Quantum Programming

J. W. Sanders and P. Zuliani

Programming Research Group, OUCL, Oxford

Abstract

In this paper a programming language is presented for the expression of quantum algorithms. It contains the features required to program a 'universal' quantum computer (including initialisation and observation), has a formal semantics and body of laws, and provides a refinement calculus supporting the verification and derivation of programs against their specifications. A representative selection of quantum algorithms are expressed in the language and one of them is derived from its specification.

1 Introduction

The purpose of this paper is to present a programming language for quantum computation.

So far quantum algorithms have been described by pseudo code or quantum network [6, 0]. To a computer scientist the former has little attraction. The latter provides a data-flow view of computation and so is useful when considering implementation in terms of gates (a view which is perhaps slightly premature since we have little idea what might constitute the primitive gates). Whilst it expresses modularisation well, it fails to express nondeterminism or probability (both essential ingredients of quantum computation) and does not support data representation or contain high-level control structures.

There is one other model of quantum computation: the quantum Turing machine [5]. That model is as unrealistic for the description of quantum algorithms as Turing machines are for the description of standard algorithms — benefitting from neither data representation nor high-level control structures (including procedure call). It was proposed by physicists presumably because quantum algorithms are justified by their efficiency, and the (standard) Turing machine was perceived as the model for analysing computational complexity.

All three of those notations — pseudo code, quantum networks and quantum Turing machines — suffer by the standards of modern computer science by being too narrow to support specification and rigorous program derivation. Yet quantum algorithms, being intricate and exploiting non-standard intuition, provide exactly the kinds of program for which refinement calculi were developed. A successful programming language for quantum computation should therefore support program refinement.

We introduce an extension of the guarded-command language to express quantum algorithms. It contains both nondeterminism and probability; the former arises in specifying some quantum algorithms and the latter is required in order to 'observe' a quantum system. It has a rigorous semantics and body of laws as a result of other work (see for example [21]) and enjoys the usual features of a refinement calculus. Moreover it abstracts implementation concerns.

Only experience will show whether this language is pitched at the right level of abstraction. However to support that view we here express in it a representative selection of quantum algorithms and perform an exemplary derivation.

After the invention of various efficient quantum algorithms there seems to have been a period of consolidation in which frameworks have been sought to relate those algorithms. The 'hidden subgroup problem' [23] has been seen has a conceptually unifying principle whilst 'multi-particle interference' [4] has been proposed as a unifying principle closer to implementation. Our proposal complements such views by providing a single programming context equally appropriate for the various algorithms at any level of abstraction or generality. It is hoped that quantum algorithms can henceforth be treated with the same precision as standard algorithms.

2 Quantum types

In this section we define, for use in quantum computation, a transformation q that converts a classical type to its quantum analogue. With but one exception, in section 6.5, we restrict consideration of q to registers.

Let $\mathbb{B}$ denote the type of booleans $\{0, 1\}$. For natural number n let $0\,..\,n$ denote the interval of natural numbers at least 0 but less than n

$$0\,..\,n \;\widehat{=}\; \{i \mid 0 \le i < n\}.$$

A (classical) register of size n is a vector of n booleans. The type of all registers of size n is thus defined to be the set of boolean-valued functions on $0\,..\,n$

$$\mathbb{B}^n \;\widehat{=}\; 0\,..\,n \longrightarrow \mathbb{B}.$$

Naturally we are interested in n at least 1 and identify $\mathbb{B}^1$ with $\mathbb{B}$.

The state of a classical system can be expressed using registers. From quantum mechanics we learn — for example from Young's double-slit experiment — that the state of a quantum system is modelled using 'phase' information associated with each standard state [11, 24], consisting of a uni-modular complex number. The probability of observing a state is the modulus squared of its phase; and all probabilities sum to 1. That leads to the following definition.

The quantum analogue of $\mathbb{B}^n$ is

$$q(\mathbb{B}^n) \;\widehat{=}\; \{\chi : \mathbb{B}^n \longrightarrow \mathbb{C} \mid \sum_{x:\mathbb{B}^n} |\chi(x)|^2 = 1\}. \tag{1}$$

An element of $q(\mathbb{B})$ is called a *qubit* [25] and that of $q(\mathbb{B}^n)$ a *qureg*.

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

$$\delta : \mathbb{B}^n \longrightarrow q(\mathbb{B}^n)$$
$$\delta_x(y) \;=\; (y = x).$$

The range of δ, $\{\delta_x \mid x \in \mathbb{B}^n\}$, forms a *basis* for quantum states in this sense:

> any qureg $\chi : q(\mathbb{B}^n)$ is a square-convex complex superposition of standard states
>
> $$\chi \;=\; \textstyle\sum_{x:\mathbb{B}^n} \chi(x)\delta_x, \qquad\qquad \textstyle\sum_{x:\mathbb{B}^n} |\chi(x)|^2 = 1.$$

(In physics δ_x is denoted by the ket $|\,x\rangle$.)

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 $q(\mathbb{B}^n)$ because it is the Hilbert space of lowest dimension containing $q(\mathbb{B}^n)$ as unit sphere. We shall see that, because the elements of the range of δ are pairwise orthogonal in the enveloping space, they are observably distinct with probability 1.

