Towards a Quantum Programming Language

Peter Selinger

Department of Mathematics, University of Ottawa

Abstract

The field of quantum computation suffers from a lack of syntax. In the absence of a convenient programming language, algorithms are frequently expressed in terms of hardware circuits or Turing machines. Neither approach particularly encourages structured programming or abstractions such as data types. In this paper, we describe the syntax and semantics of a simple quantum programming language. This language provides high-level features such as loops, recursive procedures, and structured data types. It is statically typed, and it has an interesting denotational semantics in terms of complete partial orders of superoperators.

1 Introduction

Quantum computation is traditionally studied either at the level of gates and circuits, or in terms of quantum Turing machines. The former viewpoint emphasizes data flow and neglects control flow; indeed, control mechanisms are usually dealt with at the meta-level, as a set of instructions on how to construct a parameterized family of quantum circuits. On the other hand, quantum Turing machines can express both data flow and control flow, but in a sense that is sometimes considered too general to have practical applications outside of complexity theory.

In this paper, we seek to investigate quantum computation from the point of view of programming languages. We propose a view of quantum computation which is able to express both data flow and control flow, while not relying on any particular hardware model. Our approach can be summarized by the slogan "quantum data, classical control". Thus, the data which programs manipulate may involve quantum superpositions, but the control state of a program is always classical; there is no "quantum branching" and no notion of executing a quantum superposition of two different statements. This is more general than quantum circuits, where control flow is not modeled at all, but more restrictive than quantum Turing machines, where both data and control may be "quantum". The paradigm "quantum data, classical control" seems to be precisely what is embodied in most known practical quantum algorithms, such as Shor's factoring algorithm or the Quantum Fourier Transform.

The programming language presented in this paper is *functional*, in the sense that each (atomic or composite) statement operates by transforming a specific set of inputs to outputs. This is in contrast to *imperative* programming languages, which operate by updating global variables. Our language is also *statically typed*, which implies that the well-formedness of a program can be checked at compile time, rather than relying on run-time checks. A well-typed program is guaranteed to be free of run-time errors. The language provides high-level control features such as loops and recursion, and it can accommodate structured data types such as lists or trees. Syntactically, we give two alternative representations of quantum programs: a graphical representation in terms of flow charts, and a textual, more structured representation. The choice of syntax is a matter of taste, since semantically, the two representations are equivalent.

Perhaps the most important feature of our programming language is that is admits a *denotational semantics*. This is achieved by assigning a *superoperator* to each program fragment. For the semantics of loops and recursion, we use the fact that the superoperators of any given type form a complete partial order. The denotational semantics is *fully abstract* in the sense that two program fragments are denotationally equivalent if and only if they are indistinguishable in the context of any larger program.

While the semantics of our quantum programming language can (and will) be described without any reference to any particular hardware model, it helps the intuition to think of a particular hardware device on which the language might be implemented. Here, it is understood that actual future quantum hardware may differ; these differences must be handled by implementors and are not part of the language design. A suitable hardware model for our purposes is the *QRAM* machine of [8]. This machine consists of a general-purpose classical computer which controls a special quantum hardware device. The quantum device provides a potentially large number of individually addressable quantum bits. Quantum bits can be manipulated via two fundamental operations: (1) unitary transformations and (2) measurements. Typically, the quantum device will implement a fixed, finite set of unitary transformations which operate on one or two quantum bits at a time. The classical controller communicates with the quantum device by sending a sequence of instructions, specifying which fundamental operations are to be performed. The only output from the quantum device consists of the results of measurements, which are sent back to the classical controller. Note that in the QRAM model, it is not necessary for the classical hardware and the quantum device to be in the same physical location; it is even possible for several classical controllers to share access to a single quantum device.

In quantum complexity theory, algorithms are often presented in a certain normal form: a quantum computation consists of an initialization, followed by a unitary transformation, followed by a single measurement at the very end, just before the classical result of the algorithm is read. While no theoretical generality is lost in postponing all measurements until the end, this presentation does not necessarily lead to the most natural programming style. Quantum algorithms are often more naturally described by interleaving quantum and classical operations, allowing the results of measurements to influence subsequent operations. In addition to being conceptually more natural, this style of programming can also lead to savings in the resources consumed by an algorithm, for instance, in the number of quantum bits that must be allocated.

One issue not addressed in this paper is the question of *quantum communication*. The programming language described here deals with quantum computation in a closed world, i.e., it describes a single process which computes a probabilistic result depending only on its initial input parameters. We do not deal with issues related to the passing of quantum information, or even classical information, between different quantum processes. In particular, our programming language currently does not support primitives for input or output, or other side effects such as updating shared global data structures. While adding such features would not cause any problems from an operational point of view, the denotational semantics of the resulting language is not currently very well understood. In other words, in a setting with side effects, it is not clear how the behavior of a complex system can be described in terms of the behaviors of its individual parts. These difficulties are on the one hand typical of the passage from purely functional programming languages to concurrent languages. On the other hand, there are particular complications that arise from the very nature of quantum information flow, which is a subject of ongoing research.

Another issue that we do not address in this paper is the issue of "idealized" vs. "real" hardware. Just as in classical programming language theory, we assume that our language runs on idealized hardware, i.e., hardware that never produces flawed results, where quantum states do not deteriorate, where no unintended information is exchanged between programs and their environment, etc. This seems a reasonable assumption to make from the viewpoint of language design, although it might only be approximated in actual implementations. It is intended that implementations will use quantum error correction techniques to limit the adverse effects of imperfect physical hardware. Error correction can potentially be handled, transparently to the programmer, at several different levels: it could be built into the hardware, it could be handled by the operating system, or it could be added automatically by the compiler. The tradeoffs associated with each of these choices remain the subject of future work, and we do not consider them further in this paper.

Since there are considerable differences between the languages of physics and computer science, it seems appropriate to comment on the notation used in this paper. The paper was primarily written with an eye towards computer scientists as the "intended" audience. The author has therefore chosen to use computer science notation, rather than physics notation, throughout the paper. For instance, operators are represented as matrices, the bra-ket notation is avoided, and contrary to the usual practice in physics, state vectors are written as column vectors. It is nevertheless hoped that the paper will remain readable to readers with a background in physics.

Previous work

The recent literature contains several proposals for quantum programming languages. One of the first contributions in this direction is an article by Knill [8]. While not proposing an actual programming language, Knill outlines a set of basic principles for writing pseudo-code for quantum algorithms. These principles have influenced the design of some later language designs. The first actual quantum programming language is due to Ömer [9]. Ömer defines a rich procedural language QCL, which contains a full-fledged classical sublanguage and many useful high-level quantum features, such as automatic scratch space management and syntactic reversibility of user-defined quantum operators. Partly building on Ömer's work, Bettelli et al. [2] present a quantum programming language which is an extension of the well-known language C++. The distinguishing feature of this language is that it treats quantum operators as first-class objects which can be explicitly constructed and manipulated at run-time, even allowing run-time optimizations of operator representations.

A quantum programming language of a somewhat different flavor is given by Sanders and Zuliani [11]. Their language qGCL, which is based on an extension of Dijkstra's guarded-command language, is primarily useful as a specification language. Its syntax includes high-level mathematical notation, and the language supports a mechanism for stepwise refinement which can be used for systematic program derivation and verification.

Of the languages surveyed, only that of Sanders and Zuliani possesses a formal semantics. This semantics is purely operational, and it works by assigning to each possible input state a probability distribution on output states. One problem with this approach is that the resulting probability distributions are generally very large, incorporating much information which is not physically observable. The present paper offers a better solution to this problem, using density matrices and superoperators instead of probability distributions.

A common feature of the quantum programming languages surveyed in the literature is that they are realized as *imperative* programming languages. Quantum data is manipulated in terms of *arrays* of quantum bit references. This style of data representation requires the insertion of a number of run-time checks into the compiled code, including out-of-bounds checks and distinctness checks. For instance, distinctness checks are necessary because quantum operators can only be applied to lists of *distinct* quantum bits. In imperative languages, this condition cannot be checked at compile-time. For similar reasons, most optimizations cannot be performed at compile-time and must therefore be handled at run-time in these languages. By contrast, the language proposed in the present paper is a *functional* programming language with a static type system which guarantees the absence of any run-time errors.

2 Basic notions from linear algebra

We start by recalling a few well-known concepts from linear algebra. Readers may wish to skip this section and refer back to it when needed.

2.1 Vectors and matrices

Let $\mathbb{C}$ be the set of complex numbers, which are also called scalars. The complex conjugate of $x \in \mathbb{C}$ is written $\bar{x}$. We write $\mathbb{C}^n$ for the space of n-dimensional column vectors, and $\mathbb{C}^{n\times m}$ for the space of matrices with n rows and m columns. We identify $\mathbb{C}^n$ with $\mathbb{C}^{n\times 1}$ and $\mathbb{C}^1$ with $\mathbb{C}$. Matrix multiplication is defined as usual. The identity matrix is written I. The adjoint of a matrix $A = (a_{ij}) \in \mathbb{C}^{n\times m}$ is given by $A^* = (\bar{a}_{ji}) \in \mathbb{C}^{m\times n}$.

