @\ \langle C_{\mathcal{O}}(\chi), P_{\Delta}^{j} C_{\mathcal{O}}(\chi)\rangle\ \bullet\ 0 \leqslant j < n \right]$$

$=$ law A-2

$$\oplus\ [\ (r,\chi := a_j, C_{\mathcal{O}}^{-1} P_{\Delta}^{j} C_{\mathcal{O}}(\chi))\ @\ \langle C_{\mathcal{O}}(\chi), P_{\Delta}^{j} C_{\mathcal{O}}(\chi)\rangle\ \bullet\ 0 \leqslant j < n\ ]$$

$=$ definition of $C_{\mathcal{O}}$ and $P_{\mathcal{O}}^{j}$ and logic

$$\oplus\ [\ (r,\chi := a_j, P_{\mathcal{O}}^{j}(\chi))\ @\ \langle \chi, P_{\mathcal{O}}^{j}(\chi)\rangle\ \bullet\ 0 \leqslant j < n\ ]$$

$=$ definition of **Fin**

$\mathbf{Fin}[\mathcal{O}](r,\chi)$

□

The second rule introduces register variable A in finalisation.

Lemma 3.6. *If variable A is distinct from r, then:*

$$\mathbf{Fin}[\mathcal{O}](r,\chi) = \begin{bmatrix} \mathbf{var}\ A \bullet \\ \quad \mathbf{Fin}[\mathcal{O}](A,\chi) ⨟ \\ \quad r := A \\ \mathbf{rav} \end{bmatrix}\ .$$

Proof. We reason:

$\mathbf{Fin}[\mathcal{O}](r,\chi)$

$=$ **skip** identity and law D-3

$\mathbf{Fin}[\mathcal{O}](r,\chi) \mathbin{;} \mathbf{var}\ A \bullet \mathbf{rav}$

$=$ law D-5

$\mathbf{var}\ A \bullet \mathbf{Fin}[\mathcal{O}](r,\chi)\ \mathbf{rav}$

$=$ definition of **Fin**

$\mathbf{var}\ A \bullet \oplus[\ (r,\chi := j, P^j_{\mathcal{O}}(\chi))\ @\ \langle \chi, P^j_{\mathcal{O}}(\chi)\rangle \ \mid 0 \leqslant j < m\ ]\ \mathbf{rav}$

$=$ law D-9

$\mathbf{var}\ A \bullet \oplus[\ (A,r,\chi := j,j, P^j_{\mathcal{O}}(\chi))\ @\ \langle \chi, P^j_{\mathcal{O}}(\chi)\rangle \ \mid 0 \leqslant j < m\ ]\ \mathbf{rav}$

$=$ law A-2

$\mathbf{var}\ A \bullet \oplus[\ (A,\chi := j, P^j_{\mathcal{O}}(\chi) \mathbin{;} r := A)\ @\ \langle \chi, P^j_{\mathcal{O}}(\chi)\rangle \ \mid 0 \leqslant j < m\ ]\ \mathbf{rav}$

$=$ law S-2

$\mathbf{var}\ A \bullet \oplus[\ (A,\chi := j, P^j_{\mathcal{O}}(\chi)\ @\ \langle \chi, P^j_{\mathcal{O}}(\chi)\rangle \ \mid 0 \leqslant j < m\ ] \mathbin{;} r := A\ \mathbf{rav}$

$=$ definition of **Fin**

$\mathbf{var}\ A \bullet \mathbf{Fin}[\mathcal{O}](A,\chi) \mathbin{;} r := A\ \mathbf{rav}$

□

For procedures there is little to add, but we must take into account some constraints imposed by quantum theory. Consider the (standard/quantum) procedure Z defined as:

$$\mathbf{proc/qproc}\ Z(\mathbf{value}\, p_1,\ \mathbf{result}\, p_2,\ \mathbf{value\ result}\, p_3) \mathrel{\widehat{=}} Zbody$$

where p_1 is not a qureg. A call $Z(a,b,c)$ is refined by the following program:

$$\begin{array}{l}\mathbf{var}\ p_1,p_2,p_3 \bullet \\ \quad p_1,p_3 := a,c \mathbin{;} \\ \quad Zbody \mathbin{;} \\ \quad b,c := p_2,p_3 \\ \mathbf{rav}\ .\end{array}$$

The proof is straightforward from the semantics for procedure call [16]. However, if Z is a **qproc** and either p_2 or p_3 are quregs, the no-cloning theorem would make the assignments above invalid in qGCL. Fortunately the solution is pretty simple: we use simultaneous assignments, which are valid for quantum registers, too.

Lemma 3.7. *Let Z be the (standard/quantum) procedure defined as:*

$$\mathbf{proc/qproc}\ Z(\mathbf{value}\, p_1,\ \mathbf{result}\, p_2,\ \mathbf{value\ result}\, p_3) \mathrel{\widehat{=}} Zbody$$

where p_1 is not a qureg, then:

$$
\begin{array}{l}
\textbf{var } a, b, c \bullet Z(a, b, c) \textbf{ rav} \\
\sqsubseteq \\
\textbf{var } a, b, c, p_1, p_2, p_3 \bullet \\
\quad p_1 := a ; \\
\quad p_3, c := c, p_3 ; \\
\quad Zbody ; \\
\quad b, p_2 := p_2, b ; \\
\quad c, p_3 := p_3, c \\
\textbf{rav}
\end{array}
$$

where b and c are quregs if Z is **qproc**.

Proof. The refinement is again a straightforward application of algebraic programming laws. □

The lemmata above enable us to expand theorem 5.1 of [10] to include qGCL constructs as well. Intuitively, this theorem states that any program Q is refined by a program R which "looks" closer to a hardware implementation, as high-level constructs are implemented by means of low-level constructs. Standard code in quantum procedures also needs to be made reversible (see section 2.3) by means of the technique exposed in [27].

Theorem 3.8. *If Q is a qGCL program (see section 2.5), then there is a program R such that:*

$$Q \sqsubseteq \textbf{var}\, v, A \bullet R\, \textbf{rav}$$

where:

v is a list of variables,

A is the accumulator variable,

R does not contain any declaration or undeclaration,

all conditions of conditionals and iterations in R are A,

all assignments have one of the "simple" forms

$$A := A \; bop \; t \quad or \quad A := dop \; A \quad or \quad v := A \quad or \quad A := t$$

where t is a variable or constant, bop is a binary operator and dop a unary operator,

and

all probabilistic choices are reduced to boolean generation of probability A, all finalisations are diagonal with return parameter A and code in quantum procedures is reversible.

Proof. For standard code we point the interested reader to Hoare *et al.*'s paper [10]. For procedures the thesis follows by the lemmata shown in this section and structural induction. Standard code in quantum procedures is made reversible by means of the technique presented in [27]. □

4 Normal form reduction

This phase of the compilation process eliminates high-level control structures, reducing the source program to a single flat iteration called *normal form*, which models an arbitrary executing device. A *simple* normal form is a specialisation used to represent our target architecture. We first give Hoare *et al.*'s definition of normal form. It is:

$$\mathbf{var}\ v \bullet v : [a, true] \,⨟\, \mathbf{while}\ b\ \mathbf{do}\ Q\ \mathbf{od} ⨟ : [true, c]\ \mathbf{rav}$$

where:

v is a list of variables,

a is a predicate representing an assumption about the initial state,

c is a predicate representing a coercion about the final state,

Q is a conditional

$$R \lhd b_1 \rhd (\ldots (S \lhd b_n \rhd T))$$

where $R, \ldots, S$ are simple assignments and T is arbitrary,

$b_1, \ldots, b_n, c$ are pairwise disjoint and $b \mathrel{\widehat{=}} (b_1 \vee \ldots \vee b_n)$.

This will be abbreviated by:

$$v : [a, b \longrightarrow Q, c].$$

Hoare *et al.*'s architecture [10] has two classical registers: the already introduced general-purpose register A and a sequential control register P, which contains the memory address of the next instruction to be executed. We now turn to the simple normal form, which describes the behaviour of the hardware architecture.

Definition (Hoare *et al.*). A normal form $(P, v : [a, R, c])$ is *simple* if there exist two integers s and f such that $s \leqslant f$ and

$$\begin{aligned} &a = (P = s), \quad c = (P = f), \\ &R = \square_{s \leqslant k < f}(P = k \longrightarrow R_k), \\ &R_k = (x_k, P := e_k, d_k) \end{aligned}$$

where the x_k's are program variables and the simultaneous assignment R_k can be executed by a single machine instruction, that is expressions e_k, d_k are simple and in particular expressions d_k have the form:

$$P + 1, \quad n, \quad ((P + 1) \lhd A \rhd n)$$

where n is a constant.