3 Tensor products

In a standard programming language the state of a program having independent component program variables can be expressed as a single variable equal to the Cartesian product of the components. The quantum analogue is that quantum state is the tensor product of its independent state components (equation (2)). The programmer thus has a choice between using individual variables, combining them when required (for example by finalisation) using tensor product; and using one large state but subjecting it to transformation by the tensor product of a particular function on a particular component with the identity function on the remaining components. Thus we require the tensor product both of registers and of functions.

The tensor product of (standard) registers is defined

$$\begin{aligned} &\otimes : \mathbb{B}^m \times \mathbb{B}^n \longrightarrow \mathbb{B}^{mn} \\ &(x \otimes y)(i) \;\;\widehat{=}\;\; x(i \operatorname{div} n) \times y(i \bmod n) \end{aligned}$$

and readily shown to be surjective. That definition lifts, via δ and linearity, to quantum registers

$$\otimes : q(\mathbb{B}^m) \times q(\mathbb{B}^n) \longrightarrow q(\mathbb{B}^{mn}).$$

Well definedness (i.e. square-summability to 1) is immediate. The property of q alluded to above is

$$q(\mathbb{B}^m \times \mathbb{B}^n) = q(\mathbb{B}^m) \otimes q(\mathbb{B}^n). \tag{2}$$

Next tensor product of functions on registers is defined

$$\otimes : (\mathbb{B}^m \longrightarrow \mathbb{B}^m) \times (\mathbb{B}^n \longrightarrow \mathbb{B}^n) \longrightarrow (\mathbb{B}^{mn} \longrightarrow \mathbb{B}^{mn})$$
$$(A \otimes B)(x \otimes y) \ \hat{=} \ A(x) \otimes B(y).$$

Finally $\otimes$ is extended by linearity to functions on quantum registers, for which we follow tradition and use the same symbol yet again

$$\otimes : q(\mathbb{B}^m \longrightarrow \mathbb{B}^m) \times q(\mathbb{B}^n \longrightarrow \mathbb{B}^n) \longrightarrow q(\mathbb{B}^{mn} \longrightarrow \mathbb{B}^{mn}).$$

4 pGCL

In the next section we introduce an imperative quantum-programming language. But first, in this section, we recall Dijsktra's guarded-command language [9] extended to include probabilism [19, 21].

Syntax for the guarded-command language consists of all but the last of these constructs

var	variable declaration
skip	no op
abort	abortion
$x := e$	assignment
$P \fatsemi Q$	sequencing
if $[] \ b_i \longrightarrow S_i$ **fi**	conditional
do $[] \ b_i \longrightarrow S_i$ **od**	iteration
$P \sqcap Q$	nondeterminism
$P \ {}_r\!\oplus \ Q$	probabilism.

Semantics can be given either in terms of predicate transformers [9] in which case each program is thought of as transforming a post-condition to the weakest precondition from which termination, in a state satisfying that postcondition, is guaranteed; or relationally [15] in which case each program is thought of as transforming state before to state after, with a virtual state encoding non-termination.

We require the language to be extended, as usual, to embrace procedure invocation; see for example [18].

pGCL denotes the guarded-command language extended to contain also probabilism. Program $P \ {}_r\!\oplus \ Q$ equals P with probability r and Q with probability $1-r$. Its semantics can be given either by extending pre- and post-conditions to pre- and post-expectations: real-valued random variables

[20]; or relationally [14] by a state-before being related to a set of distributions after. In either case refinement $P \sqsubseteq Q$ means that Q is at least as deterministic as P. The two models are related by a Galois connection embedding the relational in the transformer [20]. There is a family of sound laws [14, 21]. For example a nondeterministic choice between two programs is refined by any probabilistic choice between them

$$\forall\, r : [0,1] \bullet P \sqcap Q \sqsubseteq P \;{}_r\!\oplus\; Q \qquad \text{(introduce probabilism)}$$

For many authors in the area of quantum computation *nondeterminism* means probabilism. However determinism in the semantic space of *pGCL* (a cpo) is equivalent to being an extreme point with respect to the refinement order, and the probabilistic combination of deterministic programs is again deterministic [22]. Thus we write *nondeterminism* to mean *demonic* nondeterminism.

If a set E of expressions contains more than one element then in the guarded-command language the assignment $x :\in E$ means the nondeterministic choice over all individual assignments of elements of E to x. In *pGCL* that choice is interpreted to occur with *uniform* probability.

As we need them we introduce two pieces of derived syntax concerning probabilism: one a prefix combinator — display (4) to follow; the other weakening exact probability r in probabilistic choice to the interval $[r, 1]$ — definition (7) to follow.

5 The language

A *quantum program* is a *pGCL* program invoking quantum procedures. A *quantum procedure* consists of *initialisation* (or state preparation) followed by *evolution* and finally *finalisation* (or observation or state reduction). We now explain each of those three terms.

5.1 Initialisation

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

$$\begin{array}{l} \chi : q(\mathbb{B}^n) \\ \mathit{In}\,(\chi) \;\;\widehat{=}\;\; \chi := 2^{-n/2} \sum_{x:\mathbb{B}^n} \delta_x \, . \end{array}$$

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

$$
\begin{array}{l}
H_n : q(\mathbb{B}^n) \longrightarrow q(\mathbb{B}^n) \\
\begin{array}{rcl}
H_1(\chi)(x) & \widehat{=} & 2^{-1/2}(\chi(0) + (-1)^x \chi(1)) \\
H_{n+1} & \widehat{=} & H_n \otimes H_1 .
\end{array}
\end{array}
$$

5.2 Evolution

Evolution 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.) Again, evolution is feasible: it may be implemented using universal quantum gates [1, 7].