Column vectors are denoted by u, v, etc. We write e_i for the ith canonical basis vector. Note that if u is a column vector, then u^* is a row vector. The vector u is called a *unit vector* if $u^*u = 1$.

If A, B, C and D are matrices of equal dimensions, we often denote the matrix obtained by "horizontal and vertical concatenation" by

$$\left(\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right).$$

Sometimes, we also use an analogous notation for vectors.

2.2 Trace and norm.

The *trace* of a square matrix $A = (a_{ij}) \in \mathbb{C}^{n\times n}$ is defined as $\mathrm{tr}\, A = \sum_i a_{ii}$. Note that for $A, B \in \mathbb{C}^{n\times n}$, $\mathrm{tr}(AB) = \mathrm{tr}(BA)$. The *norm* $|A|$ of a matrix A is defined by $|A|^2 = \mathrm{tr}(A^*A) = \sum_{ij} \bar{a}_{ij} a_{ij}$.

2.3 Unitary matrices

A square matrix $S \in \mathbb{C}^{n\times n}$ is called *unitary* if $S^*S = I$ is the identity matrix, or equivalently, if $S^* = S^{-1}$. If S is unitary and $A = SBS^*$, then $\mathrm{tr}\, A = \mathrm{tr}\, B$ and $|A| = |B|$. Thus, trace and norm are invariant under a unitary change of basis.

2.4 Hermitian and positive matrices

A square matrix $A \in \mathbb{C}^{n\times n}$ is called *hermitian* if $A = A^*$. Note that if A is hermitian, then u^*Au is always real. A matrix A is called *positive hermitian*, or simply *positive*, if it is hermitian and $u^*Au \geqslant 0$ for all $u \in \mathbb{C}^n$. Note that hermitian matrices are closed under addition and real scalar multiples, i.e., they form an $\mathbb{R}$-linear subspace of $\mathbb{C}^{n\times n}$. Moreover, the positive matrices are closed under addition and non-negative real scalar multiples.

A matrix A is hermitian if and only if $A = SDS^*$, for some unitary matrix S and some real-valued diagonal matrix D. The diagonal entries of D are the eigenvalues of A, and they are uniquely determined up to a permutation. The columns of S are the corresponding eigenvectors. Moreover, a hermitian matrix A is positive iff all its eigenvalues are non-negative.

A matrix $A \in \mathbb{C}^{n\times n}$ is called *pure* if $A = vv^*$ for some $v \in \mathbb{C}^n$. Every pure matrix is positive; conversely, every positive matrix is a non-negative real linear combination of pure matrices.

Remark 2.1. If A is positive hermitian, then $|A| \leqslant \operatorname{tr} A$. It suffices to prove this property for diagonal matrices, where it follows trivially from the fact that $\lambda_1^2 + \ldots + \lambda_n^2 \leqslant (\lambda_1 + \ldots + \lambda_n)^2$, for $\lambda_1, \ldots, \lambda_n \geqslant 0$.

Remark 2.2. Any complex $\mathbb{C}^{n\times n}$ matrix is a linear combination of four positive hermitian matrices. Because first, any complex matrix A can be written as a linear combination $B + iC$ of two hermitian matrices, where $B = \frac{1}{2}(A^* + A)$ and $C = \frac{i}{2}(A^* - A)$. Second, any hermitian matrix B can be written as the difference of two positive matrices, namely $B = (B + \lambda I) - (\lambda I)$, where $-\lambda$ is the most negative eigenvalue of B.

Remark 2.3. Every positive matrix $A \in \mathbb{C}^{n\times n}$ is of the form BB^*, for some $B \in \mathbb{C}^{n\times n}$. This is evident for diagonal matrices, and follows for arbitrary positive matrices by a change of basis.

2.5 Tensor product

The tensor product of two vector spaces is defined as usual; here we only need two special cases: $\mathbb{C}^n \otimes \mathbb{C}^m = \mathbb{C}^{nm}$ and $\mathbb{C}^{n\times n} \otimes \mathbb{C}^{m\times m} = \mathbb{C}^{nm\times nm}$.

The tensor product $w = u \otimes v \in \mathbb{C}^{nm}$ of two vectors $u \in \mathbb{C}^n$, $v \in \mathbb{C}^m$ is defined by $w_{(i,j)} = u_i v_j$. Similarly, the tensor product $C = A \otimes B \in \mathbb{C}^{nm\times nm}$ of two matrices is defined by $c_{(i,j),(i',j')} = a_{ii'} b_{jj'}$. Here we order the pairs (i, j) lexicographically; thus, for instance,

$$\begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \otimes B = \left(\begin{array}{c|c} 0 & B \\ \hline -B & 0 \end{array}\right).$$

3 Basic notions from quantum computation

In this section, we very briefly summarize the basic notions of quantum computation. For a more thorough introduction, see e.g. [3, 5, 10].

3.1 Quantum bits

The basic data type of quantum computation is a *quantum bit*, also called a *qubit* or, even more succinctly, a *qbit*. Recall that the possible states of a classical bit b are $b = \mathbf{0}$ and $b = \mathbf{1}$. By contrast, the state of a quantum bit q can be any complex linear combination $q = \alpha\mathbf{0} + \beta\mathbf{1}$, where $\alpha, \beta \in \mathbb{C}$ and α, β are not both zero. The coefficients α and β are called the *amplitudes* of this quantum state, and they are only significant up to scalar multiples, i.e., $q = \alpha\mathbf{0} + \beta\mathbf{1}$ and $q' = \alpha'\mathbf{0} + \beta'\mathbf{1}$ denote the same quantum state if $\alpha' = \gamma\alpha$ and $\beta' = \gamma\beta$ for some non-zero $\gamma \in \mathbb{C}$. One often assumes that the amplitudes α and β are normalized, i.e., that $|\alpha|^2 + |\beta|^2 = 1$. Note, however, that this normalization does not determine α and β uniquely; they are still only well-defined up to multiplication by a complex unit.

The basis states $q = \mathbf{0}$ and $q = \mathbf{1}$ are called the *classical* states, and any other state is said to be a *quantum superposition* of $\mathbf{0}$ and $\mathbf{1}$. In the literature, quantum states are often written in the so-called "ket" notation as $\alpha\,|\mathbf{0}\rangle + \beta\,|\mathbf{1}\rangle$. Here "ket" is the second half of the word "bracket". However, we do not use this notation here; it suffices to keep in mind that bold-face $\mathbf{0}$ and $\mathbf{1}$ are boolean constants, rather than scalars.

The first interesting fact about quantum bits is that the state of two or more quantum bits is not just a tuple of its components, as one might have expected. Recall the four possible states of a pair of classical bits: $\mathbf{00}$, $\mathbf{01}$, $\mathbf{10}$, and $\mathbf{11}$. The state of a pair of quantum bits is a formal complex linear combination of these four classical states, i.e., it is a linear combination of the form

$$\alpha_{00}\mathbf{00} + \alpha_{01}\mathbf{01} + \alpha_{10}\mathbf{10} + \alpha_{11}\mathbf{11} = \sum_{i,j=\mathbf{0}}^{\mathbf{1}} \alpha_{ij}\mathbf{ij}.$$

As before, at least one of the α_{ij} must be non-zero, and the state is only well-defined up to a complex scalar multiple. If $q = \alpha\mathbf{0} + \beta\mathbf{1}$ and $p = \gamma\mathbf{0} + \delta\mathbf{1}$ are two independent quantum bits, then their combined state is

$$q \otimes p = \alpha\gamma\mathbf{00} + \alpha\delta\mathbf{01} + \beta\gamma\mathbf{10} + \beta\delta\mathbf{11}.$$

However, note that the general state of a pair of quantum bits need not be of the form $q \otimes p$. For instance, the state

$$\frac{1}{\sqrt{2}}\mathbf{00} + \frac{1}{\sqrt{2}}\mathbf{11}$$

is clearly not of the form $q \otimes p$, for any q and p. If the state of a pair of quantum bits is of the form $q \otimes p$, then the pair is said to be *independent*, otherwise it is called *entangled*.

In general, the state of n quantum bits is a non-zero vector in $\mathbb{C}^{2^n}$, i.e., a formal linear combination

$$\sum_{i_1,\ldots,i_n=\mathbf{0}}^{\mathbf{1}} \alpha_{i_1\ldots i_n}\mathbf{i}_1\ldots\mathbf{i}_n,$$

taken modulo scalar multiples.