The (simple) normal form describing the behaviour of a program stored in a memory m can be thus written as:

$$\begin{array}{l} \textbf{var } P, A \bullet \\ \quad P := s ; \\ \quad \textbf{while } \ (s \leqslant P < f) \textbf{ do } m[P] \textbf{ od} ; \\ \quad : [true, P = f] \\ \textbf{rav} \ . \end{array}$$

4.1 Target quantum architecture

We now describe our target quantum hardware architecture, which is just an augmentation of that described in [10]. In particular, the memory space is unique: it will contain standard variables and quregs in the same address space; we also suppose to have a read-only classical register X which holds finalisation's result. The target code is augmented with two new sets of instructions:

- a set $U \mathrel{\widehat{=}} \{u_i \bullet \ 0 \leqslant i < n\}$ of n primitives for implementing any unitary operator. Each u_i has the form $\chi := u_i(\chi)$ where χ is a qureg and unitary operators are achieved by sequential composition of these assignments;

- a set $O \mathrel{\widehat{=}} \{o_j \bullet \ 0 \leqslant j < m\}$ of m primitives for implementing diagonal observation. Each o_j has the form $\chi := o_j(\chi)$ where χ is a qureg and observation is achieved composing sequentially these assignments. Each observation terminates with the assignment $A := X$, which puts the result in register variable A.

Since the u_i's and o_j's are used as simple assignments, they readily comply with the definition of simple normal form.

We know that due to the continuous nature of unitary operators, it is not possible to implement them exactly using only finite means. Therefore one can only hope to produce a good approximating operator by means of a finite set of "basic" operators. Fortunately this is possible and there are various sets of quantum operators which can approximate any unitary operator: see for example the works of Barenco [1], Deutsch *et al.* [5] and Barenco *et al.* [2]. The drawback is that most unitary operators can only be approximated using an exponential amount of these "basic" operators and therefore those operators correspond to inefficient computations. So far only a few unitary operators have been shown to be efficiently (*i.e.* polynomially) implementable: quantum Fourier transform and Hadamard transform.

We are not particularly interested in any set of universal quantum operators, we just make the assumption that one is available and can approximate with arbitrary accuracy any unitary operator acting on quregs of any size. This assumption is motivated by the *threshold theorem* for quantum computation (see for example [18]). By combining quantum error-correction codes and fault-tolerant quantum gates it makes possible implementing arbitrarily large quantum computations reliably, at the expense of a polylogarithmic increase of the size of the original circuit.

With respect to finalisation we clearly assume that diagonal observation is efficiently implementable by some hardware. These assumptions are equivalent to others found in different models for quantum computation, for example the quantum circuit model [5, 2, 18].

In an alternative approach one might allow the use of special-purpose hardware in which particular (non-universal) unitary gates are used. Such a view is very similar to the hardware compilation approach, in which high-level source programs are compiled into special-purpose hardware chips (see for example [4, 19]). A clear advantage of this approach is the greater speed at which programs are executed, since the supporting hardware is specialised, thus simpler and faster. This simplicity of the hardware might also turn out to be useful for quantum computing, because a special-purpose quantum chip would be easier to build than a general-purpose one. On the other hand, specialised hardware has a narrow scope of utilisation, but due to the limited number of quantum algorithms developed so far this might not be an issue.

Since in this work we do not focus on any specific set U and set O of quantum primitives, we do not have to specify how unitary operators and diagonal finalisation are compiled in terms of those primitives. Nevertheless, the assumptions made enable us to carry on with our reasoning and deduce the correctness of our compiler. Therefore, one might think the part of the compiler which deals with unitary operators and finalisations as a separate module which can be later improved or changed, without affecting the overall compiler's correctness (of course the module must comply with the requirements above).

Before turning to the main theorem of this section we have to give compilation rules for quantum procedures.

4.2 Compiling quantum procedures

In this section we deal with the problem of simulating classical code on a quantum processor. We recall that such a possibility is allowed by the definition of suitable **qproc**'s.

Procedures defined as **qproc**'s differ from standard procedures in the fact that standard code is executed on quantum hardware rather than on standard hardware. pGCL code can be made reversible by the technique set out in [27] and here we give an implementation of that technique in terms of unitary operations. We make use of the so-called *controlled-U* operations, which are just a simplification of the standard conditional.

Definition. Let *cond* be a predicate and U a reversible statement:

$$C(cond, U) \mathrel{\widehat{=}} (U \triangleleft cond \triangleright \mathbf{skip}).$$

We demand U be reversible because we want to execute controlled operations on the quantum hardware. Our choice is motivated by the existence of efficient quantum implementations of the controlled operation, as described in [2, 3, 18]. We note that Feynman's CNOT [9], the earliest example of a controlled operation, clearly belongs to this class of transformations.

Our aim is thus to rewrite the standard conditional and loop constructor by means of these controlled operations, in order to have code executable on the quantum hardware. Since by theorem 3.8 we deal with simple assignments involving only binary and unary mathematical-logic operators, assignments are thus better treated using the quantum version of those operators, which are readily available from their classical counterparts (see

[18] for example). We stress once again that here we do not specify how controlled operations are actually implemented by means of quantum low-level instructions u_i's: we just assume that an efficient quantum implementation exists.

In the following lemma we provide an implementation for the reversible conditional by means of controlled operations (the reversible conditional is explained in [27], stacks and related procedures are also defined in Appendix C for convenience). We recall that control structures are first simplified by means of theorem 3.8.

Lemma 4.1. *Let P and Q be reversible statements. If variable A appears in neither P nor Q then:*

$$(P \fatsemi \mathbf{push}\ T) \triangleleft cond \triangleright (Q \fatsemi \mathbf{push}\ F) = \left[\begin{array}{l} \mathbf{var}\ A{:}\mathbb{B} \bullet \\ \quad \mathbf{push}\ F \fatsemi \\ \quad C(cond,\ (\mathbf{pop}\ A \fatsemi P \fatsemi \mathbf{push}\ T)) \fatsemi \\ \quad \mathbf{top}\ A \fatsemi \\ \quad C(\neg A,\ (\mathbf{pop}\ A \fatsemi Q \fatsemi \mathbf{push}\ F)) \\ \mathbf{rav} \end{array}\right].$$

Proof. We reason from the RHS:

$$\begin{array}{ll}
RHS & \\
= & \text{introduce } K \\
\mathbf{var}\ A \bullet \mathbf{push}\ F \fatsemi C(cond,\ (\mathbf{pop}\ A \fatsemi P \fatsemi \mathbf{push}\ T)) \fatsemi K\ \mathbf{rav} & \\
= & \text{controlled operation and law S-2} \\
\mathbf{var}\ A \bullet \mathbf{push}\ F \fatsemi (\mathbf{pop}\ A \fatsemi P \fatsemi \mathbf{push}\ T \fatsemi K) \triangleleft cond \triangleright K\ \mathbf{rav} & \\
= & \text{semantics of } \mathbf{push} \text{ and law A-1} \\
\mathbf{var}\ A \bullet (\mathbf{push}\ F \fatsemi \mathbf{pop}\ A \fatsemi P \fatsemi \mathbf{push}\ T \fatsemi K) \triangleleft cond \triangleright (\mathbf{push}\ F \fatsemi K)\ \mathbf{rav} &
\end{array}$$

We now reason on the left-hand branch of the previous conditional:

$$\begin{array}{ll}
\mathbf{push}\ F \fatsemi \mathbf{pop}\ A \fatsemi P \fatsemi \mathbf{push}\ T \fatsemi K & \\
= & \text{definition of } K \text{ and programming laws} \\
A := F \fatsemi P \fatsemi \mathbf{push}\ T \fatsemi \mathbf{top}\ A \fatsemi C(\neg A,\ (\mathbf{pop}\ A \fatsemi Q \fatsemi \mathbf{push}\ F)) & \\
= & \text{definition of controlled-U and } \mathbf{top} \\
A := F \fatsemi P \fatsemi \mathbf{push}\ T \fatsemi \mathbf{pop}\ A \fatsemi \mathbf{push}\ A \fatsemi ((\mathbf{pop}\ A \fatsemi Q \fatsemi \mathbf{push}\ F) \triangleleft \neg A \triangleright \mathbf{skip})) & \\
= & \text{law ST-1} \\
A := F \fatsemi P \fatsemi A := T \fatsemi \mathbf{push}\ A \fatsemi ((\mathbf{pop}\ A \fatsemi Q \fatsemi \mathbf{push}\ F) \triangleleft \neg A \triangleright \mathbf{skip})) & \\
= & A \text{ does not appear in } P \\
P \fatsemi A := T \fatsemi \mathbf{push}\ A \fatsemi ((\mathbf{pop}\ A \fatsemi Q \fatsemi \mathbf{push}\ F) \triangleleft \neg A \triangleright \mathbf{skip})) & \\
= & \text{programming laws} \\
P \fatsemi \mathbf{push}\ T \fatsemi A := T \fatsemi ((\mathbf{pop}\ A \fatsemi Q \fatsemi \mathbf{push}\ F) \triangleleft \neg A \triangleright \mathbf{skip})) &
\end{array}$$

$=$ law A-1 and **skip** identity

$P \,;\, \mathbf{push}\ T \,;\, A := T$

Similar reasoning shows that the right-hand branch of the conditional reduces to:

$Q \,;\, \mathbf{push}\ F \,;\, A := F$

Therefore we can rewrite the RHS as:

$\mathbf{var}\ A \bullet (P \,;\, \mathbf{push}\ T \,;\, A := T) \triangleleft cond \triangleright (Q \,;\, \mathbf{push}\ F \,;\, A := F)\ \mathbf{rav}$

$=$ laws D-2 and D-5

$(P \,;\, \mathbf{push}\ T \,;\, \mathbf{var}\ A \bullet A := T\ \mathbf{rav}) \triangleleft cond \triangleright (Q \,;\, \mathbf{push}\ F \,;\, \mathbf{var}\ A \bullet A := F\ \mathbf{rav})$

$=$ laws D-4, D-3 and **skip** identity

$(P \,;\, \mathbf{push}\ T) \triangleleft cond \triangleright (Q \,;\, \mathbf{push}\ F)$

We note that *cond* should not involve stack variables. This is insured by the fact that the stack is "invisible" to the programmer, as it is a structure generated during the compilation process. □

We now deal with loops: we have to transform the standard loop constructor into a quantum, but equivalent, operator.