For example on $\mathbb{B}$, 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 conjugate transpose 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). That fact is used in procedure DF in Simon's algorithm and procedure Q in Grover's algorithm.

Later we use this important example of evolution: for function $f : \mathbb{B}^n \longrightarrow \mathbb{B}^n$ between registers, transformation T_f between quregs inverts χ (pointwise) about 0 if f holds and otherwise leaves it unchanged

$$
\begin{array}{l}
T_f : q(\mathbb{B}^n) \longrightarrow q(\mathbb{B}^n) \\
(T_f \chi)(x) \ \widehat{=} \ (-1)^{f(x)} \chi(x) .
\end{array}
\tag{3}
$$

Evidently T_f is unitary.

More complicated evolutions appear in section 6.

5.3 Finalisation

Finalisation is a little more complicated to define and requires some notation. If $[\, (P_j, r_j) \mid 0 \leq j < m \,]$ denotes a finite indexed family of (program, number) pairs with $\sum_{0 \leq j < m} r_j = 1$, then the probabilistic choice in which P_j is chosen with probability r_j is written in prefix form

$$
\oplus[\, P_j \ @ \ r_j \mid 0 \leq j < m \,] \tag{4}
$$

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

Given an indexed family $\mathcal{V} = [\, V_j \mid 0 \leq j < m \,]$ of pairwise orthogonal subspaces which together generate enveloping space,

$$\oplus[\, V_j \mid 0 \leq j < m \,] \;=\; \mathbb{B}^n \longrightarrow \mathbb{C},$$

finalisation is a procedure which reduces state to lie in one of those subspaces, with probability determined by the current state according to a formula involving inner product and the projection $P_V(\chi)$ of state onto subspace V

$$\begin{array}{l} i : 0 \mathinner{..} m,\;\; \chi : q(\mathbb{B}^n) \\ \mathit{Fin}[\mathcal{V}]\,(i,\chi) \;\;\widehat{=}\;\; \oplus[\,(i,\chi :\in \{j\},\, V_j) \mathbin{@} \langle \chi, P_{V_j}(\chi)\rangle \;\mid 0 \leq j < m\,] \end{array}$$

wherein i is a result parameter determining the subspace to which state is reduced and χ is a value-result parameter giving that state. In most cases (and with good physical reason if the observation is not 'complete' — i.e. V_i is more than one-dimensional) χ is not used, in which case we simply suppress it. That definition provides the law 'introduce finalisation'.

A special case of finalisation is important enough to deserve its own notation and consideration of it may help to explain that definition. Let $\mathbb{C}\xi$ denote the subspace of the enveloping space consisting of all scalar mutliples of ξ. If $\Delta = [\,\mathbb{C}\,\delta_x \mid x : \mathbb{B}^n\,]$ denotes the indexed family consisting of the (one-dimensional) subspaces spanned by the standard states, then finalisation is called *diagonal* and written

$$\begin{array}{l} i : \mathbb{B}^n \\ \mathit{Fin}[\Delta]\,(i). \end{array}$$

Its definition simplifies to

$$\oplus[\, x \mathbin{@} |\chi(x)|^2 \;\mid\; x : \mathbb{B}^n \,]$$

and the suppressed value of χ is determined by that of i since $q(\mathbb{B}^n) \cap \mathbb{C}\,\delta_x$ is a singleton. When an output number $i : 0 \mathinner{..} 2^n$ is required, it is produced by applying to $i : \mathbb{B}^n$ the function which yields a number whose binary representation equals its argument

$$\begin{array}{l} \mathit{num} : \mathbb{B}^n \longrightarrow 0 \mathinner{..} 2^n \\ \mathit{num}(x) \;\;\widehat{=}\;\; \sum_{j:0..n} x(j)2^j. \end{array} \tag{5}$$

To clarify the definition of finalisation let us consider diagonal finalisation on $\mathbb{B}$. For $\chi : q(\mathbb{B})$ the inner product simplifies

$$\langle \chi, P_{\mathbb{C}\delta_0}(\chi)\rangle \;=\; \chi(0)\overline{\chi(0)} \;=|\chi(0)|^2$$

and so diagonal finalisation produces the probabilistic assignment

$$\begin{array}{l} i : \mathbb{B} \\ (i := 0) \;\; {}_{|\chi(0)|^2}\!\oplus \;\; (i := 1)\;. \end{array}$$

In other words, diagonal finalisation in state

$$\chi \;=\; \chi(0)\delta_0 + \chi(1)\delta_1$$

returns 0 or 1 and reduces state to δ_0 or δ_1 with probability determined respectively by the modulus squared of the relevant state coefficient, i.e. phase.

That definition accords with general principles of quantum theory (e.g. [11, 16, 24]), which furthermore permit *simultaneous* finalisation (or observation) — i.e. in either order with the same result — if the subspaces in $\mathcal{V}$ are orthogonal. Thus feasibility of that definition is assured by general principles and in particular by Jozsa's characterisation [17] of quantum-observable functions.

It is interesting to note that finalisation is no more restrictive than probabilistic choice. Indeed a simple trigonometric argument shows that $P\;{}_r\!\oplus\;Q$ can be achieved by a quantum program which uses $\oplus$ only in the form defined by finalisation.