3.2 Indexing conventions

Consider a quantum state $q = \alpha\mathbf{00} + \beta\mathbf{01} + \gamma\mathbf{10} + \delta\mathbf{11}$. By identifying the basis vectors $\mathbf{00}$, $\mathbf{01}$, $\mathbf{10}$, and $\mathbf{11}$ with the canonical basis of $\mathbb{C}^4$, we can write the state q as a column vector

$$q = \begin{pmatrix} \alpha \\ \beta \\ \gamma \\ \delta \end{pmatrix}.$$

Here, the ordering of the basis vectors is relevant. In general, we need an indexing convention which determines how the bit vectors of length n are to be identified with the canonical basis vectors of $\mathbb{C}^{2^n}$.

Convention 3.1. Consider the set of bit vectors of length n, i.e., n-tuples of the elements $\{0, 1\}$. We identify each such bit vector with the number $i \in 0, \ldots, 2^n - 1$ of which it is the binary representation, and also with the ith canonical basis vector of $\mathbb{C}^{2^n}$.

Note that this convention implies that we always order bit vectors lexicographically when they are used to index the rows or columns of some vector or matrix.

Sometimes, we need to consider a permutation of quantum bits, for instance, exchanging the first with the second quantum bit in a sequence. This induces a corresponding permutation of states. More precisely, any permutation of n elements, $\phi : \{1, \ldots, n\} \to \{1, \ldots, n\}$, induces a permutation 2^ϕ of the set of bit vectors of length n, which is defined by $2^\phi(x_1, \ldots, x_n) = (x_{\phi^{-1}(1)}, \ldots, x_{\phi^{-1}(n)})$, for $x_1, \ldots, x_n \in \{\mathbf{0}, \mathbf{1}\}$.

3.3 Unitary transformations

There are precisely two kinds of operations by which we can manipulate a quantum state: unitary transformations and measurements. We describe unitary transformations in this section and measurements in the next one.

The state of a quantum system can be transformed by applying a unitary transformation to it. For instance, consider a quantum bit in state $u = \alpha\mathbf{0} + \beta\mathbf{1}$, and let S be a unitary 2×2-matrix. Then we can perform the operation

$$\begin{pmatrix} \alpha \\ \beta \end{pmatrix} \mapsto S \begin{pmatrix} \alpha \\ \beta \end{pmatrix}.$$

Similarly, if $v = \alpha\mathbf{00} + \beta\mathbf{01} + \gamma\mathbf{10} + \delta\mathbf{11}$ is the state of a 2-qbit quantum system, we can transform it by a unitary 4×4-matrix, and similarly for three or more quantum bits. A unitary operation on n quantum bits is also known as an n-ary *quantum gate*. While every unitary matrix can be realized in principle, one usually assumes a fixed finite set of gates that are built into the hardware. The particular choice of basic gates is not important, because a compiler will easily be able to translate one such set to another. One possible choice is the following, which consists of four unary and four binary gates:

$$N = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \qquad H = \tfrac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \qquad V = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}, \qquad W = \begin{pmatrix} 1 & 0 \\ 0 & \sqrt{i} \end{pmatrix},$$

$$N_c = \left(\begin{array}{c|c} I & 0 \\ \hline 0 & N \end{array}\right), \qquad H_c = \left(\begin{array}{c|c} I & 0 \\ \hline 0 & H \end{array}\right), \qquad V_c = \left(\begin{array}{c|c} I & 0 \\ \hline 0 & V \end{array}\right), \qquad W_c = \left(\begin{array}{c|c} I & 0 \\ \hline 0 & W \end{array}\right).$$

The unary gate N is called the *not*-gate, because it induces the following mapping of basis vectors: $\mathbf{0} \mapsto \mathbf{1}$ and $\mathbf{1} \mapsto \mathbf{0}$. The binary gate N_c is called the *controlled not*-gate. It also corresponds to a permutation of basis vectors: $\mathbf{00} \mapsto \mathbf{00}$, $\mathbf{01} \mapsto \mathbf{01}$, $\mathbf{10} \mapsto \mathbf{11}$, and $\mathbf{11} \mapsto \mathbf{10}$. Its action can be described as follows: if the first bit is $\mathbf{0}$, do nothing, else apply the N-gate to the second bit. In this sense, the first bit controls the action of the second one. More generally, for each unary gate S, there is a corresponding binary controlled gate S_c.

Unitary transformations can be used to create quantum superpositions, and they can also be used to create entanglement between quantum bits. For example, the so-called *Hadamard* gate H, when applied to the classical state $\mathbf{0}$, creates a quantum superposition $\frac{1}{\sqrt{2}}\mathbf{0} + \frac{1}{\sqrt{2}}\mathbf{1}$. Also, the controlled not-gate, applied to two independent quantum bits $(\frac{1}{\sqrt{2}}\mathbf{0} + \frac{1}{\sqrt{2}}\mathbf{1}) \otimes \mathbf{0}$, creates an entangled state $\frac{1}{\sqrt{2}}\mathbf{00} + \frac{1}{\sqrt{2}}\mathbf{11}$.

A quantum gate can be applied in the presence of additional quantum bits. For example, to apply a unary gate S to the 2nd quantum bit in a 4-bit system, we transform the system by the matrix $I \otimes S \otimes I \otimes I$. Here I is the 2×2-identity matrix.

Proposition 3.2 (see [3]). *The above set of standard gates is* complete*, i.e., any n-ary gate can be approximated, up to an arbitrarily small error ϵ, by a combination of the eight given gates.*

3.4 Measurement

Besides unitary transformations, there is another fundamental operation on the state of a quantum system, known as a *measurement*. This occurs when we try to "observe" the value of a quantum bit and convert it into a classical bit. Consider for instance the state of a single quantum bit $q = \alpha\mathbf{0} + \beta\mathbf{1}$. For simplicity, let us assume that the amplitudes have been normalized such that $|\alpha|^2 + |\beta|^2 = 1$. The act of measuring this quantum bit will yield an answer which is either $\mathbf{0}$ or $\mathbf{1}$. The answer $\mathbf{0}$ will occur with probability $|\alpha|^2$, and $\mathbf{1}$ with probability $|\beta|^2$. Moreover, the measurement causes the quantum state to *collapse*: after the measurement, the quantum state will have changed to either $\mathbf{0}$ or $\mathbf{1}$, depending on the result of the measurement. In particular, if we immediately measure the same quantum bit again, we always get the same answer as the first time.

The situation is more complex if more than one quantum bit is involved. Consider a 2-qbit system in the state $\alpha\mathbf{00} + \beta\mathbf{01} + \gamma\mathbf{10} + \delta\mathbf{11}$. We assume again that the amplitudes have been normalized. If we measure the value of the first bit, one of the following things will happen:

1. with probability $|\alpha|^2 + |\beta|^2$, the result of the measurement will be $\mathbf{0}$ and the quantum state will collapse to $\alpha\mathbf{00} + \beta\mathbf{01}$, and

2. with probability $|\gamma|^2 + |\delta|^2$, the result of the measurement will be $\mathbf{1}$ and the quantum state will collapse to $\gamma\mathbf{10} + \delta\mathbf{11}$.

Note that only the portion of the quantum state pertaining to the bit that we are observing collapses. If we were observing the second bit instead, the observed answer would be $\mathbf{0}$ with probability $|\alpha|^2 + |\gamma|^2$, and $\mathbf{1}$ with probability $|\beta|^2 + |\delta|^2$, and the quantum state would collapse, respectively, to $\alpha\mathbf{00} + \gamma\mathbf{10}$ or $\beta\mathbf{01} + \delta\mathbf{11}$.

Now let us see what happens if we measure the second quantum bit after the first one. The situation can be summarized in the following diagram:

$$
\begin{tikzcd}[column sep=tiny]
& & & \alpha\mathbf{00}+\beta\mathbf{01}+\gamma\mathbf{10}+\delta\mathbf{11} \arrow[dll, "p_0=|\alpha|^2+|\beta|^2"', "\mathbf{0}"] \arrow[drr, "\mathbf{1}"', "p_1=|\gamma|^2+|\delta|^2"] & & & \\
& \alpha\mathbf{00}+\beta\mathbf{01} \arrow[dl, "p_{00}=\frac{|\alpha|^2}{|\alpha|^2+|\beta|^2}"', "\mathbf{0}"] \arrow[dr, "\mathbf{1}"', "p_{01}=\frac{|\beta|^2}{|\alpha|^2+|\beta|^2}"] & & & & \gamma\mathbf{10}+\delta\mathbf{11} \arrow[dl, "p_{10}=\frac{|\gamma|^2}{|\gamma|^2+|\delta|^2}"', "\mathbf{0}"] \arrow[dr, "\mathbf{1}"', "p_{11}=\frac{|\delta|^2}{|\gamma|^2+|\delta|^2}"] & \\
\alpha\mathbf{00} & & \beta\mathbf{01} & & \gamma\mathbf{10} & & \delta\mathbf{11}
\end{tikzcd}
$$

In this tree, the nodes are labeled with states, and the transitions are labeled with probabilities. Note that the overall probability of observing $\mathbf{00}$ as the result of the two measurements is $p_0p_{00} = |\alpha|^2$, the probability of observing $\mathbf{01}$ is $p_0p_{01} = |\beta|^2$, and so forth. In particular, these probabilities are independent of the order of measurement; thus, the result does not depend on which quantum bit we measure first (or indeed, whether we measure both of them simultaneously).