4.2.1 Quantum loops

In this section we shall define, by means of recursion, a possible quantum loop constructor and show that such a constructor refines the standard loop.

We start by defining the "standard" way to evaluate boolean functions via quantum operators.

Definition. Let $b{:}\mathbb{B}^n \to \mathbb{B}$ be a boolean function. We define:

$$\forall i{:}\mathbb{B}^n, j{:}\mathbb{B} \bullet \gamma_b(\delta_i, \delta_j) \mathrel{\widehat{=}} \delta_{(j \odot b(i))}$$

where $\odot$ denotes standard exclusive-or.

Function γ_b links standard predicates with their quantum implementation, in the sense precised by the next theorem:

Theorem 4.2. *Let $b{:}\mathbb{B}^n \to \mathbb{B}$ be a boolean function. Then:*

$$\forall i{:}\mathbb{B}^n \bullet\ b(i) = (\gamma_b(\delta_i, \delta_0) = \delta_1)$$

Proof. We start reasoning from the RHS:

$(\gamma_b(\delta_i, \delta_0) = \delta_1)$

$=$ definition of γ_b

$(\delta_{(0 \odot b(i))} = \delta_1)$

$=$ XOR property

$(\delta_{b(i)} = \delta_1)$

$=$ logic

$b(i)$

□

Next we define the quantum operator for evaluating boolean functions.

Definition. Let $b{:}\mathbb{B}^n \to \mathbb{B}$ be a boolean function. We define:

$$\Gamma_b \left(\sum_{i:\mathbb{B}^n} \delta_i \otimes \sum_{j:\mathbb{B}} \delta_j \right) \mathrel{\widehat{=}} \sum_{i:\mathbb{B}^n} \sum_{j:\mathbb{B}} (\delta_i \otimes \gamma_b(\delta_i, \delta_j)) \ .$$

The definition of Γ_b is readily shown to be linear and unitary: it is thus a feasible quantum operator. We are mainly interested in the particular case of $\delta_0{:}q(\mathbb{B})$ because:

$$\Gamma_b \left(\sum_{i:\mathbb{B}^n} \delta_i \otimes \delta_0 \right) = \sum_{i:\mathbb{B}^n} \delta_i \otimes \delta_{b(i)}$$

that is, we can evaluate a condition on a superposition of input values.

We now define the quantum version of the controlled-U operator defined in the previous section.

Definition. Let Q be a quantum operator over $q(\mathbb{B}^n)$. We define:

$$C_u(Q) \left(\sum_{i:\mathbb{B}^n} \delta_i \otimes \sum_{j:\mathbb{B}} \delta_j \right) \mathrel{\widehat{=}} \sum_{i:\mathbb{B}^n, j:\mathbb{B}} \left(Q^j(\delta_i) \otimes \delta_j \right) ,$$

where $Q^1 = Q$ and Q^0 is defined to be the identity operator.

Again, that definition of C_u satisfies linearity and unitarity. We note that when the right-hand qureg is a standard basis state, then C_u reduces to the standard controlled-U operator. This argument is formalised in the next theorem.

Theorem 4.3. *Let $\chi{:}q(\mathbb{B}^n)$ be a qureg and Q a quantum operator over $q(\mathbb{B}^n)$. Then:*

$$\left[\begin{array}{l} \textbf{var } \psi{:}\delta(\mathbb{B}) \bullet \\ \quad \chi, \psi := C_u(Q)(\chi \otimes \psi) \\ \textbf{rav} \end{array} \right] = \left[\begin{array}{l} \textbf{var } \psi{:}\delta(\mathbb{B}) \bullet \\ \quad C(\psi = \delta_1, \chi := Q(\chi)) \\ \textbf{rav} \end{array} \right] .$$

Proof. We start reasoning from the LHS:

LHS

$=$ laws D-8, S-2

$$\begin{array}{l}\textbf{var } \psi{:}\delta(\mathbb{B})\bullet \\ \quad (\psi := \delta_0 \,\text{;}\, \chi, \psi := C_u(Q)(\chi \otimes \psi)) \sqcap (\psi := \delta_1 \,\text{;}\, \chi, \psi := C_u(Q)(\chi \otimes \psi)) \\ \textbf{rav}\end{array}$$

$=$ combine assignment (law A-2)

$$\begin{array}{l}\textbf{var } \psi{:}\delta(\mathbb{B})\bullet \\ \quad (\chi, \psi := C_u(Q)(\chi \otimes \delta_0)) \sqcap (\chi, \psi := C_u(Q)(\chi \otimes \delta_1)) \\ \textbf{rav}\end{array}$$

$=$ definition of C_u

$$\begin{array}{l}\textbf{var } \psi{:}\delta(\mathbb{B})\bullet \\ \quad (\chi, \psi := \chi \otimes \delta_0) \sqcap (\chi, \psi := Q(\chi) \otimes \delta_1) \\ \textbf{rav}\end{array}$$

$=$ decompose tensor product (χ, ψ not entangled)

$$\begin{array}{l}\textbf{var } \psi{:}\delta(\mathbb{B})\bullet \\ \quad (\psi := \delta_0 \,\text{;}\, \chi := \chi) \sqcap (\psi := \delta_1 \,\text{;}\, \chi := Q(\chi)) \\ \textbf{rav}\end{array}$$

$=$ remove vacuous assignment

$$\begin{array}{l}\textbf{var } \psi{:}\delta(\mathbb{B})\bullet \\ \quad (\psi := \delta_0) \sqcap (\psi := \delta_1 \,\text{;}\, \chi := Q(\chi)) \\ \textbf{rav}\end{array}$$

$=$ definition of conditional and logic

$$\begin{array}{l}\textbf{var } \psi{:}\delta(\mathbb{B})\bullet \\ \quad (\psi := \delta_0 \,\text{;}\, (\chi := Q(\chi) \triangleleft \psi = \delta_1 \triangleright \textbf{skip})) \sqcap (\psi := \delta_1 \,\text{;}\, (\chi := Q(\chi) \triangleleft \psi = \delta_1 \triangleright \textbf{skip})) \\ \textbf{rav}\end{array}$$

$=$ definition of C

$$\begin{array}{l}\textbf{var } \psi{:}\delta(\mathbb{B})\bullet \\ \quad (\psi := \delta_0 \,\text{;}\, C(\psi = \delta_1, \chi := Q(\chi)) \sqcap (\psi := \delta_1 \,\text{;}\, C(\psi = \delta_1, \chi := Q(\chi)) \\ \textbf{rav}\end{array}$$

$=$ laws S-2, D-8

$$\begin{array}{l}\textbf{var } \psi{:}\delta(\mathbb{B})\bullet \\ \quad C(\psi = \delta_1, \chi := Q(\chi)) \\ \textbf{rav}\end{array}$$

$=$

RHS

$\square$

Finally, we present our quantum loop constructor.

Definition. Let $b{:}\mathbb{B}^n \to \mathbb{B}$ be a boolean function, $\chi{:}q(\mathbb{B}^n)$ a qureg and Q a quantum operator over $q(\mathbb{B}^n)$. We define:

$$qloop(b,\chi,Q) \mathrel{\widehat{=}} \mu X. \left[\begin{array}{l} \textbf{var}\ \psi{:}q(\mathbb{B}) \bullet \\ \quad \psi := \delta_0 ; \\ \quad \chi,\psi := \Gamma_b(\chi \otimes \psi) ; \\ \quad \chi,\psi := C_u(Q ; X)(\chi \otimes \psi) \\ \textbf{rav} \end{array}\right]$$

where μ denotes the recursion operator.

Unitarity of *qloop* follows from the unitarity of Γ_b and C_u. The assignment $\psi := \delta_0$ does not pose any problem, since it might be thought of as a variable initialisation not performed by the quantum hardware.

The following theorem states that *qloop* is, with respect to basis states, a valid refinement for loops with bodies made of quantum operators. This kind of "semi-classical" loop arises in the "quantumisation" of classical loops.

The program state is now represented by a qureg. In fact, since we are transforming classical code into quantum code, a qureg over $\delta(\mathbb{B}^n)$ suffices.

The guard of the loop has to be changed accordingly: it is implemented through the γ_b operator defined before. If $b{:}\mathbb{B}^n \to \mathbb{B}$ is the guard of the loop, then via δ standard state $i{:}\mathbb{B}^n$ is mapped to $\delta_i{:}\delta(\mathbb{B}^n)$ and, by means of theorem 4.2, condition $b(i)$ becomes $\gamma_b(\delta_i, \delta_0) = \delta_1$.