For examples of finalisation we proceed to the next section.

6 Example programs

In this section we demonstrate the expressive power of the language by casting in it a representative selection of quantum algorithms and their specifications. Although it is their efficiency which validates these algorithms, we are interested here in formalising their functionality.

6.1 Fair coin

The toss of a fair coin is formalised by specifying the result to be a uniformly-distributed boolean:

$$\begin{array}{l} \textbf{var}\ i : \mathbb{B} \bullet \\ \quad i :\in \mathbb{B} \end{array}$$

A quantum implementation is

$$\begin{array}{l} \textbf{var}\ \chi : q(\mathbb{B}),\ \ i : \mathbb{B} \bullet \\ \quad In\,(\chi)\ ⨟ \\ \quad Fin[\Delta]\,(i) \end{array}$$

which may be checked informally to satisfy its specification since the probability with which $i = 0$ is

$$|\,\chi(0)\,|^2 \;=\; (2^{-1/2})^2 \;=\; \tfrac{1}{2}.$$

Of course a formal proof is immediate from the definitions of initialisation and finalisation.

That program is simple enough to require neither evolution nor phase information about its state. However similar structure, using the Hadamard transform for evolution, can be used to model a beam-splitter: a half-silvered mirror which either transmits or reflects incident photons with equal probability. In that way a quantum program can be written to model a 'Mach-Zehnder interferometer' and, in particular, so-called 'interference-free measurement' [10].

6.2 Grover's point search

The previous program (always) achieves the result required of its probabilistic specification. The next example provides a more typical quantum algorithm which achieves its naïve specification only to within a margin of error.

The point search problem is: given an array f of 2^n bits containing a single 1, locate it. A program which is correct on every execution is specified without any recourse to probability:

$$\begin{array}{l} \textbf{var}\ j : 0\,..\,2^n \bullet \\ \quad j := f^{-1}(1) \end{array} \tag{6}$$

A standard algorithm is at best $O(2^n)$ in both the worst and average case. However Grover's quantum algorithm [12], although correct only to within a margin of error ε (dependent on the number of loop iterations), is $O(\sqrt{N})$ in both those cases. It is conveniently specified by introducing some derived syntax: $P\ {}_{\geq r}\!\oplus\ Q$ is P with probability at least r and is otherwise Q. It is defined

$$P\ {}_{\geq r}\!\oplus\ Q \;\;\widehat{=}\;\; \sqcap\{P\ {}_{s}\!\oplus\ Q \mid r \leq s \leq 1\} \tag{7}$$

which by a semantic convexity argument [21] simplifies to $(P\ {}_r\!\oplus\ Q) \sqcap P$.

The error-prone point-search problem is thus specified to behave, with probability at least ε, like the naïve behaviour (6) and otherwise to terminate with an arbitrary value for j

$$
\begin{array}{l}
\textbf{var}\ j : 0\,..\,2^n \bullet \\
\quad (j := f^{-1}(1))\ {}_{\geq\varepsilon}\!\oplus\ (j \in 0\,..\,2^n)
\end{array}
$$

Grover's implementation contains evolution expressed as a loop and uses the function *num* (see (5).

$$
\begin{array}{l}
\textbf{var}\ \chi : q(\mathbb{B}^n),\ i : \mathbb{B}^n,\ j : 0\,..\,2^n \bullet \\
\quad In\,(\chi)\ ; \\
\quad \textbf{do}\ \ N \text{ times } \longrightarrow \\
\qquad\qquad\qquad \chi := T_f\,(\chi)\ ; \\
\qquad\qquad\qquad \chi := M(\chi) \\
\quad \textbf{od}\ ; \\
\quad Fin[\Delta]\,(i)\ ; \\
\quad j := num(i)
\end{array}
$$

There transform T_f is defined by (3) and transform M inverts χ (pointwise) about its average

$$
\begin{array}{l}
M : q(\mathbb{B}^n) \longrightarrow q(\mathbb{B}^n) \\
(M\chi)(x) \ \ \widehat{=}\ \ 2\,[2^{-n} \sum_{y:\mathbb{B}^n}\ \chi(y)] - \chi(x).
\end{array}
$$

We are not here concerned with the choice of N which determines the number of iterations of the loop. The function $\varepsilon = \varepsilon(N)$ is investigated in [2] and its place in a semantic (expectation-transformer) proof of correctness is explored in [3].

6.3 Deutsch-Jozsa classification

So far we have used only diagonal finalisation. The next example uses non-diagonal finalisation and requires no margin for error.

A truth function $f : \mathbb{B}^n \longrightarrow \mathbb{B}$ is *constant* iff it takes only a single value. It is *balanced* iff it takes values 0 and 1 equally often

$$\#f^{-1}(0) = \#f^{-1}(1).$$

For use in the next section we note:

$$
\begin{aligned}
&f \text{ is constant iff } \#f^{-1}(1) \in \{0, 2^n\}, && (8)\\
&f \text{ is balanced iff } \#f^{-1}(1) = 2^{n-1},\\
&\text{and } \#f^{-1}(1) \;=\; \textstyle\sum_{x:\mathbb{B}^n} f(x).
\end{aligned}
$$

Any constant truth function f is not balanced. So any truth function is either not balanced or not constant, usually both. The Deutsch-Jozsa classification problem is to decide, for a given truth function, which holds; if both hold then either answer is correct.