We mentioned that it is customary to normalize quantum states so that the sum of the squares of the amplitudes is 1. However, in light of the above diagram, we find that it is often more convenient to normalize states differently.

Convention 3.3. We normalize each state in such a way that the sum of the squares of the amplitudes is equal to the total *probability that this state is reached.*

With this convention, it becomes unnecessary to renormalize the state after a measurement has been performed. We will see later that this convention greatly simplifies our computations.

3.5 Pure and mixed states

We know that the state of a quantum system at any given time can be completely described by its state vector $u \in \mathbb{C}^{2^n}$, modulo a scalar multiple. However, an outside observer might have incomplete knowledge about the state of the system. For instance, suppose that we know that a given system is either in state u or in state v, with equal probability. We use the ad hoc notation $\frac{1}{2}\{u\} + \frac{1}{2}\{v\}$ to describe this situation. In general, we write $\lambda_1\{u_1\} + \ldots + \lambda_m\{u_m\}$ for a system which, from the viewpoint of some observer, is known to be in state u_i with probability λ_i, where $\sum_i \lambda_i = 1$. Such a probability distribution on quantum states is called a *mixed state*. The underlying quantum states, such as u, are sometimes called *pure states* to distinguish them from mixed states.

It is important to realize that a mixed state is a description of an observers *knowledge* of the state of a quantum system, rather than a property of the system itself. In particular, a given system might be in two different mixed states from the viewpoints of two different observers. Thus, the notion of mixed states does not have an independent physical meaning. Physically, any quantum system is in a (possibly unknown) pure state at any given time.

Unitary transformations operate componentwise on mixed states. Thus, the unitary transformation S maps the mixed state $\lambda_1\{u_1\} + \ldots + \lambda_m\{u_m\}$ to $\lambda_1\{Su_1\} + \ldots + \lambda_m\{Su_m\}$.

The collapse of a quantum bit takes pure states to mixed states. For instance, if we measure a quantum bit in state $\alpha\mathbf{0} + \beta\mathbf{1}$, but ignore the outcome of the measurement, the system enters (from our point of view) the mixed state $|\alpha|^2\{\mathbf{0}\} + |\beta|^2\{\mathbf{1}\}$.

3.6 Density matrices

We now introduce a better notation for mixed quantum states, which is due to von Neumann. First, consider a pure state, represented as usual by a unit column vector u. In von Neumann's notation, this quantum state is represented by the matrix uu^*, called its *density matrix*. For example, the state of a quantum bit $u = \frac{1}{\sqrt{2}}\mathbf{0} - \frac{1}{\sqrt{2}}\mathbf{1}$ is represented by the density matrix

$$
uu^* = \begin{pmatrix} \frac{1}{2} & -\frac{1}{2} \\ -\frac{1}{2} & \frac{1}{2} \end{pmatrix}.
$$

Note that no information about a pure state is lost in this notation, because the vector u is uniquely determined, up to a scalar multiple, by the matrix uu^*. There are several advantages to the density matrix notation. A mundane advantage is that we do not have to write so many square roots. More importantly, if $u = \gamma v$, for some complex scalar γ with $|\gamma| = 1$, then $uu^* = \gamma\bar{\gamma}vv^* = vv^*$. Thus, the scalar factor disappears, and the normalized representation of each pure quantum state as a density matrix is unique. Also note that $\mathrm{tr}(uu^*) = |u|^2$. In particular, if u is a unit vector, then the density matrix has trace 1.

But the real strength of density matrices lies in the fact that they also provide a very economical notation for mixed states. A mixed state $\lambda_1 \{u_1\} + \ldots + \lambda_n \{u_n\}$ is simply represented as a linear combination of the density matrices of its pure components, i.e., as $\lambda_1 u_1u_1^* + \ldots + \lambda_n u_nu_n^*$. For example, the mixed state $\frac{1}{2}\{\mathbf{0}\} + \frac{1}{2}\{\mathbf{1}\}$ is expressed as the matrix

$$\frac{1}{2}\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} + \frac{1}{2}\begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{pmatrix}.$$

Remark 3.4. Some information about a mixed state is lost in the density matrix notation. For example, let $u = \frac{1}{\sqrt{2}}\mathbf{0} + \frac{1}{\sqrt{2}}\mathbf{1}$ and $v = \frac{1}{\sqrt{2}}\mathbf{0} - \frac{1}{\sqrt{2}}\mathbf{1}$. Then the mixed state $\frac{1}{2}\{u\} + \frac{1}{2}\{v\}$ gives rise to the following density matrix:

$$\frac{1}{2}\begin{pmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} \end{pmatrix} + \frac{1}{2}\begin{pmatrix} \frac{1}{2} & -\frac{1}{2} \\ -\frac{1}{2} & \frac{1}{2} \end{pmatrix} = \begin{pmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{pmatrix},$$

which is the same as the density matrix for the mixed state $\frac{1}{2}\{\mathbf{0}\} + \frac{1}{2}\{\mathbf{1}\}$ calculated above. But as we shall see in the next section, there is no observable difference between two mixed states that share the same density matrix, and thus there is no harm (and indeed, much benefit) in identifying such states.

We note that a matrix A is of the form $\lambda_1 u_1u_1^* + \ldots + \lambda_n u_nu_n^*$, for unit vectors u_i and non-negative coefficients λ_i with $\sum_i \lambda_i \leqslant 1$, if and only if A is positive hermitian and satisfies $\mathrm{tr}\, A \leqslant 1$. (The reason for allowing $\sum_i \lambda_i \leqslant 1$, rather than $\sum_i \lambda_i = 1$, will become apparent later). The left-to-right direction is trivial, and the right-to-left direction follows because any such A can be diagonalized as $A = SDS^*$, for some $D = \sum_i \lambda_i e_ie_i^*$. We then have $A = \sum_i \lambda_i u_iu_i^*$, where $u_i = Se_i$. This motivates the following definition:

Definition. A *density matrix* is a positive hermitian matrix A which satisfies $\mathrm{tr}\, A \leqslant 1$. We write $D_n \subseteq \mathbb{C}^{n\times n}$ for the set of density matrices of dimension n.

3.7 Quantum operations on density matrices

The two kinds of quantum operations, namely unitary transformation and measurement, can both be expressed with respect to density matrices. A unitary transformation S maps a pure quantum state u to Su. Thus, it maps a pure density matrix uu^* to Suu^*S^*. Moreover, a unitary transformation extends linearly to mixed states, and thus, it takes any mixed density matrix A to SAS^*.

Now consider the effect of a measurement on a density matrix. We begin by considering a pure state uu^*, for some unit vector u. Suppose that

$$u = \left(\begin{array}{c} v \\ \hline w \end{array}\right), \text{ therefore } uu^* = \left(\begin{array}{c|c} vv^* & vw^* \\ \hline wv^* & ww^* \end{array}\right).$$

Assuming that the rows of u are ordered according to Convention 3.1, then if we perform a measurement on the first bit, the outcome will be $\left(\begin{array}{c} v \\ \hline 0 \end{array}\right)$ with probability $|v|^2$, and $\left(\begin{array}{c} 0 \\ \hline w \end{array}\right)$ with probability $|w|^2$. Or in density matrix notation, the outcome will be

$$\left(\begin{array}{c|c} vv^* & 0 \\ \hline 0 & 0 \end{array}\right) \text{ or } \left(\begin{array}{c|c} 0 & 0 \\ \hline 0 & ww^* \end{array}\right),$$

where the first matrix occurs with probability $|v|^2$, and the second matrix occurs with probability $|w|^2$. Note that the probability that each matrix occurs is equal to its trace: $|v|^2 = \mathrm{tr}(vv^*)$ and $|w|^2 = \mathrm{tr}(ww^*)$. Thus, Convention 3.3 extends naturally to density matrices: the density matrix of a state is to be normalized in such a way that its trace corresponds to the overall probability that this state is reached. Note that with this convention, each of the two possible outcomes of a measurement is a linear function of the incoming state.

The measurement operation extends linearly from pure to mixed states. Thus, performing a measurement on a mixed state of the form

$$\left(\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right)$$

results in one of the two states

$$\left(\begin{array}{c|c} A & 0 \\ \hline 0 & 0 \end{array}\right) \text{ or } \left(\begin{array}{c|c} 0 & 0 \\ \hline 0 & D \end{array}\right),$$

where each of the two matrices occurs with probability equal to its trace. If one ignores the classical bit of information that is observed from the measurement, then the resulting state is a mixed state

$$\left(\begin{array}{c|c} A & 0 \\ \hline 0 & D \end{array}\right).$$

Thus, collapsing a quantum bit (measuring it while ignoring the result) corresponds to setting a certain region of the density matrix to 0.