We stress again the requirement that the body of the "semi-classical" loop must be the "quantumisation" of its classical counterpart. This is formalised by requiring that the corresponding quantum operator is closed over the set of standard basis states $\delta(\mathbb{B}^n)$.

Theorem 4.4. *Let $b{:}\mathbb{B}^n \to \mathbb{B}$ be a boolean function, $\chi{:}\delta(\mathbb{B}^n)$ a qureg and Q a quantum operator over $\delta(\mathbb{B}^n)$. Then:*

$$\left[\begin{array}{l} \textbf{while}\ \ \gamma_b(\chi,\delta_0) = \delta_1\ \textbf{do} \\ \quad \chi := Q(\chi) \\ \textbf{od} \end{array}\right] \sqsubseteq qloop(b,\chi,Q)\,.$$

Proof. We reason from the RHS (we drop arguments for simplicity):

$qloop$

$=$ fixed-point theorem

$$\begin{array}{l} \textbf{var}\ \psi{:}q(\mathbb{B}) \bullet \\ \quad \psi := \delta_0 ; \\ \quad \chi,\psi := \Gamma_b(\chi \otimes \psi) ; \\ \quad \chi,\psi := C_u(Q ; qloop)(\chi \otimes \psi) \\ \textbf{rav} \end{array}$$

$=$ combine assignments (law A-2)

$$\begin{array}{l} \textbf{var}\ \psi{:}q(\mathbb{B}) \bullet \\ \quad \chi,\psi := \Gamma_b(\chi \otimes \delta_0) ; \\ \quad \chi,\psi := C_u(Q ; qloop)(\chi \otimes \psi) \\ \textbf{rav} \end{array}$$

$=$ definition of Γ_b

$$\begin{array}{l}\mathbf{var}\ \psi{:}q(\mathbb{B})\bullet\\ \quad \chi,\psi := \chi\otimes\gamma_b(\chi,\delta_0)\,;\\ \quad \chi,\psi := C_u(Q\,;qloop)(\chi\otimes\psi)\\ \mathbf{rav}\end{array}$$

$=$ remove vacuous assignment

$$\begin{array}{l}\mathbf{var}\ \psi{:}q(\mathbb{B})\bullet\\ \quad \psi := \gamma_b(\chi,\delta_0)\,;\\ \quad \chi,\psi := C_u(Q\,;qloop)(\chi\otimes\psi)\\ \mathbf{rav}\end{array}$$

$=$ theorem 4.3

$$\begin{array}{l}\mathbf{var}\ \psi{:}q(\mathbb{B})\bullet\\ \quad \psi := \gamma_b(\chi,\delta_0)\,;\\ \quad C(\psi=\delta_1,(Q\,;qloop))\\ \mathbf{rav}\end{array}$$

$=$ definition of C

$$\begin{array}{l}\mathbf{var}\ \psi{:}q(\mathbb{B})\bullet\\ \quad \psi := \gamma_b(\chi,\delta_0)\,;\\ \quad (\chi := (Q\,;qloop)(\chi))\lhd\psi=\delta_1\rhd\mathbf{skip}\\ \mathbf{rav}\end{array}$$

$=$ law A-1 and **skip** identity

$$\begin{array}{l}\mathbf{var}\ \psi{:}q(\mathbb{B})\bullet\\ \quad (\psi := \gamma_b(\chi,\delta_0)\,;\chi := (Q\,;qloop)(\chi))\lhd\gamma_b(\chi,\delta_0)=\delta_1\rhd(\psi := \gamma_b(\chi,\delta_0))\\ \mathbf{rav}\end{array}$$

$\sqsupseteq$ remove initialisation (laws D-2,D-9)

$$\begin{array}{l}\mathbf{var}\ \psi{:}q(\mathbb{B})\bullet\\ \quad (\chi := (Q\,;qloop)(\chi))\lhd\gamma_b(\chi,\delta_0)=\delta_1\rhd\mathbf{skip}\\ \mathbf{rav}\end{array}$$

$=$ reduce scope (law D-5)

$$\begin{array}{l}\mathbf{var}\ \psi{:}q(\mathbb{B})\bullet\mathbf{rav}\,;\\ (\chi := (Q\,;qloop)(\chi))\lhd\gamma_b(\chi,\delta_0)=\delta_1\rhd\mathbf{skip}\end{array}$$

$=$ law D-3 and **skip** identity

$$(\chi := (Q\,;qloop)(\chi))\lhd\gamma_b(\chi,\delta_0)=\delta_1\rhd\mathbf{skip}$$

$=$ sequential composition

$$(\chi := Q(\chi)\,;qloop)\lhd\gamma_b(\chi,\delta_0)=\delta_1\rhd\mathbf{skip}$$

$=$ fixed-point theorem

$$
\begin{array}{l}
\quad \textbf{while } \gamma_b(\chi, \delta_0) = \delta_1 \textbf{ do} \\
\quad\quad \chi := Q(\chi) \\
\quad \textbf{od} \\
= \\
LHS
\end{array}
$$

□

We note that we could have derived the same result using the reversible rules set out in [27], as we did for the conditional. However, in this case that route is longer: from a classical loop one starts by developing the reversible loop; then it is quite simple to prove that:

$$
\left[\begin{array}{l} \textbf{while } b \textbf{ do} \\ \quad S_r \,⨾\, \textbf{push } T \\ \textbf{od} \end{array} \right] \sqsubseteq \left[\begin{array}{l} \textbf{var } x{:}\mathbb{B} \bullet \\ \quad \mu X.(x := b \,⨾\, C((S_r \,⨾\, \textbf{push } T \,⨾\, X), x)) \\ \textbf{rav} \end{array} \right].
$$

The last step is to replace the assignment $x := b$ with the γ_b operator and the controlled operation with the C_u operator. For simplicity we decided to give directly the quantum implementation. One could of course give that implementation for the conditional, as well.

4.3 Normal form for qGCL programs

We are thus in a position to give the theorem which relates a source program with a simple normal form. qGCL programs are transformed according to theorem 3.8 and we can directly use Hoare *et al.*'s theory without modifications because:

- probabilistic choice is split into a quantum computation followed by a conditional, already treated by Hoare *et al.*;
- code in quantum procedures is compiled in terms of unitary evolution (section 4.2);
- unitary evolution and finalisation are compiled into sequences of simple assignments (primitives u_i and o_j) directly executable by the quantum hardware.

Theorem 4.5. *If Q is a program in the form of R in theorem 3.8, then:*

$$\textbf{var } v, A \bullet Q \textbf{ rav} \sqsubseteq P, A, v : [P = s, R, P = f]$$

where $R \mathrel{\widehat=} (\Box_{s \leqslant k < f} P = k \to R_k)$ and R_k is an assignment directly implementable as a single machine instruction.

Proof. See Hoare *et al.*'s paper [10] for the proof and the compilation rules which define assignments R_k. □

5 Introducing machine state

The simple normal form introduced by theorem 4.5 does not represent compiled code yet, as it still features symbolic identifiers, *i.e.* variables. The purpose of this section is to provide a correct method for replacing variables by their numeric addresses in the machine memory M. We use Hoare *et al.*'s work without any modification, since we have shown how to reduce a qGCL source program into a simple normal form. In the following we give a brief explanation of their arguments.

A symbol table can be thought as a (total, injective) function Ψ which maps each variable name of the program into the address of the corresponding location in M, so that $M[\Psi(x)]$ is the memory location holding the value for variable x. Hoare *et al.* [10] show the correctness of the syntactic substitution of program variable x with $M[\Psi(x)]$.

Next they show how to combine all the preceding theorems to fulfil the compiling task. Consider a source program Q with variables $x_1, \ldots, x_n$. By theorem 3.8 we refine Q by a program Q' which contains only simple expressions. Then, theorem 4.5 shows how to write a simple normal form for Q':

$$\mathbf{var}\ v, A \bullet Q'\ \mathbf{rav} \sqsubseteq P, A, v : [P = s, R, P = f]$$

where $R \mathrel{\widehat{=}} (\Box_{s \leqslant k < f} P = k \to R_k)$ and R_k is an assignment directly implementable as a single machine instruction.

By means of the symbol table Ψ we perform the necessary link between program variables and addresses in memory M. For $s \leqslant k < f$ we define:

$$m[k] \mathrel{\widehat{=}} R_k[M[\Psi(x_1)]\backslash x_1, \ldots, M[\Psi(x_n)]\backslash x_n]\ .$$

We are now ready to write the simple normal form which describes the low-level behaviour of program Q':

$$\begin{array}{l} \mathbf{var}\ P, A \bullet \\ \quad P := s ; \\ \quad \mathbf{while}\ \ (s \leqslant P < f)\ \mathbf{do}\ m[P]\ \mathbf{od} ; \\ \quad : [true, P = f] \\ \mathbf{rav}\ . \end{array}$$

6 Example

In this section we exemplify the use of some compilation rules by applying them to a simple quantum program.