Letting the result be encoded by variable $i : \mathbb{B}$, the problem is specified

$$
\begin{array}{l}
\textbf{var } i : \mathbb{B} \bullet\\
\quad \begin{array}{lrcl}
\textbf{if} & i & \longrightarrow & f \text{ not balanced}\\
[\,] & \neg\, i & \longrightarrow & f \text{ not constant}\\
\textbf{fi} & & &
\end{array}
\end{array}
$$

A standard algorithm for the Deutsch-Jozsa classification problem is at least $O(2^n)$ in the worst case and on average evaulates f thrice. Deutsch and Jozsa's quantum algorithm [8] contains just *one* evolution step using the transformation T_f defined by equation (3). It is expressed in our notation:

$$
\begin{array}{l}
\textbf{var } \chi : q(\mathbb{B}^n),\ i : \mathbb{B} \bullet\\
\quad In\,(\chi)\ ;\\
\quad \chi := T_f(\chi)\ ;\\
\quad Fin[\mathcal{V}]\,(i)
\end{array}
$$

where finalisation is non-diagonal

$$
\begin{array}{rcl}
\mathcal{V} & \mathrel{\widehat{=}} & [\, V, V^{\perp}\,]\\
V & \mathrel{\widehat{=}} & \mathbb{C} \sum_{y:\mathbb{B}^n} \delta_y\\
V^{\perp} & \mathrel{\widehat{=}} & \text{the orthogonal complement of } V.
\end{array}
$$

A derivation of that algorithm (and hence its correctness) is exhibited in the next section.

6.4 Shor's factorisation algorithm

Shor's quantum algorithms [26] for factorisation and for discrete log are at once the most mathematically sophisticated and relatively efficient practical quantum algorithms known. We consider the former algorithm which, as has been widely advertised, makes factorisation feasible by achieving an average-case polynomial efficiency instead of the standard exponential.

The factorisation problem is: given a natural number $n > 1$ find a prime divisor d of n. It is thus naturally nondeterministic:

$$
\begin{array}{l}
\textbf{var } d : 1 \,..\, (n{+}1) \bullet \\
\quad d \text{ is a prime divisor of } n
\end{array}
$$

For natural numbers x and y, we write $x \sqcup y$ for their maximum and $gcd(x, y)$ for their greatest-common divisor. Shor's algorithm is

$$
\begin{array}{l}
\textbf{var } t : \mathbb{B},\ \ a, d, p : 0 \,..\, (n{+}1) \bullet \\
\quad t := 0 \ ⨾ \\
\quad \textbf{do } \neg t \ \longrightarrow \\
\qquad\quad a :\in 2 \,..\, n \ ⨾ \\
\qquad\quad d := gcd(a, n) \ ⨾ \\
\qquad\quad \textbf{if } d \neq 1 \ \longrightarrow \ \ t := 1 \\
\qquad\quad [] \ \ d = 1 \ \longrightarrow \ \ Q(a, n; p) \ ⨾ \\
\qquad\qquad\qquad\qquad\qquad \textbf{if } p \text{ odd } \ \longrightarrow \ \ t := 1 \\
\qquad\qquad\qquad\qquad\qquad [] \ p \text{ even } \ \longrightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad d := gcd(a^{p/2}{-}1, n) \sqcup gcd(a^{p/2}{+}1, n) \ ⨾ \\
\qquad\qquad\qquad\qquad\qquad\qquad t := (d \neq 1) \\
\qquad\qquad\qquad\qquad\qquad \textbf{fi} \\
\qquad\quad \textbf{fi} \\
\quad \textbf{od}
\end{array}
$$

The quantum content lies in procedure Q. It is our first example to use quantum state after finalisation, though it does so for only standard purposes.

$$
\begin{array}{l}
\textbf{var } \chi : q(\mathbb{B}^m \times \mathbb{B}^m),\ \ x : \mathbb{B}^{2m},\ \ c : \mathbb{B}^m \ \bullet \\
\quad In\,(\chi) \ ⨾ \\
\quad \chi := (I_m \otimes H_m)(\chi) \ ⨾ \\
\quad \chi := E(\chi) \ ⨾ \\
\quad \chi := (F_m \otimes I_m)(\chi) \ ⨾ \\
\quad Fin[\Delta]\,(x, \chi) \ ⨾ \\
\quad c := P_m(\chi) \ ⨾ \\
\quad p := \text{post processing of } c
\end{array}
$$

where:

m satisfies $n^2 \leq 2^m \leq 2n^2$;

unitary transformation $E : q(\mathbb{B}^m \times \mathbb{B}^m) \longrightarrow q(\mathbb{B}^m \times \mathbb{B}^m)$ is defined in terms of modular exponentiation

$$E(\chi)(x, y) \;\widehat{=}\; (x, y \oplus num^{-1}(a^{num(x)} \bmod n));$$

$F_m : q(\mathbb{B}^m) \longrightarrow q(\mathbb{B}^m)$ is the quantum Fourier transform (see [13] section 3.2.2);

finalisation has been extended to return also state χ ;

$P_m : \delta(\mathbb{B}^m \times \mathbb{B}^m) \longrightarrow \mathbb{B}^m$ denotes a kind of projection

$$P_m(\delta_x \otimes \delta_y) \;\widehat{=}\; x \; ; \text{ and}$$

the post processing of c is standard, using continued fractions to find efficiently the unique p for which

$$\mid num(c)/2^m - d/p \mid \;\leq\; 2^{-(m+1)}.$$

The first two lines of procedure Q are equivalent (see section 5.2) to

$$\chi := (H_m \otimes I_n)(\delta_{\mathbf{0}})$$

(where $\delta_{\mathbf{0}}$ denotes the qureg containing $m+n$ zeroes); however our insistence that quantum programs begin with a standard initialisation obliges us to take the longer version.