We have seen that the effect of the two fundamental operations of quantum mechanics, unitary transformations and measurements, can be described in terms of their action on density matrices. Since unitary transformations and measurements are our only means of interacting with a quantum state, it follows that there is no observable difference between mixed states which have the same density matrix representation.

3.8 The complete partial order of density matrices

Recall that D_n is the set of density matrices of dimension n:

$$D_n = \{A \in \mathbb{C}^{n\times n} \mid A \text{ positive hermitian and } \operatorname{tr} A \leqslant 1\}.$$

Definition. For matrices $A, B \in \mathbb{C}^{n\times n}$, we define $A \sqsubseteq B$ if the matrix $B - A$ is positive.

It is immediately obvious that $\sqsubseteq$ defines a partial order on the set $\mathbb{C}^{n\times n}$. When restricted to the set D_n of density matrices, this partial order has the zero matrix 0 as its least element. We also denote the zero matrix by $\perp$ in this context.

Proposition 3.5. *The poset $(D_n, \sqsubseteq)$ is a complete partial order, i.e., it has least upper bounds of increasing sequences.*

Proof. Consider the standard Euclidean topology on $\mathbb{C}^{n\times n}$, which is the one that is induced by the norm $|A|$. By Remark 2.1, every $A \in D_n$ satisfies $|A| \leqslant \operatorname{tr} A \leqslant 1$. Thus D_n is a compact subset of $\mathbb{C}^{n\times n}$. Now suppose $A_0 \sqsubseteq A_1 \sqsubseteq \ldots$ is an increasing sequence in D_n. Let $a_i = \operatorname{tr} A_i$, and note that $a_0 \leqslant a_1 \leqslant \ldots \leqslant 1$. Because the sequence $(a_i)_{i\in\mathbb{N}}$ is real-valued, bounded, and increasing, it has a limit a in $\mathbb{R}$. Then for any $i \leqslant j$, the matrix $A_j - A_i$ is positive, and thus $|A_j - A_i| \leqslant \operatorname{tr}(A_j - A_i) = a_j - a_i$ by Remark 2.1. Thus, $(A_i)_{i\in\mathbb{N}}$ is a Cauchy sequence in D_n, and by compactness, it converges to some $A \in D_n$ in the standard topology.

It remains to be shown that A is the least upper bound of the sequence $(A_i)_{i\in\mathbb{N}}$. First note that for any $B \in D_n$, the sets $\downarrow B = \{A \in D_n \mid A \sqsubseteq B\}$ and $\uparrow B = \{A \in D_n \mid B \sqsubseteq A\}$ are closed in the standard topology. Now for every $i \in \mathbb{N}$, since $A_{i+1}, A_{i+2}, \ldots \in \uparrow A_i$, it follows that in the limit, $A \in \uparrow A_i$. Thus, A is an upper bound of the sequence. If B is another upper bound, then $A_i \in \downarrow B$ for all i, hence in the limit, $A \in \downarrow B$, thus A is the least upper bound. □

Remark 3.6. The proof of Proposition 3.5 shows that, for any increasing sequence in D_n, A is the least upper bound if and only if A is the topological limit of the sequence. It follows, among other things, that a monotone function $f : D_n \to D_m$ is Scott continuous (i.e., preserves least upper bounds of increasing sequences) if it is topologically continuous.

4 Quantum flow charts (QFC)

We now turn to the question of how to design a quantum programming language, and how to define its semantics. One of the first issues that we are faced with is the choice of a convenient syntax. Out of the many possibilities, we choose to represent programs as *flow charts*, also known as *control flow diagrams*. However, unlike traditional flow charts for imperative programming languages, our flow charts have a more "functional" flavor; in our setting, commands act by transforming specific inputs to outputs, rather than by updating global variables. Thus, we combine the language of control flow with some elements of data flow.

In this section and the next one, we describe the syntax of flow charts, and we introduce their semantics informally. The formal semantics is given in Section 6. Some alternative language choices are discussed in Section 7.

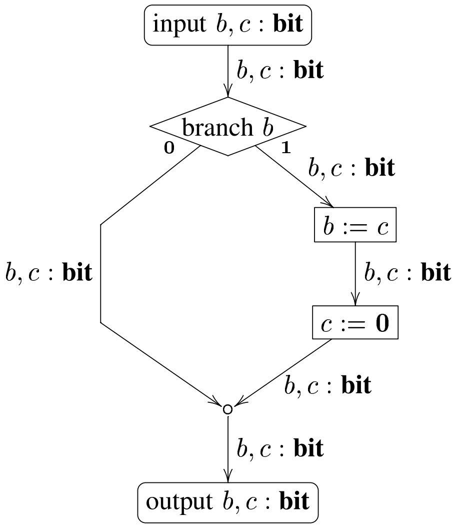


Figure 1: A simple classical flow chart

4.1 Classical flow charts

The concept of a flow chart in "functional style" is best illustrated by giving some examples. It is instructive to consider the classical (i.e., not quantum) case first. Consider the simple flow chart shown in Figure 1.

Unless otherwise indicated by arrows, the flow of control is from top to bottom. Each edge is labeled with a typing judgment, i.e., by a list of typed variables. These are the variables which are available at that given point in the program. For simplicity, we consider only a single data type for now: the type **bit** of a classical bit (also known as the type of booleans). In the example in Figure 1, no variables are created or disposed of, so each edge is labeled with the same typing judgment b, c : **bit**.

This program fragment inputs a pair of bits, performs a conditional branch and some updates, and then outputs the pair (b, c). The semantics of this program can be described as a map from its inputs to its outputs. Specifically, the map computed by this program is:

$$\begin{array}{ccc} \mathbf{00} & \mapsto & \mathbf{00} \\ \mathbf{01} & \mapsto & \mathbf{01} \\ \mathbf{10} & \mapsto & \mathbf{00} \\ \mathbf{11} & \mapsto & \mathbf{10} \end{array}$$

The *state* of the program, between instructions, is given by a pair (e, ρ), where e is an edge of the flow chart (thought of as the *program counter*), and ρ is an assignment of values to the variables with which e is labeled.

An important observation is that the two components of the state, instruction pointer and value assignment, are fundamentally of the same nature. Thus, the instruction pointer could be thought of a variable (and indeed, in most hardware implementations, it is represented by a machine register). Conversely, the content of a boolean variable can be thought of as encoding a choice between two alternative control paths. For example, an edge labeled with a boolean variable b can be equivalently replaced by two parallel (unlabeled) edges, corresponding to the states $b = 0$ and $b = 1$, respectively. Similarly, an edge labeled with two boolean variables can be replaced by four parallel edges, corresponding to the four possible states $\mathbf{00}$, $\mathbf{01}$, $\mathbf{10}$, and $\mathbf{11}$, and so on. In this way, each classical flow chart can be transformed into an equivalent (though much larger) flow chart that uses no variables. After such a transformation, each conditional branch has a predetermined outcome, and each assignment corresponds to a jump to the appropriate parallel component. To illustrate this point, the expansion of the flow chart from Figure 1 is explicitly shown in Figure 2. Here, the four entrance points of the expanded program correspond to the four possible pairs of boolean inputs of the original program, and the four exit points correspond to the four potential pairs of boolean outputs.

It is this connection between control and classical state, namely, the fact that a control edge labeled with a tuple of classical variables can be replaced by a number of parallel control edges, which we mean when we say "control is classical". We will see later that a similar observation does not apply to quantum data; quantum data is of a fundamentally different nature.

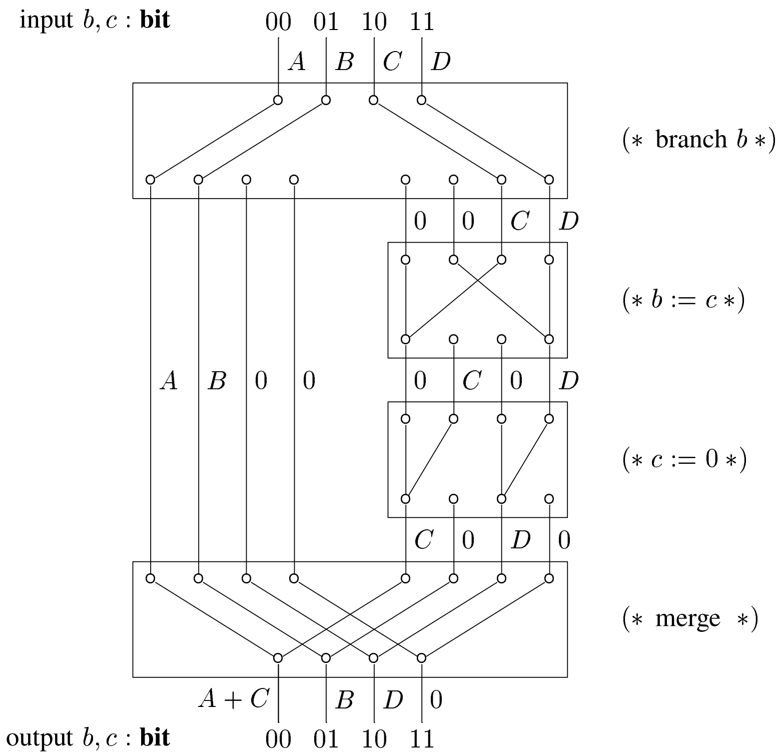


Figure 2: Classical flow chart from Figure 1, with boolean variables expanded

4.2 Probabilistic view of classical flow charts

Consider again the flow chart from Figure 2. We will describe an alternative view of its semantics. This time, imagine that one of the four possible entrance points **00**, **01**, **10**, or **11** is selected randomly, with respective probabilities A, B, C, and D. Then we can annotate, in a top-down fashion, each edge of the flow chart with the probability that this edge will be reached. In particular, any edge that is unreachable will be annotated with "0". The resulting annotation is shown in Figure 2. We find that the probabilities of the final outcomes **00**, **01**, **10**, and **11** are $A + C$, B, D, and 0, respectively. In this way, each program gives rise to a function from tuples of input probabilities to tuples of output probabilities. In our example, this function is $F(A, B, C, D) = (A + C, B, D, 0)$.