Consider the following standard procedure for natural exponentiation:

$$\begin{array}{l} \mathbf{proc}\ exp(\mathbf{value}\ x, y{:}\mathbb{B}^k ; \mathbf{result}\ \chi{:}\mathbb{B}^n) \mathrel{\widehat{=}} \\ \mathbf{var}\ \bullet \\ \quad \chi := 1 ; \\ \quad \mathbf{while}\ \ y > 0\ \mathbf{do} \\ \qquad \chi, y := \chi \cdot x, y - 1 \\ \quad \mathbf{od} \\ \mathbf{rav}. \end{array}$$

Procedure *exp* computes x^y and returns the result in variable χ. Since in this paper we do not deal with compilation of data structures, we directly code naturals with bit registers. We also assume that n is big enough to contain the outcome of the computation (given k, then $n \geq k \cdot (2^k - 1)$).

We want to use procedure *exp* as a quantum procedure, so that it is possible to compute exponentiation on quregs in a superposition of standard states. For this purpose, we have first to map classical states (via Dirac δ) to their quantum analogue and then to compile the code into a quantum implementation, by means of the techniques set out. We simply replace **proc** by **qproc** and let the compiler doing all the necessary type changes, as explained in section 2.5. Procedure *exp* now becomes *qexp*:

$$\textbf{qproc}\ qexp(\textbf{value result}\ x, y{:}\delta(\mathbb{B}^k)\ ⨟\ \textbf{value result}\ \chi{:}\delta(\mathbb{B}^n))\ .$$

As explained in section 4.2, the simple assignments constituting the body of *exp* are readily compiled by means of the quantum primitives for mathematical-logic operations. In our case we need a multiplication operator $Mul(\chi, \psi)$, a decrement operator $Dec(\chi)$ and a swap operator $Swap(\chi, \psi)$. Initialisation $\chi := 1$ is readily modelled via the Dirac δ map (this may be viewed as a trivial example of quantum data structures compilation). Quantum procedure *qexp* is:

$$\begin{array}{l}
\textbf{qproc}\ qexp(\textbf{value result}\ x, y{:}\delta(\mathbb{B}^k)\ ⨟\ \textbf{value result}\ \chi{:}\delta(\mathbb{B}^n)) \mathrel{\widehat{=}} \\
\textbf{var}\ \bullet \\
\quad \chi := \delta_1 ⨟ \\
\quad \textbf{while}\ \ (y > 0)\ \textbf{do} \\
\qquad \chi, x, y := Mul(\chi, x), Dec(y) \\
\quad \textbf{od} \\
\textbf{rav}\ .
\end{array}$$

We note that condition $(y > 0)$ should be more properly written as $(\delta^{-1}(y) > 0)$, since now y is a qureg of type $\delta(\mathbb{B}^k)$. We prefer the former notation for clarity reasons.

We now write a simple quantum program which uses *qexp*:

$$E \mathrel{\widehat{=}} \left[\begin{array}{l}
\textbf{var}\ r{:}\delta(\mathbb{B}^n)\ ⨟\ a, b{:}\delta(\mathbb{B}^k)\ \bullet \\
\quad qexp(a, b, r) \\
\textbf{rav}
\end{array}\right]$$

We now apply the techniques developed to get a version of program E which looks closer to an hypothetical quantum hardware implementation. For simplicity we do not deal with simplification of standard code (introduction of register variable A, *etc.*).

E

$=$ lemma 3.7 (procedure call)

var $r{:}\delta(\mathbb{B}^n)\,;\,a,b{:}\delta(\mathbb{B}^k)\ \bullet$
    var $\chi{:}\delta(\mathbb{B}^n)\,;\,x,y{:}\delta(\mathbb{B}^k)\ \bullet$
        $x, a := a, x;$
        $y, b := b, y;$
        $\chi, r := r, \chi;$
        $\chi := \delta_1;$
        while $(y > 0)$ **do**
            $\chi, x, y := Mul(\chi, x), Dec(y)$
        od;
        $a, x := x, a;$
        $b, y := y, b;$
        $r, \chi, := \chi, r$
    rav
rav

$=$ enlarge scope

var $r,\chi{:}\delta(\mathbb{B}^n)\,;\,a,b,x,y{:}\delta(\mathbb{B}^k)\ \bullet$
    $x, a := a, x;$
    $y, b := b, y;$
    $\chi, r := r, \chi;$
    $\chi := \delta_1;$
    while $(y > 0)$ **do**
        $\chi, x, y := Mul(\chi, x), Dec(y)$
    od;
    $a, x := x, a;$
    $b, y := y, b;$
    $r, \chi, := \chi, r$
rav

$=$ compose assignments and introduce *Swap*

var $r,\chi{:}\delta(\mathbb{B}^n)\,;\,a,b,x,y{:}\delta(\mathbb{B}^k)\ \bullet$
    $x, a, y, b, \chi, r := Swap(x, a), Swap(y, b), Swap(\chi, r);$
    while $(y > 0)$ **do**
        $\chi, x, y := Mul(\chi, x), Dec(y)$
    od;
    $x, a, y, b, \chi, r := Swap(x, a), Swap(y, b), Swap(\chi, r)$
rav

$=$ theorem 4.2 (substitute boolean condition)

$$
\begin{array}{l}
\textbf{var } r,\chi{:}\delta(\mathbb{B}^n)\, ⨾\, a,b,x,y{:}\delta(\mathbb{B}^k) \bullet \\
\quad x,a,y,b,\chi,r := Swap(x,a), Swap(y,b), Swap(\chi,r) ⨾ \\
\quad \textbf{while } (\gamma_{(y>0)}(y,\delta_0) = \delta_1) \textbf{ do} \\
\quad\quad \chi,x,y := Mul(\chi,x), Dec(y) \\
\quad \textbf{od} ⨾ \\
\quad x,a,y,b,\chi,r := Swap(x,a), Swap(y,b), Swap(\chi,r) \\
\textbf{rav}
\end{array}
$$

$\sqsubseteq$ theorem 4.4 (introduce *qloop*)

$$
\begin{array}{l}
\textbf{var } r,\chi{:}\delta(\mathbb{B}^n)\, ⨾\, a,b,x,y{:}\delta(\mathbb{B}^k) \bullet \\
\quad x,a,y,b,\chi,r := Swap(x,a), Swap(y,b), Swap(\chi,r) ⨾ \\
\quad qloop\,((y>0), \chi \otimes x \otimes y, Mul(\chi,x) \otimes Dec(y)) ⨾ \\
\quad x,a,y,b,\chi,r := Swap(x,a), Swap(y,b), Swap(\chi,r) \\
\textbf{rav}
\end{array}
$$

We compiled program E into a program containing quantum transformations and quantum data structures only: that program is thus executable by quantum hardware. We note that our refinement techniques prove that quantum code refines standard code with respect to quantum basis states. It would be of course not possible to prove refinement for general quregs, as classical computation does not allow superposition of standard states.

7 Conclusions

The normal-form approach to compilation is divided into three steps: simplification of expressions, reduction to normal form and introduction of machine state.

In the first step we showed how to simplify non-standard qGCL constructs such as finalisation and probabilistic choice. To this end we extended Hoare *et al.*'s work introducing some algebraic laws satisfied by the probabilistic choice constructor and we gave simplification rules for finalisation, probabilistic choice and procedures.

In the second step we introduced a very simple quantum architecture which fits into Hoare *et al.*'s computing model. That architecture seems to pose minimum requirements for universal quantum computing but it is currently unclear on which architecture quantum computers will be based upon. Also, it might be the case that quantum computers will be built using special-purpose quantum hardware, a view enforced by the low number of quantum algorithms developed so far. However, the normal-form approach exposed remains valid independent of the quantum architecture chosen.

The third and last step involves the substitution of symbolic identifiers (*i.e.* variables) by numeric addresses of memory locations. Here we apply directly Hoare *et al.*'s theory, because qGCL programs are reduced to normal form by means of the second step of compilation.

Future work on this approach includes the study of a more efficient quantum implementation of the reversible loop. Furthermore, we did not discuss any compilation problem relative to quantum data structures or any efficiency issue.

8 Acknowledgements

The author would like to thank Jeff Sanders for suggesting the use of the normal-form approach for compiler design and for commenting a draft of this paper. The author would also like to thank the anonymous referees for their useful suggestions.

This work has been financially supported by the Engineering and Physical Sciences Research Council (UK) and Consiglio Nazionale delle Ricerche (Italy).

References

[1] Adriano Barenco. A universal two-bit gate for quantum computation. *Proceedings of the Royal Society of London* A, 449:679–683, 1995.

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

[3] D. Beckman, A.N. Chari, S. Devabhaktuni, and J. Preskill. Efficient networks for quantum factoring. *Physical Review A*, 54(2):1034, 1996.

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

[5] D. Deutsch, A. Barenco, and A. Ekert. Universality in quantum computation. *Proceedings of the Royal Society of London* A, 449:669–677, 1995.

[6] David Deutsch. Quantum computational networks. *Proceedings of the Royal Society of London*, A425:73–90, 1989.

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

[8] Richard P. Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21(6/7):467–488, 1982.

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

[10] C.A.R. Hoare, He Jifeng, and A. Sampaio. Normal form approach to compiler design. *Acta Informatica*, 30:701–739, 1993.