Simon's quantum algorithm for his masking problem [27] is similar in structure, from our current point of view, to Shor's factorisation algorithm and so we omit it.

6.5 Finite automaton

We now consider another example which uses quantum state after finalisation Recall that (standard) finite automata, whether deterministic or not and one-way or two-way, accept just regular languages. For quantum finite automata enough is already known to demonstrate that the picture is quite different (see [13], chapter 4).

A *one-measurement one-way quantum finite automaton* employs finalisation after reading its input string and so is readily expressed in our notation. So instead we consider the *many-measurement* version which employs finalisation after reading each symbol on its input string. It turns out (Kondacs and Watrous; see, for example, [13] p. 159) that a many-measurement one-way quantum finite automaton accepts a proper subset of regular expressions. Here we give sufficient of the definition to permit its translation into our programming language.

For any set Σ, let Σ^* denote the set of all finite sequences of elements of Σ. Suppose that set $\mathcal{S} = \{S_a, S_r, S_n\}$ of subsets of Σ^* *partitions* Σ^*. A sequence $s : \Sigma^*$ is said to be *accepted* by $\mathcal{S}$ if $s \in S_a$, to be *rejected* by it if $s \in S_r$ and to *fail classification* if $s \in S_n$. Evaluation of that is specified:

$$\begin{array}{l} \mathbf{var}\ i : \{a, r, n\} \bullet \\ \quad s \in S_i. \end{array}$$

But here we are interested in computing whether a prefix of a given sequence is accepted or rejected, since that gives rise to an automaton which continues to use its quantum state after finalisation. Its specification thus extends the previous one. In it $t \leq s$ means that sequence t is a prefix of sequence s.

$$\begin{array}{l} \mathbf{var}\ i : \{a, r, n\} \bullet \\ \quad \left(\begin{array}{l} i = a \Rightarrow \exists\, t \leq s \bullet t \in S_a \\ i = r \Rightarrow \exists\, t \leq s \bullet t \in S_r \\ i = n \Rightarrow s \in S_n. \end{array} \right) \end{array}$$

(A stronger specification might reflect the fact that computation proceeds from left to right and so ensure that sequence t there is smallest.)

A *many-measurement one-way quantum finite automaton* is designed to achieve such a computation with efficient quantum evolution. It has a finite set Q of states, with distinguished acceptance states Q_a and rejection states Q_r

$$\begin{array}{l} Q_a \subseteq Q \\ Q_r \subseteq Q \\ Q_a \cap Q_r = \{\,\}. \end{array}$$

Thus $Q_a \cup Q_r$ need not exhaust Q.

On input sequence $s = [\sigma_0, \ldots, \sigma_{n-1}] : \Sigma^*$ the automaton evolves successively under unitary transformations

$$U_{init}, U_{\sigma_0}, \ldots, U_{\sigma_{n-1}}$$

subject to finalisation after each. If a finalisation leaves the automaton in an acceptance state then computation terminates with output value $i = a$; if it leaves the automaton in a rejection state then computation terminates with output value $i = r$; but otherwise the automaton reiterates. If it has not accepted or rejected a prefix of the input sequence, it terminates when the entire sequence has been read, with value $i = n$.

We thus let quantum state belong to $q(Q)$, defined as was qureg state by (1). Initialisation over $q(Q)$ is defined as for registers; its feasibility is assured by solubility of the appropriate simultaneous equations describing a unitary transformation that yields a uniform image of $\delta_{\mathbf{0}}$. For finalisation we take

$$\begin{array}{rcl} \mathcal{V} & \widehat{=} & [\, V_a, V_r, V_n \,] \\ V_a & \widehat{=} & span\{\delta_x \mid x \in Q_a\} \\ V_r & \widehat{=} & span\{\delta_x \mid x \in Q_r\} \\ V_n & \widehat{=} & (V_a \oplus V_r)^{\perp} \end{array}$$

where $spanE$ denotes the (complex) vector space generated by any subset E of enveloping space.

A program for such an automaton is

$$\begin{array}{l} \mathbf{var}\ \chi : q(Q),\ \ b : \mathbb{B} \bullet \\ \quad In(\chi) \ ⨾ \\ \quad \chi, b := U_{init}(\chi), 0 \ ⨾ \\ \quad \mathbf{do}\ \ \neg(b \vee s = [\,]) \ \longrightarrow \\ \qquad\qquad\qquad Fin[\mathcal{V}]\,(i,\chi) \ ⨾ \\ \qquad\qquad\qquad \mathbf{if}\ \ i \in \{a, r\} \ \longrightarrow \ b := 1 \\ \qquad\qquad\qquad [\,]\ i \notin \{a, r\} \ \longrightarrow \ \chi, s := U_{head(s)}(\chi), tail(s) \\ \qquad\qquad\qquad \mathbf{fi} \\ \quad \mathbf{od} \end{array}$$

That can be expressed only because we allow quantum state to be returned by finalisation.