Note that our original reading of a flow chart, as a function from inputs to outputs, is completely subsumed by this probabilistic view. For instance, the fact that the input **11** is mapped to **10** is easily recovered from the fact that $F(0, 0, 0, 1) = (0, 0, 1, 0)$.

In practice, it is usually preferable to think of a small number of variables rather than of a large number of control paths. Therefore, we will of course continue to draw flow charts in the style of Figure 1, and not in that of Figure 2. However, the preceding discussion of probabilities still applies, with one modification: each edge that is labeled with n boolean variables should be annotated by a tuple of 2^n probabilities, and not just a single probability.

4.3 Summary of classical flow chart components

The basic operations for boolean flow charts are summarized in Figure 3. Here, Γ denotes an arbitrary typing context, and A and B denote tuples of probabilities. If A and B are tuples of equal length, we use the notation (A, B) to denote the concatenation of A and B.

We distinguish between the *label* of an edge and its *annotation*. A *label* is a typing context, and it is part of the syntax of our flow chart language. An *annotation* is a tuple of probabilities, and it is part of the semantics. We use the equality symbol "=" to separate labels from annotations. Thus, Figure 3 defines both the syntax and the semantics of the language. Note that the statement of the rules makes use of the indexing convention of Section 3.2, because the order of the probabilities in each tuple is determined by the lexicographical ordering of the corresponding states.

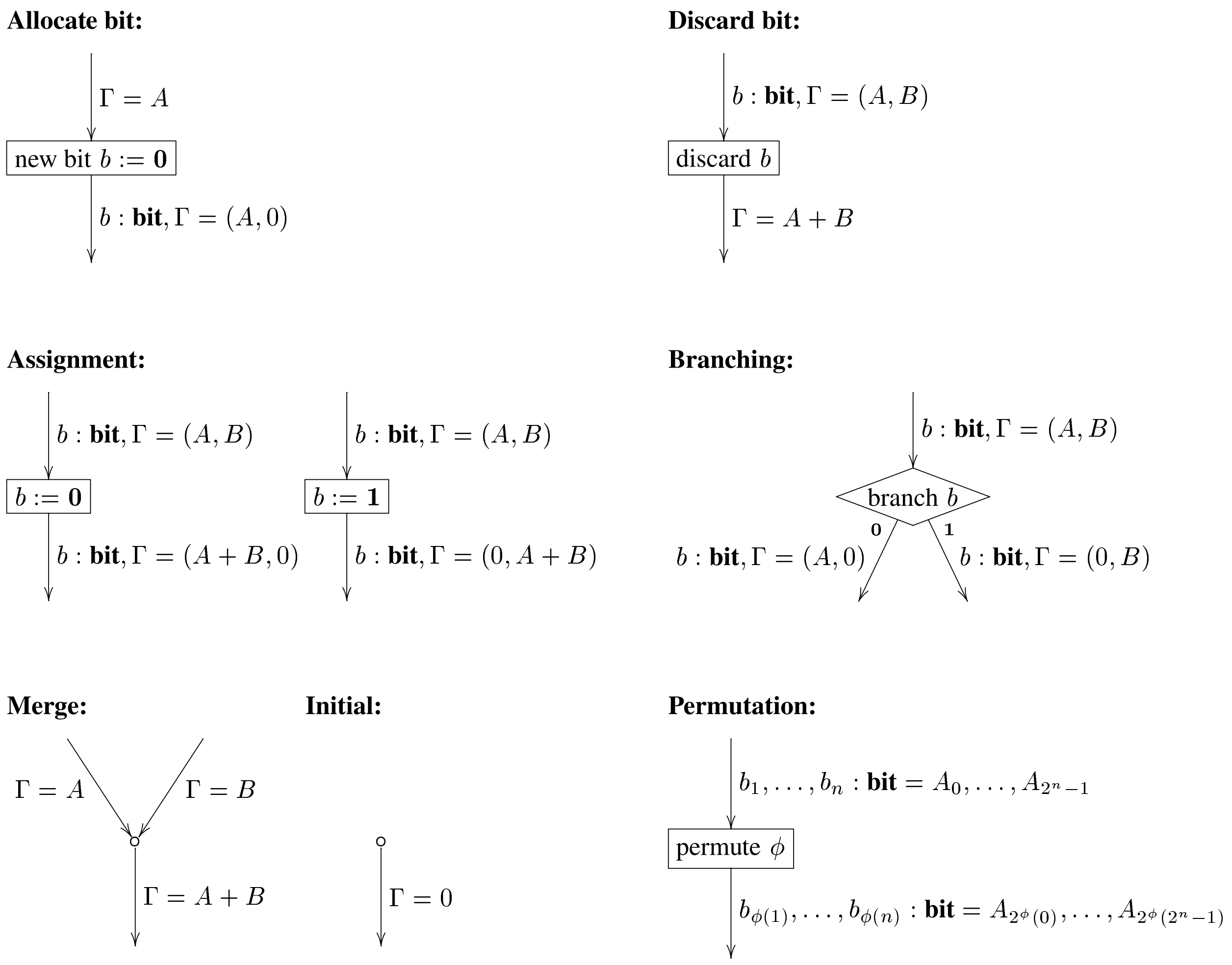


Figure 3: Rules for classical flow charts

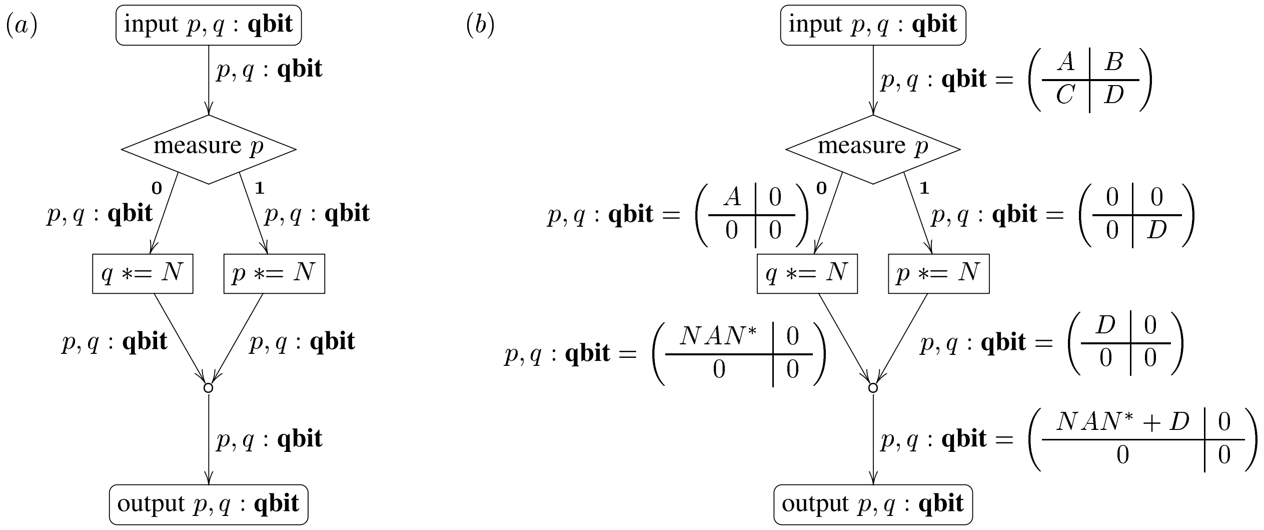
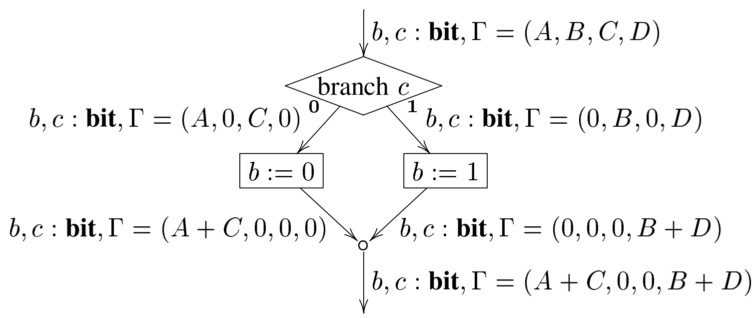


Figure 4: A simple quantum flow chart

There are four rules in Figure 3 that have not been discussed so far: “new” allocates a new variable and initializes it to **0**, “discard” deallocates a variable, “initial” creates an unreachable control path (this was used e.g. in Figure 2), and “permute” is a dummy operation which allows us to rearrange the variables in the current typing context. Note that, although we have not imposed a block structure on the uses of “new” and “discard”, the typing rules nevertheless ensure that every allocated variable is eventually deallocated, unless it appears in the output. Also note that the “initial” node is essentially a 0-ary version of “merge”.