[11] Chris J. Isham. *Lectures on quantum theory*. Imperial College Press, 1997.

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

[13] Annabelle McIver and Carroll Morgan. Partial correctness for probabilistic demonic programs. Technical report, Oxford University Computing Laboratory, 2000. Available at http://www.comlab.ox.ac.uk.

[14] Graeme Mitchison and Richard Jozsa. Counterfactual computation. *Proceedings of the Royal Society of London* A, 457:1175–1193, 2001.

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

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

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

[18] Micheal A. Nielsen and Isaac L. Chuang. *Quantum computation and quantum information.* Cambridge University Press, 2000.

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

[20] M. Reed and B. Simon. *Methods of Mathematical Physics. I:Functional Analysis.* Acamedic Press, 1972.

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

[22] Benjamin Schumacher. Quantum coding. *Physical Review* A, 51(4):2738–2747, 1995.

[23] John von Neumann. *Mathematical Foundations of Quantum Mechanics.* Princeton University Press, 1955.

[24] W. K. Wootters and W. H. Zurek. A single quantum cannot be cloned. *Nature*, 299(5886):802–803, 1982.

[25] Takeo Yokonuma. *Tensor spaces and exterior algebra.* American Mathematical Society, 1992.

[26] Paolo Zuliani. Formal reasoning for quantum mechanical nonlocality. Technical Report RR-01-05, Oxford University Computing Laboratory, 2001. Submitted for publication. Available at http://www.comlab.ox.ac.uk.

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

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

A Basic quantum mechanics

To support our formalism we rely on von Neumann's approach to quantum mechanics [23], that is the theory of linear operators over Hilbert spaces. It is a well founded theory, since in nearly seventy years there has not been any discrepancy between theoretical predictions and experimental results.

Among its advantages is the uniform treatment given to observation and state evolution, which are treated as subclasses of linear operators. Furthermore, it is able to express in a formal way any quantum system, thus helping to carry out correct reasoning.

A good self-contained exposition of this theory can be found in [11] (it does not cover infinite quantum systems, but they require many mathematical subtleties and here we need finite systems only).

A.1 Hilbert spaces

A Hilbert space $\mathcal{H}$ is a vector space equipped with a scalar product making it a complete inner product space. Here we consider only complex vector spaces $\mathbb{C}^n$, for $n{:}\mathbb{N}$. The scalar product is therefore the application $\langle\cdot,\cdot\rangle{:}\mathbb{C}^n \times \mathbb{C}^n \to \mathbb{C}$ defined by:

$$\langle\psi,\phi\rangle \mathrel{\widehat{=}} \sum_{0\leqslant i<n} \psi_i^* \phi_i$$

where ψ_i is the i-th component of $\psi{:}\mathbb{C}^n$, and z^* is the complex conjugate of $z{:}\mathbb{C}$.

The *length* of a vector ψ is defined $\|\psi\| \mathrel{\widehat{=}} \langle\psi,\psi\rangle^{\frac{1}{2}}$; ψ is *normalised* if $\|\psi\|^2 = 1$. The n-dimensional unit sphere is the set $\mathbb{C}_1^n \mathrel{\widehat{=}} \{\psi{:}\mathbb{C}^n \bullet \|\psi\|^2 = 1\}$. If $f{:}A \to \mathbb{C}$ for some set A, the *square norm* of f is $\|f\|^2 \mathrel{\widehat{=}} \sum_{a:A} |f(a)|^2$.

Two vectors ψ,ϕ are said to be *orthogonal* if $\langle\psi,\phi\rangle = 0$, written $\psi \perp \phi$. Two linear subspaces E,F of a Hilbert space $\mathcal{H}$ are said to be *orthogonal* if $\forall\psi{:}E,\phi{:}F \bullet \psi \perp \phi$; we shall write $E \perp F$.

The *orthogonal complement* of a linear subspace E of $\mathcal{H}$ is the set of all vectors perpendicular to it:

$$E^\perp \mathrel{\widehat{=}} \{\psi{:}\mathcal{H} \bullet \forall\phi{:}E, \phi \perp \psi\}.$$

It is also a linear subspace of $\mathcal{H}$.

A function $A{:}\mathcal{H} \to \mathcal{H}$ is also called an *operator*. The *adjoint* (or *hermitian conjugate*) of a linear operator A is the operator $A^\dagger$ defined by:

$$\forall\psi,\phi{:}\mathcal{H} \bullet \langle\psi, A^\dagger\phi\rangle = \langle A\psi,\phi\rangle.$$

A linear operator A is *self-adjoint* (or *hermitian*) if $A = A^\dagger$. In the case of infinite-dimensional Hilbert spaces there is a difference between self-adjointness and hermitianity, but since here we deal only with finite spaces we consider them equivalent. Also, we should check that $A^\dagger$ defined above is actually an operator (which it is, as a matter of fact). All these mathematical details can be found in Reed and Simon's book [20] for example.

In von Neumann's approach to quantum mechanics the state of a physical system is modelled by a vector of some n-dimensional complex Hilbert space and state evolution is modelled by linear operators. As a consequence any quantum operator on $\mathcal{H}$ can always be written as a $n \times n$ complex matrix.

Let A be a $n \times n$ matrix representing a quantum operator $\mathcal{A}$. Then, with respect to an orthonormal basis, the elements of the matrix representing $\mathcal{A}^\dagger$ satisfy:

$$\forall i,j{:}\{1,\ldots,n\} \bullet (A^\dagger)_{ij} = A_{ji}^*.$$

We note that if $\mathcal{A}$ is self-adjoint then $A_{ij} = A_{ji}^*$.

Quantum transformations satisfy also another property: they are *unitary*. Such an operator guarantees the existence of the inverse operator and preserves scalar products, that is for an operator U unitary we have:

$$\forall \psi, \phi : \mathcal{H} \bullet \langle U\psi, U\phi \rangle = \langle \psi, \phi \rangle$$

In terms of matrices it means that the matrix U modelling the evolution of the system must satisfy:

$$U \cdot U^{\dagger} = U^{\dagger} \cdot U = \mathbb{1}$$

where $\mathbb{1}$ is the identity matrix of appropriate size. The set of complex unitary matrices forms a group with the usual matrix multiplication.

A (non-zero) vector $\psi : \mathcal{H}$ is an *eigenvector* of an operator A with *eigenvalue* $a : \mathbb{C}$ if:

$$A\psi = a\psi.$$

In quantum mechanics an *observable* is represented by a self-adjoint operator and the measurable values are exactly the eigenvalues of that operator. In physics there are three famous matrices, the *Pauli spin* matrices S_x, S_y and S_z:

$$S_z \mathrel{\widehat{=}} \frac{1}{2}\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}, \quad S_x \mathrel{\widehat{=}} \frac{1}{2}\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad S_y \mathrel{\widehat{=}} \frac{1}{2}\begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix},$$

which represent the spin observables in the three Euclidean spatial directions for a spin-$\frac{1}{2}$ particle; their eigenvalues are $\pm\frac{1}{2}$, hence their name.

The fundamental *spectral theorem* [11] for finite-dimensional Hilbert spaces states that the set of all eigenvectors of a self-adjoint operator is an orthonormal basis set for $\mathcal{H}$.

For $\psi : \mathcal{H}$ we write P_ψ for the projector onto the one-dimensional subspace spanned by vector ψ:

$$\forall \phi : \mathcal{H} \bullet P_\psi(\phi) \mathrel{\widehat{=}} \frac{\psi}{\|\psi\|^2} \cdot \langle \psi, \phi \rangle.$$

For an observable $\mathcal{O}$ (with discrete eigenvalue spectrum) and eigenvalue λ we denote by $E_{\mathcal{O},\lambda}$ the set of eigenvectors associated to eigenvalue λ; we write $P_{\mathcal{O}}^{\lambda}$ for the projector associated to eigenvalue λ:

$$P_{\mathcal{O}}^{\lambda} \mathrel{\widehat{=}} P_{(\sum_{v:E_{\mathcal{O},\lambda}} v)}\ .$$

The rules of quantum theory state that if the state of a system is described by some normalised vector $\psi : \mathcal{H}$ then, if a measurement of observable $\mathcal{O}$ is made, the probability that the result will be the particular eigenvalue λ is:

$$\mathrm{Prob}(\mathcal{O} = \lambda \mid \psi) = \langle \psi, P_{\mathcal{O}}^{\lambda} \psi \rangle.$$

By the spectral theorem we deduce that the family of eigenspaces of an observable $\mathcal{O}$ is a partition for $\mathcal{H}$ and we have seen that to each eigenspace there is an associated projector $P_{\mathcal{O}}^{\lambda}$. A projector is a self-adjoint operator with just two eigenvalues, 0 and 1; therefore a measurement of an observable $\mathcal{O}$ tells us in which subspace the state vector ψ has "fallen", as a consequence of the measurement process.

The probability rule written above is a special case of the following rule, which holds even for observables with continuous eigenvalue spectrum. For observable $\mathcal{O}$ and normalised state $\psi:\mathcal{H}$, the *expected result* $\langle\mathcal{O}\rangle_\psi$ of measuring $\mathcal{O}$ is:

$$\langle\mathcal{O}\rangle_\psi \mathrel{\widehat{=}} \langle\psi, \mathcal{O}\psi\rangle \ .$$

Alternatively, one can define an observable from a family of pairwise orthogonal subspaces which together span the whole space $\mathcal{H}$ and then consider the projector of each subspace.