7 Example derivation

We conclude by outlining an algebraic derivation of the Deutsch-Jozsa classification algorithm. It is to be emphasised that derivations (or verifications) using the refinement calculus (i.e. the laws concerning refinement between programs — see for example [18]) are quite different in style from those phrased in terms of semantics (c.f. [3]). We are interested primarily in the shape of the derivation; and we shall see that it is largely standard.

The following derivation can be followed intuitively as well as rigorously; the steps involved correspond largely to steps in an informal explanation of why the algorithm works. At one point $\sqsubseteq$ is extended to mean also data refinement.

$$
\begin{array}{l}
\textbf{var } i : \mathbb{B} \bullet \\
\quad \begin{array}{lrcl}
\textbf{if} & i & \longrightarrow & f \text{ not balanced} \\
[] & \neg\, i & \longrightarrow & f \text{ not constant} \\
\textbf{fi} & & &
\end{array}
\end{array}
$$

$\sqsubseteq$ (8) and standard reasoning

$$
\begin{array}{l}
\textbf{var } i : \mathbb{B},\ j : 0\,..\,2^n \bullet \\
\quad j := \sum_{x:\mathbb{B}^n} f(x)\ ; \\
\quad \begin{array}{lrcl}
\textbf{if} & j \neq 2^{n-1} & \longrightarrow & i := 1 \\
[] & j \notin \{0, 2^n\} & \longrightarrow & i := 0 \\
\textbf{fi} & & &
\end{array}
\end{array}
$$

$\sqsubseteq$ standard reasoning

$$
\begin{array}{l}
\textbf{var } i : \mathbb{B},\ j : 0\,..\,2^n \bullet \\
\quad j := \sum_{x:\mathbb{B}^n} f(x)\ ; \\
\quad \begin{array}{lrcl}
\textbf{if} & j \in \{0, 2^n\} & \longrightarrow & i := 1 \\
[] & j = 2^{n-1} & \longrightarrow & i := 0 \\
[] & j \notin \{0, 2^{n-1}, 2^n\} & \longrightarrow & (i := 1) \sqcap (i := 0) \\
\textbf{fi} & & &
\end{array}
\end{array}
$$

$\sqsubseteq$ arithmetic and standard and probabilistic reasoning

$$
\begin{array}{l}
\textbf{var } i : \mathbb{B},\ j : 0\,..\,2^n \bullet \\
\quad j := \sum_{x:\mathbb{B}^n} f(x)\ ; \\
\quad \begin{array}{lrcl}
\textbf{if} & j \in \{0, 2^{n-1}, 2^n\} & \longrightarrow & (i := 1)\ {}_{|1-j/2^{n-1}|}\!\oplus (i := 0) \\
[] & j \notin \{0, 2^{n-1}, 2^n\} & \longrightarrow & (i := 1) \sqcap (i := 0) \\
\textbf{fi} & & &
\end{array}
\end{array}
$$

$\sqsubseteq$ injective data refinement $k = 1 - j/2^{n-1}$

$$
\begin{array}{l}
\quad \mathbf{var}\ i : \mathbb{B},\ \ k : [-1,1] \bullet \\
\qquad k := 2^{-n} \sum_{x:\mathbb{B}^n} (-1)^{f(x)} \ ⨾ \\
\qquad \mathbf{if} \quad k \in \{-1,0,1\} \quad \longrightarrow \quad (i := 1)\ {}_{|k|}\oplus\ (i := 0) \\
\qquad [] \quad\ \ k \notin \{-1,0,1\} \quad \longrightarrow \quad (i := 1) \sqcap (i := 0) \\
\qquad \mathbf{fi} \\
\sqsubseteq \qquad\qquad\qquad\qquad\qquad\qquad \text{'introduce probabilism'} \\
\quad \mathbf{var}\ i : \mathbb{B},\ \ k : [-1,1] \bullet \\
\qquad k := 2^{-n} \sum_{x:\mathbb{B}^n} (-1)^{f(x)} \ ⨾ \\
\qquad (i := 1)\ {}_{|k|}\oplus\ (i := 0) \\
\sqsubseteq \qquad\qquad\qquad\qquad\qquad\qquad \text{'sequential composition'} \\
\quad \mathbf{var}\ i : \mathbb{B},\ \ k : [-1,1],\ \ \chi : q(\mathbb{B}^n) \bullet \\
\qquad \chi = 2^{-n/2} \sum_{x:\mathbb{B}^n} (-1)^{f(x)} \delta_x \ ⨾ \\
\qquad k := 2^{-n/2} \sum_{x:\mathbb{B}^n}\ \ \chi(x) \ ⨾ \\
\qquad (i := 1)\ {}_{|k|}\oplus\ (i := 0) \\
\sqsubseteq \qquad\qquad\qquad\qquad \text{'sequential composition' and definition of } T_f \\
\quad \mathbf{var}\ i : \mathbb{B},\ \ k : [-1,1],\ \ \chi : q(\mathbb{B}^n) \bullet \\
\qquad \chi := 2^{-n/2} \sum_{x:\mathbb{B}^n}\ \delta_x \ ⨾ \\
\qquad \chi := T_f(\chi) \ ⨾ \\
\qquad k := 2^{-n/2} \sum_{x:\mathbb{B}^n}\ \ \chi(x) \ ⨾ \\
\qquad (i := 1)\ {}_{|k|}\oplus\ (i := 0) \\
\sqsubseteq \qquad\qquad\qquad\qquad \text{diminish by } k = \langle \chi, 2^{-n/2} \sum_{x:\mathbb{B}^n} (-1)^{f(x)} \delta_x \rangle \\
\quad \mathbf{var}\ i : \mathbb{B},\ \ \chi : q(\mathbb{B}^n) \bullet \\
\qquad \mathit{In}\,(\chi) \ ⨾ \\
\qquad \chi := T_f(\chi) \ ⨾ \\
\qquad \mathit{Fin}[\mathcal{V}]\,(i)
\end{array}
$$

with family $\mathcal{V} = [\,V\,,\,V^{\perp}\,]$, where $V = \mathbb{C} \sum_{y:\mathbb{B}^n} \delta_y$.