Naturally, we will allow various kinds of syntactic sugar in writing flow charts that are not directly covered by the rules in Figure 3. For instance, an assignment of the form $b := c$ can be regarded as an abbreviation for the following conditional assignment:



4.4 Quantum flow charts

Quantum flow charts are similar to classical flow charts, except that we add a new type **qbit** of quantum bits, and two new operations: unitary transformations and measurements. A unitary transformation operates on one or more quantum bits; we write $q \mathrel{*}= S$ for the operation of applying a unary quantum gate S to the quantum bit q. Note that this operation updates q; the notation is analogous to the notation $b := \mathbf{1}$ for classical assignment. For the application of a binary quantum gate S to a pair p, q of quantum bits, we use the notation $p, q \mathrel{*}= S$, and so forth for gates of higher arity. Sometimes we also use special notations such as $q \oplus= 1$ for $q \mathrel{*}= N$, and $q \oplus= p$ for $p, q \mathrel{*}= N_c$, where N and N_c are the not- and controlled not-gate, respectively. The second new operation, the measurement, is a branching statement since it can have two possible outcomes.

As a first example, consider the simple flow chart shown in Figure 4(a). This program fragment inputs two quantum bits p and q, measures p, and then performs one of two possible unitary transformations depending on the outcome of the measurement. The output is the modified pair p, q.

The behavior of a quantum flow chart can be described as a function from inputs to outputs. For instance, in the flow chart of Figure 4(a), the input $\mathbf{00}$ leads to the output $\mathbf{01}$, and the input $\frac{1}{\sqrt{2}}\mathbf{00} + \frac{1}{\sqrt{2}}\mathbf{01}$ leads to the output $\frac{1}{\sqrt{2}}\mathbf{00} + \frac{1}{\sqrt{2}}\mathbf{01}$. However, due to the probabilistic nature of measurement, the output is not always a pure state: for example, the input $\frac{1}{\sqrt{2}}\mathbf{00} + \frac{1}{\sqrt{2}}\mathbf{10}$ will lead to the outputs $\mathbf{00}$ or $\mathbf{01}$ with equal probability. We can represent this outcome by the mixed state $\frac{1}{2}\{\mathbf{00}\} + \frac{1}{2}\{\mathbf{01}\}$.

As the example shows, the output of a quantum flow chart is in general a mixed state. We may take the input to be a mixed state as well. Thus, the semantics of a quantum flow chart is given as a function from mixed states to mixed states. To calculate this function, let us use density matrix notation, and let us assume that the input to the program is some mixed state

$$M = \left(\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right),$$

where each of A, B, C, D is a 2×2-matrix. Recall that the indexing convention of Section 3.2 prescribes that the rows and columns of M are indexed by the basic states $\mathbf{00}$, $\mathbf{01}$, $\mathbf{10}$, and $\mathbf{11}$ in this respective order. We can now decorate the flow chart in top-down fashion, by annotating each edge with the mixed state that the program is in when it reaches that edge. The resulting annotation is shown in Figure 4(b). The semantics of the entire program fragment is thus given by the following function of density matrices:

$$F\left(\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right) = \left(\begin{array}{c|c} NAN^* + D & 0 \\ \hline 0 & 0 \end{array}\right).$$

Note that we have followed our usual convention of normalizing all density matrices so that their trace equals the probability that the corresponding edge is reached. This convention has several nice properties: First and foremost, it ensures that the annotation of each edge is a linear function of the input. In particular, the merge operation (the joining of two control edges) amounts to a simple matrix addition. The normalization convention also implies that the traces of the matrices along any horizontal section of the flow chart add up to 1 (assuming that the trace of the input matrix is 1).

Another interesting observation about the program in Figure 4 is that if the input is a pure state, then the states along each of the two branches of the measurement continue to be pure. Unitary transformations also preserve pure states. It is finally the merge operation, denoted by a small circle "$\circ$", which combines two pure states into an impure state. Thus, the source of impure states in a quantum system is not the measurement operation (as one might have thought), but rather the merge operation, i.e., the erasure of classical information.

4.5 Summary of quantum flow chart operations

The operations for basic quantum flow charts are summarized in Figure 5. As before, we distinguish between the *label* of an edge and its *annotation*. The label is a typing context Γ, and it is part of the syntax of the language. The annotation is a density matrix, and it is part of the semantics. Quantum bits can be allocated and discarded; here it is understood that allocating a quantum bit means to request an unused quantum bit from the operating system. Such newly allocated quantum bits are assumed to be initialized to $\mathbf{0}$. Unitary transformations and measurement were discussed in the previous section, and "merge", "initial", and "permute" are as for classical flow charts. Note that in the rule for unitary transformations, $\bar{q}$ stands for a list of *distinct* variables. The type system ensures that syntactically distinct variables will refer to distinct objects at runtime. In particular, note that it is never possible to "copy" a quantum bit.

4.6 Detour on implementation issues

We briefly interrupt our description of quantum flow charts to contemplate some possible implementation issues. It is understood that no actual quantum hardware currently exists; thus any discussion of implementations is necessarily speculative. But as outlined in the introduction, it is useful to keep in mind a hypothetical hardware device on which the language can be implemented. Following [8], we imagine that practical quantum computing will take place on a