A.2 Tensor products

The state of a composite quantum system is described by the tensor product of Hilbert spaces. Consider two complex Hilbert spaces $\mathcal{H}_1, \mathcal{H}_2$ of dimensions n_1, n_2 respectively. For any pair of vectors $\psi:\mathcal{H}_1, \phi:\mathcal{H}_2$, the *tensor vector* $\psi\otimes\phi$ is given by the map $(\cdot\otimes\cdot):\mathcal{H}_1\times\mathcal{H}_2 \to \mathbb{C}^{n_1\cdot n_2}$:

$$(\psi\otimes\phi)_i \mathrel{\widehat{=}} \psi_{i \operatorname{div} n_2}\phi_{i \bmod n_2} \qquad 0 \leqslant i < n_1{\cdot}n_2 \ .$$

Tensor products are linear in each argument, that is:

$$\forall\alpha,\beta:\mathbb{C},\ \psi,\phi:\mathcal{H}_1,\ \chi:\mathcal{H}_2 \bullet (\alpha\psi+\beta\phi)\otimes\chi = (\alpha\psi)\otimes\chi + (\beta\phi)\otimes\chi,$$
$$\forall\alpha,\beta:\mathbb{C},\ \psi,\phi:\mathcal{H}_1,\ \chi:\mathcal{H}_2 \bullet \psi\otimes(\alpha\phi+\beta\chi) = \psi\otimes(\alpha\phi) + \psi\otimes(\beta\chi).$$

Multiplication by a complex number distributes across the tensor product:

$$\forall\alpha:\mathbb{C},\ \psi:\mathcal{H}_1,\ \phi:\mathcal{H}_2 \bullet \alpha(\psi\otimes\phi) = (\alpha\psi)\otimes\phi = \psi\otimes(\alpha\phi).$$

Consider now the vector space $\mathbb{C}^{n_1\cdot n_2}$: defining the scalar product $\langle\psi_1\otimes\psi_2, \phi_1\otimes\phi_2\rangle \mathrel{\widehat{=}} \langle\psi_1,\phi_1\rangle_{\mathcal{H}_1}\langle\psi_2,\phi_2\rangle_{\mathcal{H}_2}$ enable us to define a new Hilbert space $\mathcal{H}_1\otimes\mathcal{H}_2$, called the *tensor product* of $\mathcal{H}_1$ and $\mathcal{H}_2$. Vectors in $\mathcal{H}_1\otimes\mathcal{H}_2$ which cannot be written as a single product $\psi\otimes\phi$ for any $\psi:\mathcal{H}_1$ or $\phi:\mathcal{H}_2$, are called *entangled*. However, every vector in $\mathcal{H}_1\otimes\mathcal{H}_2$ can be written as a *sum* of such product vectors.

The tensor product can be extended to linear operators over Hilbert spaces. Let $A_1:\mathcal{H}_1 \to \mathcal{H}_1$ and $A_2:\mathcal{H}_2 \to \mathcal{H}_2$ be two linear operators over Hilbert spaces $\mathcal{H}_1$ and $\mathcal{H}_2$ respectively. The operator $A_1\otimes A_2:\mathcal{H}_1\otimes\mathcal{H}_2 \to \mathcal{H}_1\otimes\mathcal{H}_2$ is defined as:

$$(A_1\otimes A_2)\psi_1\otimes\psi_2 \mathrel{\widehat{=}} (A_1\psi_1)\otimes(A_2\psi_2)$$

where $\psi_1:\mathcal{H}_1$ and $\psi_2:\mathcal{H}_2$. By linearity it is extended to any vector in $\mathcal{H}_1\otimes\mathcal{H}_2$.

Since linear operators can be represented by matrices, the tensor product is readily available for them, too. Let $A = (a_{i,j})$ and B be two matrices of dimensions $m\times n$ and $p\times q$ respectively: the tensor product $A\otimes B$ is the $mp\times nq$ matrix:

$$\begin{pmatrix} a_{0,0}B & a_{0,1}B & \cdots & a_{0,n-1}B \\ a_{1,0}B & & & \\ \vdots & & & \vdots \\ a_{m-1,0}B & & \cdots & a_{m-1,n-1}B \end{pmatrix}$$

The tensor product of matrices preserves unitarity and distributes over standard matrix multiplication, that is for operators M, N, L, P we have:

$$(M \cdot N) \otimes (L \cdot P) = (M \otimes L) \cdot (N \otimes P).$$

A more rigorous treatment of tensor products for general vector spaces can be found in [25].

B Semantics for pGCL

Semantics for pGCL can be given either relationally [12] or in terms of expectation transformers [15]. We briefly present the main definitions and concepts of the transformer model.

Definition B.1. The ***state*** x of a program P is the array of global variables used during the computation. That is

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

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

The only problem that might arise is when input and output have different types: this is 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.

Before describing the semantic space we introduce a powerful tool: Morgan's specification statement [16]. It is:

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

It describes a computation which changes the (possibly empty) list of variables 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. If there is no value for x satisfying *post*, the computation will terminate achieving *post* miraculously. We denote an initial state by putting a dash $'$ after the variable name.

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:

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

Expectations can be ordered using the standard pointwise functional ordering for which we shall use the symbol $\Rrightarrow$. Standard predicates are easily embedded in $\mathcal{Q}$ by identifying *true* with expectation $\mathbf{1}$ and *false* with $\mathbf{0}$. For standard predicate p we shall write $[p]$ for its embedding.

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 $j \Rrightarrow i$ (or $i \Lleftarrow j$).

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

$$\mathcal{J} \mathrel{\widehat{=}} \mathcal{Q} \to \mathcal{Q}.$$

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 transformer corresponds to a computation: only the *sublinear* ones do [15, 17]. 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

$$x \ominus y \mathrel{\widehat{=}} (x - y) \;\sqcup\; \mathbf{0}.$$

where $\sqcup$ denotes the least upper bound.

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

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

$$
\begin{array}{rcl}
wp.\mathbf{abort}.q & \mathrel{\widehat{=}} & \mathbf{0} \\
wp.\mathbf{skip}.q & \mathrel{\widehat{=}} & q \\
wp.(x := E).q & \mathrel{\widehat{=}} & q[x \backslash E] \\
wp.(R \mathbin{;} S).q & \mathrel{\widehat{=}} & wp.R.(wp.S.q) \\
wp.(R \lhd cond \rhd S).q & \mathrel{\widehat{=}} & [cond] * (wp.R.q) + [\neg cond] * (wp.S.q) \\
wp.(R \sqcap S).q & \mathrel{\widehat{=}} & (wp.R.q) \;\sqcap\; (wp.S.q) \\
wp.(R\; {}_p\!\oplus S).q & \mathrel{\widehat{=}} & p * (wp.R.q) + (1-p) * (wp.S.q) \\
wp.(\mu F) & \mathrel{\widehat{=}} & \text{least fixed point of } F{:}\mathcal{Q} \to Q \\
wp.(z : [pre, post]).q & \mathrel{\widehat{=}} & [pre] * ([\forall z \bullet [post] \Rrightarrow q])[x' \backslash x]
\end{array}
$$

where $q{:}\mathcal{Q}$, $x{:}X$, $p{:}[0,1]$ 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 by expression 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. In the specification statement expectation q must not contain any variable in x'. Recursion is treated in general using the existence of fixed points in $\mathcal{J}$.

Note that binary conditional $R \triangleleft cond \triangleright S$ is a special case of probabilistic choice: it is just $R\ {}_{[cond]}\!\oplus S$.

For procedures we have to distinguish three cases, depending on the kind of parameter. Consider a procedure P defined by:

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

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

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

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

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

pGCL enjoys a refinement calculus, which derives from the semantics above; when we say that program Q refines program P, written $P \sqsubseteq Q$, we mean:

$$P \sqsubseteq Q \mathrel{\widehat{=}} \forall q{:}\mathcal{Q} \bullet wp.P.q \Rrightarrow wp.Q.q\ .$$

Intuitively, $P \sqsubseteq Q$ means that Q is at least at deterministic as P. There is a family of sound laws [12, 17], including those for data refinement, so that the language pGCL is embedded in a refinement calculus (some laws are reported in Appendix C).

The converse of refinement is *abstraction* and it is denoted by $\sqsupseteq$:

$$P \sqsupseteq Q \mathrel{\widehat{=}} \forall q{:}\mathcal{Q} \bullet wp.P.q \Lleftarrow wp.Q.q\ .$$

When $P \sqsupseteq Q$ and $P \sqsubseteq Q$ then P and Q are equal programs and we write $P = Q$.