8 Conclusions

A general-purpose quantum computer will require a programming language sharing the features of successful programming languages of its day. Abstraction from implementation concerns (like the representation of a function f on the underlying standard types by its Lecerf-Bennett form on their quantum analogues) is then supported by a process of compilation (considered elsewhere [28]) with consequent simplification of the programs themselves.

The language proposed here supports that endeavour with these features:

1. feasibility: each program is executable
2. expressivity: the language is sufficiently expressive to capture existing quantum algorithms
3. simplicity: the language seems to be as simple as possible whilst containing nondeterminism and probability (the latter either in a form restricted to 'observation' or for general use)
4. abstraction: the language contains control structures and data structures at the level of abstraction of today's imperative languages whilst abstracting implementation concerns
5. calculation: the language has a formal semantics, sound laws and provides a refinement calculus supporting verification and derivation of programs
6. the language provides a uniform treatment of 'observation'.

References

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

[1] Adriano Barenco. A universal two-bit gate for quantum computation. *Proc. R. Soc. Lond.* A, **449**:679–683, 1995.

[2] Michel Boyer, Gilles Brassard, Peter Hoyer and Alain Tapp. Tight bounds on quantum searching. In *Fourth Workshop on Physics and Computation*, editors T. Toffoli, M. Biaford and J. Lean, pages 36–43. New England Complex System Institute, 1996.

[3] Michael Butler and Pieter Hartel. Reasoning about Grover's quantum search algorithm using probabilistic wp. University of Southampton technical report DSSE-TR-98-10, 1998.

[4] R. Cleve, A. Ekert, C. Macchiavello and M. Mosca. Quantum algorithms revisited. *Proc. R. Soc. Lond.*, A. **454**:339–354, 1998.

[5] D. Deutsch. Quantum theory, the Church-Turing principle and the universal quantum computer. *Proc. R. Soc. Lond.* A, **400**:97–117, 1985.

[6] D. Deutsch. Quantum computational networks. *Proc. R. Soc. Lond.* A, **425**:73–90, 1989.

[7] David Deutsch, Adriano Barenco and Artur Ekert. Universality in quantum computation. *Proc. R. Soc. Lond.* A, **449**:669–677, 1995.

[8] David Deutsch and Richard Jozsa. Rapid solution of problems by quantum computation. *Proc. R. Soc. Lond.* A, **439**:553–558, 1992.

[9] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall International, 1976.

[10] Avshalom C. Elitzur and Lev Vaidman. Quantum mechanical interaction-free measurements. *Foundations of Physics*, **32**(7):987–997, 1993.

[11] R. P. Feynman. *The Feynman Lectures on Physics*, volume 3. Addison-Wesley, 1964.

[12] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the 28th ACM STOC*, pages 212–219, 1996.

[13] Jozef Gruska. *Quantum Computing.* McGraw-Hill International (UK), Advanced Topics in Computer Science, 1999.

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

[15] C. A. R. Hoare He Jifeng. The weakest prespecification, parts I and II. Fundamenta Informatica, IX, 51–84, 217–252, 1986.

[16] Chris J. Isham. *Lectures on Quantum Theory.* Imperial College Press, 1995.

[17] Richard Josza. Characterising classes of functions computable by quantum parallelism. *Proc. R. Soc. Lond.* A, **435**:563–574, 1991.

[18] Carroll Morgan. *Programming from Specifications*, second edition. Prentice-Hall International, 1994.

[19] K. Seidel, C. C. Morgan and A. K. McIver. Probabilistic imperative programming: a rigorous approach. 1996. Available at `http//www.comlab.ox.ac.uk/oucl/groups/probs/bibliography.html`.

[20] Carroll Morgan, Annabelle McIver and Karen Seidel. Probabilistic predicate transformers. *TOPLAS*, **18**(3):325–353, 1996.

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

[22] Carroll Morgan and A. K. McIver. Demonic, angelic and unbounded probabilistic choices in sequential programs. To appear in *Acta Informatica.*

[23] Michele Mosca and Artur Ekert. The hidden subgroup problem and eigenvalue estimation on a quantum computer. In *Proceedings of the 1st NASA Internation Conference on Quantum Compuing and Quantum Communication.* LNCS 1509, Springer-Verlag, 1998.

[24] Asher Peres. *Quantum Theory: Concepts and Methods.* Kluwer Academic Publishers, 1998.

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

[26] Peter W. Shor. Algorithms for quantum computation: discrete log and factoring. In *Proceedings of the 35th IEEE FOCS*, pages 124–134, 1994.

[27] Daniel R. Simon. On the power of quantum computation. In *Proceedings of the 35th IEEE FOCS*, pages 116–123, 1994.

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