In pGCL (demonic) nondeterminism is expressed semantically as the combination of all possible probabilistic resolutions:

$$wp.(P \sqcap Q) = \sqcap\{wp.(P\ {}_r\!\oplus Q) \bullet r{:}[0,1]\}.$$

Thus a (demonic) nondeterministic choice between two programs is refined by any probabilistic choice between them:

$$\forall r{:}[0,1] \bullet P \sqcap Q \sqsubseteq P\ {}_r\!\oplus Q$$

originating thereby the law *introduce probabilism.*

Probabilism does not itself yield nondeterminism: if P and Q are deterministic (maximal with respect to the refinement order) then so is $P\ {}_r\!\oplus Q$, but for most authors in the area of quantum computation *nondeterminism* means probabilism.

C Algebraic programming laws

We list a few algebraic laws which hold for pGCL programs. The proofs can be found in [12, 15]; for a complete exposition of an algebra of programming for the Guarded-Command Language see [10]. In the following laws we use the term e to indicate an expression whose type is determined by the context.

Law (Id "skip identity"). $(P \,;\, \mathbf{skip}) = (\mathbf{skip} \,;\, P) = P$

Law (P-1). $P \;{}_1\!\oplus\; Q = P$

Law (P-2). $P \;{}_r\!\oplus\; Q = Q \;{}_{1-r}\!\oplus\; P$

Law (P-3). $P \;{}_r\!\oplus\; P = P$

Law (S-2). $(P \;{}_r\!\oplus\; Q) \,;\, R = (P \,;\, R) \;{}_r\!\oplus\; (Q \,;\, R)$

Law S-2 holds for nondeterministic choice, too.

Law (A-1). $(x := e) \,;\, (P \;{}_r\!\oplus\; Q) = (x := e \,;\, P) \;{}_{r[x\backslash e]}\!\oplus\; (x := e \,;\, Q)$

Law (A-2). $(x := e \,;\, x := f) = (x := f[e\backslash x])$

Law (D-2). *If variable x does not appear in expression p, then:*

$$\mathbf{var}\, x \bullet (P \;{}_p\!\oplus\; Q) = (\mathbf{var}\, x \bullet P) \;{}_p\!\oplus\; (\mathbf{var}\, x \bullet Q)$$
$$(P \;{}_p\!\oplus\; Q)\; \mathbf{rav}\, x = (P\; \mathbf{rav}\, x) \;{}_p\!\oplus\; (Q\; \mathbf{rav}\, x)$$

Since standard conditional is a particular case of probabilistic choice, laws S-2, A-1 and D-2 hold for that, too.

Law (D-3). $\mathbf{var}\, x \bullet\; \mathbf{rav}\, x = \mathbf{skip}$

Law (D-4). $\mathbf{rav}\, x = (x := e \,;\, \mathbf{rav}\, x)$

Law (D-5). *If x is not free in Q,then:*

$$\mathbf{var}\, x \bullet Q = Q \,;\, \mathbf{var}\, x$$
$$\mathbf{rav}\, x \bullet Q = Q \,;\, \mathbf{rav}\, x$$

Declaring a variable is equivalent to a nondeterministic assignment over the variable's data type.

Law (D-8). *Let D be a finite data type. Then:*

$$\mathbf{var}\, x{:}D = \mathbf{var}\, x{:}D \bullet\; \Box[x := d \bullet d \in D]$$

The following refinement is a direct consequence of law D-8:

Law (D-9). *Let e be an expression of the type of variable x, then:*

$$\mathbf{var}\, x \sqsubseteq \mathbf{var}\, x \bullet x := e$$

Law D-9 means that initialising a variable will make a program more deterministic.

We now 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):

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

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; **top** copies in f the last inserted value in the stack. 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** and **top** is that x_0 must not be empty.

For stacks we find useful the following law, whose proof can be found in [27]:

Law (ST-1). *For variable $v{:}D$ and expression $e{:}D$ we have*

$$(\textbf{push } e \mathbin{;} \textbf{pop } v) = (v := e)$$

D Quantum cloning

In this section we explain the cloning problem, that is the production of a perfect copy of some object, and derive the no-cloning theorem for quantum mechanics.

Classically, to send an unknown datum to another party, one just makes a copy of the original and then transmits it over some medium. There is no need for the sender to know the datum since the copying is always possible. It turns out that in quantum mechanics it is not possible to create a perfect copy of an unknown arbitrary quantum state, *i.e.* "cloning" the state, and therefore a quantum cloning operator cannot exist. This feature of quantum theory was firstly discovered by Wootters and Zurek [24] in 1982 and it is due to the linearity of quantum mechanical operators together with the existence of entanglement and hence the need to model combination of state spaces using tensor product.

Before formalising the proof we distinguish two definitions of cloning: cloning element by element (*weak cloning*) and uniform cloning (*strong cloning*). Let A be a set and $C{:}A{\times}A \to A{\times}A$ the putative cloning operator.

Definition. C clones **weakly** over A if:

$$\forall x{:}A\ \exists y{:}A,\ (y \neq x) \bullet C(x, y) = (x, x)\ .$$

Definition. C clones **strongly** over A if:

$$\exists y{:}A\ \forall x{:}A \bullet C(x,y) = (x,x)\ .$$

In quantum mechanics, weak cloning is readily ruled out since it is not reversible. Therefore, it does not need any further investigation.

For strong cloning we first note that since in quantum mechanics composite systems are described through tensor products instead of Cartesian products, that definition of cloning must be modified accordingly. Let $\mathcal{H}$ be a finite Hilbert space and C an operator over $\mathcal{H} \otimes \mathcal{H}$, then we have:

Definition. C clones **strongly** over $\mathcal{H}$ if:

$$\exists y{:}\mathcal{H}\ \forall x{:}\mathcal{H} \bullet C(x \otimes y) = (x \otimes x)\ .$$

Next we have the following theorem, also known as the *no-cloning* theorem.

Theorem D.1 (Wootters and Zurek). *Strong cloning is not possible in quantum mechanics.*

Proof. We shall assume the existence of quantum mechanical strong cloning operator C and we shall prove that it leads to a contradiction. We recall that any valid quantum operator must be linear.

We start from the quantum mechanical definition of strong cloning, omitting the Hilbert space $\mathcal{H}$ for brevity. We reason:

$\exists y\ \forall x \bullet C(x \otimes y) = (x \otimes x)$

$\Rightarrow$ $\qquad$ $\mathcal{H}$ vector space

$\exists y\ \forall x, a \bullet C((x+a) \otimes y) = ((x+a) \otimes (x+a))$

$=$ $\qquad$ tensor product is linear

$\exists y\ \forall x, a \bullet C((x \otimes y) + (a \otimes y)) = (x+a) \otimes (x+a)$

$=$ $\qquad$ C must be linear

$\exists y\ \forall x, a \bullet (C(x \otimes y) + C(a \otimes y)) = (x+a) \otimes (x+a)$

$\Rightarrow$ $\qquad$ cloning property

$\forall x, a \bullet (x \otimes x) + (a \otimes a) = (x+a) \otimes (x+a)$

$=$ $\qquad$ tensor product is linear

$\forall x, a \bullet (x \otimes x) + (a \otimes a) = (x \otimes x) + (a \otimes x) + (x \otimes a) + (a \otimes a)$

$=$ $\qquad$ logic

$\forall x, a \bullet 0 = (a \otimes x) + (x \otimes a)$

$=$ $\qquad$ take for example $a = x \neq 0$

$false$

□

However the theorem does not forbid that for some restricted subset of $\mathcal{H}$ it is possible to have a strong cloning operator. In fact Feynman's CNOT operator [9, 8] clones standard states. The definition of CNOT is:

$$\forall x, c{:}\mathbb{B} \bullet CNOT(x, c) \mathrel{\widehat{=}} (\neg xc + \neg cx, c) \ ,$$

fixing $x = 0$ we get that:

$$\forall c{:}\mathbb{B} \bullet CNOT(0, c) = (c, c) \ ,$$

which satisfies the definition of strong cloning. Embedding standard bits via the Dirac δ function (see section 2.1) enables us to have a strong cloning operator for quantum basis states:

$$\forall c{:}\mathbb{B} \bullet CNOT(\delta_0 \otimes \delta_c) = (\delta_c \otimes \delta_c) \ .$$

However CNOT fails to clone arbitrary states. Take the state $\delta_0 + \delta_1$ for example:

$$
\begin{array}{ll}
CNOT(\delta_0 \otimes (\delta_0 + \delta_1)) & \\
= & \text{logic} \\
CNOT(\delta_0 \otimes \delta_0 + \delta_0 \otimes \delta_1) & \\
= & \text{CNOT linear} \\
CNOT(\delta_0 \otimes \delta_0) + CNOT(\delta_0 \otimes \delta_1) & \\
= & \text{definition of C} \\
\delta_0 \otimes \delta_0 + \delta_1 \otimes \delta_1 &
\end{array}
$$

while:

$$(\delta_0 + \delta_1) \otimes (\delta_0 + \delta_1) = \delta_0 \otimes \delta_0 + \delta_1 \otimes \delta_1 + \delta_0 \otimes \delta_1 + \delta_1 \otimes \delta_0 \, .$$

We note that a basis change does not either diminish or augment the cloning capability of CNOT.

Extension to basis states of n-dimensional Hilbert spaces is readily available using the n-bit version of CNOT:

$$\forall x, c{:}\mathbb{B}^n \bullet CNOT(x, c) \mathrel{\widehat{=}} (\neg xc + \neg cx, c) \ .$$