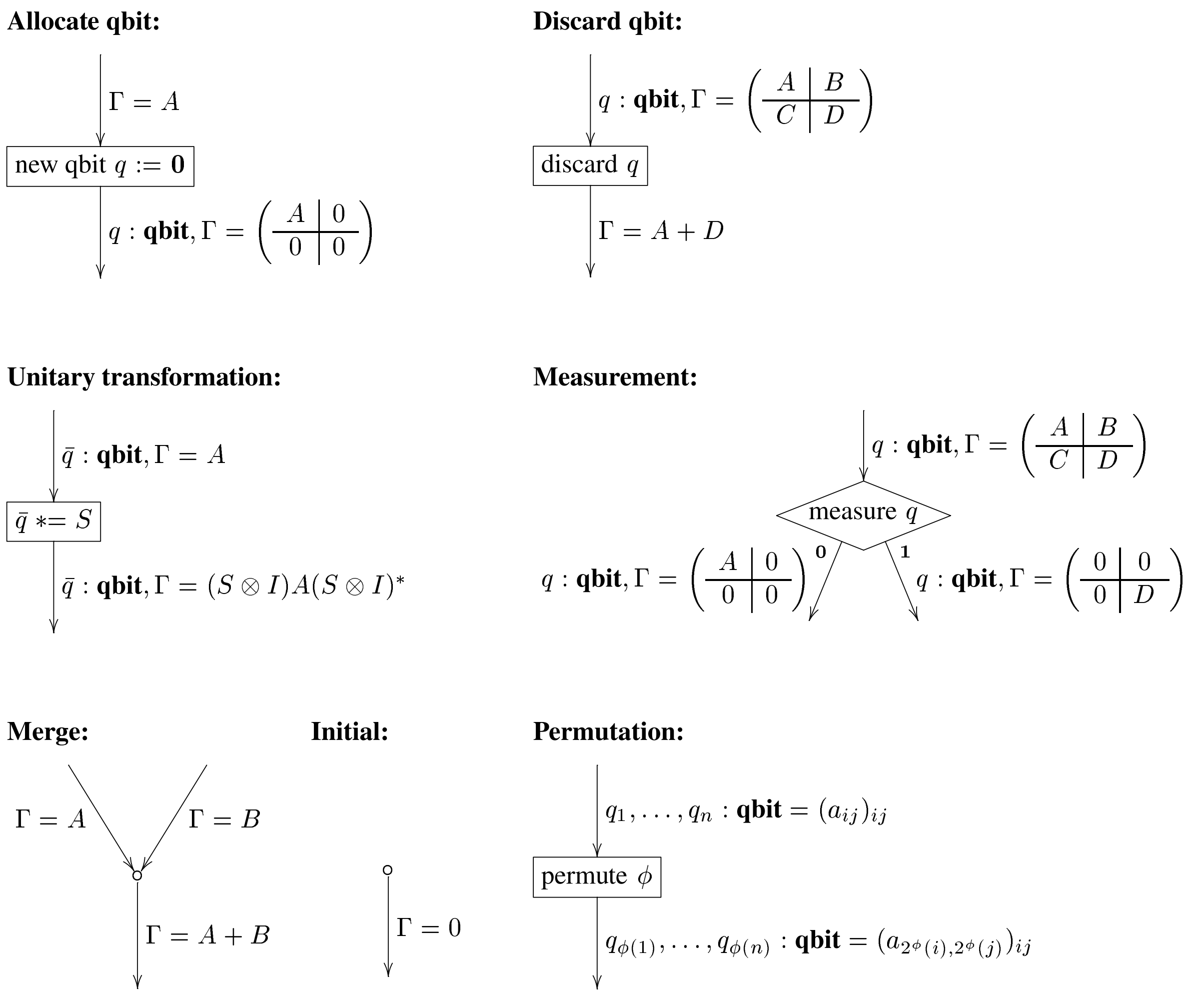


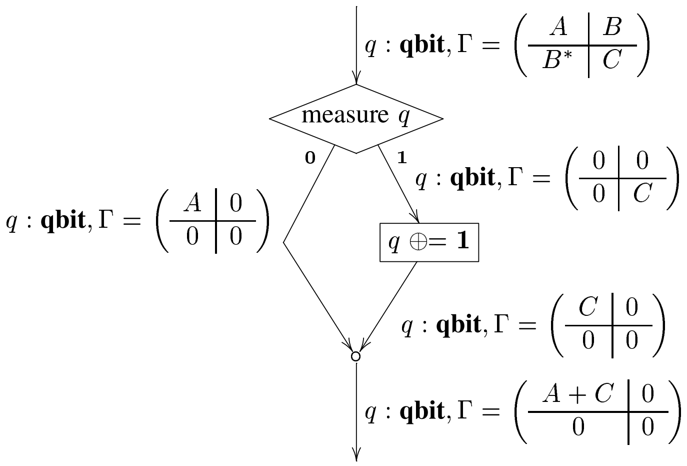
Figure 5: Rules for quantum flow charts

QRAM machine, which consists of a general-purpose classical computer controlling a special quantum hardware device which provides a bank of individually addressable quantum bits. The classical computer determines the sequence of elementary quantum operations (built-in unitary gates and measurements) to be performed by the quantum device.

To make the quantum device available to user programs, we further imagine that the operating system provides a number of services. One of these services is access control. The operating system keeps a list of quantum bits that are currently in use by each process. When a process requests a new quantum bit, the operating system finds a quantum bit that is not currently in use, marks that bit as being in use by the process, initializes the contents to $\mathbf{0}$, and returns the address of the newly allocated bit. The process can then manipulate the quantum bit, for instance via operating system calls which take the bit's address as a parameter. The operating system ensures that processes cannot access quantum bits that are not currently allocated to them – this is very similar to classical memory management. Finally, when a process is finished using a certain quantum bit, it may deallocate it via another operating system call; the operating system will then reset the bit to $\mathbf{0}$, and mark it as unused.

In practice, there are many ways of making this scheme more efficient, for instance by dividing the available quantum bits into regions, and allocating and deallocating them in blocks, rather than individually. However, for the purpose of this theoretical discussion, we are not too concerned with such implementation details. What is important is the interface presented to user programs by the operating system. In particular, the above discussion is intended to clarify the operations of "allocating" and "discarding" quantum bits; clearly, these concepts do not refer to physical acts of creation and destruction of quantum bits, but rather to access control functions performed by the operating system.

We have mentioned several instances in which the operating system resets or initializes a quantum bit to $\mathbf{0}$. This is indeed possible, and can be implemented by first measuring the quantum bit, and then performing a conditional "not" operation dependent on the outcome of the measurement. The following program fragment illustrates how an arbitrary quantum bit q can be reset to $\mathbf{0}$:



4.7 Combining classical data with quantum data

We observed in Section 4.1 that classical data can be equivalently viewed in terms of control paths. Since our quantum flow charts from Section 4.4 already combine quantum data with control paths, there is nothing particularly surprising in the way we are going to combine classical data with quantum data. The combined language has two data types, **bit** and **qbit**. Typing contexts are defined as before. For the semantics, observe that an edge which is labeled with n bits and m qbits can be equivalently replaced by 2^n edges which are labeled with m qbits only. Thus, a *state* for a typing context Γ containing n bits and m qbits is given by a 2^n-tuple $(A_0, \ldots, A_{2^n-1})$ of density matrices, each of dimension $2^m \times 2^m$. We extend the notions of trace, adjoints, matrix multiplication, and norm to tuples of matrices as follows:

$$\begin{array}{lcl}
\mathrm{tr}(A_0, \ldots, A_{2^n-1}) & := & \sum_i \mathrm{tr}\, A_i, \\
(A_0, \ldots, A_{2^n-1})^* & := & (A_0^*, \ldots, A_{2^n-1}^*), \\
S(A_0, \ldots, A_{2^n-1})S^* & := & (SA_0S^*, \ldots, SA_{2^n-1}S^*), \\
|(A_0, \ldots, A_{2^n-1})|^2 & := & \sum_i |A_i|^2.
\end{array}$$

We often denote tuples by letters such as A, B, C, and as before, we use the notation (A, B) for concatenation of tuples, if A and B have the same number of components. If A, B, C, D are tuples of matrices of identical dimensions, then we write

$$\left(\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right)$$

to denote the tuple whose ith component is $\left(\begin{array}{c|c} A_i & B_i \\ \hline C_i & D_i \end{array}\right)$, where A_i is the ith component of A etc. In this way, we use componentwise notation both along the "tuple dimension" and along the "matrix dimension".

Flow charts are acyclic graphs whose edges are labeled with typing contexts, and whose nodes are of the types shown in Figures 3 and 5. A flow chart may have any number of global incoming (input) and outgoing (output) edges.

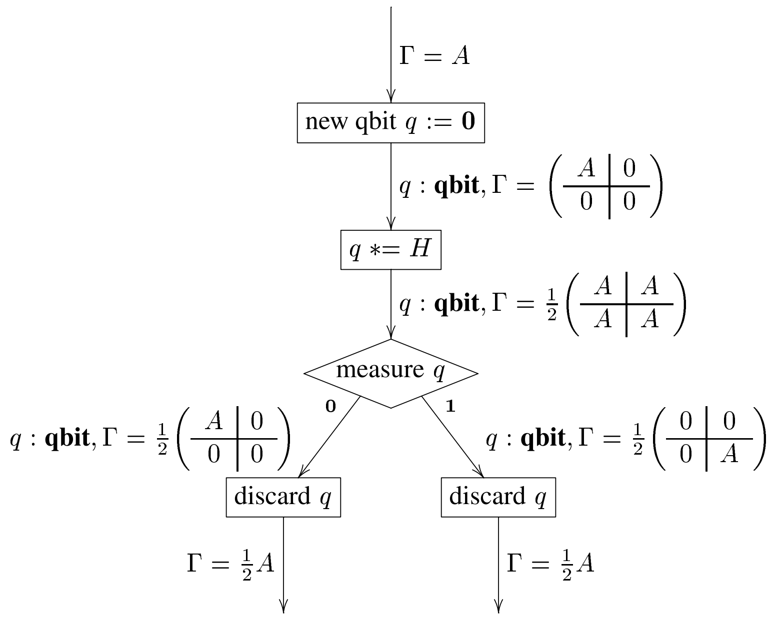
An *annotation* of a flow chart is an assignment of a matrix tuple to each edge, consistent with the rules of Figures 3 and 5. Here it is now understood that A, B, C, D denote matrix tuples of the correct dimensions determined by the corresponding typing context. For reasons that will become clear in Section 5.3, we allow flow charts to be annotated with arbitrary matrices, and not just density matrices.

The annotation of a flow chart is uniquely determined by the annotation of its input edges, and can be calculated in a top-down fashion. The semantics of a flow chart is given by the function which maps each annotation of the input edges to the resulting annotation of the output edges. By inspecting the rules in Figures 3 and 5, we observe that this function is necessarily linear. Moreover, it takes adjoints to adjoints, and thus it preserves hermitian matrix tuples. Moreover, this function preserves positivity, and it preserves trace, in the sense that the sum of the traces of the inputs is equal to the sum of the traces of the outputs. Intuitively, the last property reflects the fact that the probability of entering a program fragment is equal to the probability of leaving it. When we introduce loops into the flow chart language in Section 5, we will see that the trace preservation property no longer holds: in the presence of loops, some programs have a non-zero probability of non-termination. For such programs, the trace of the outputs will in general be less than the trace of the inputs.

4.8 Examples

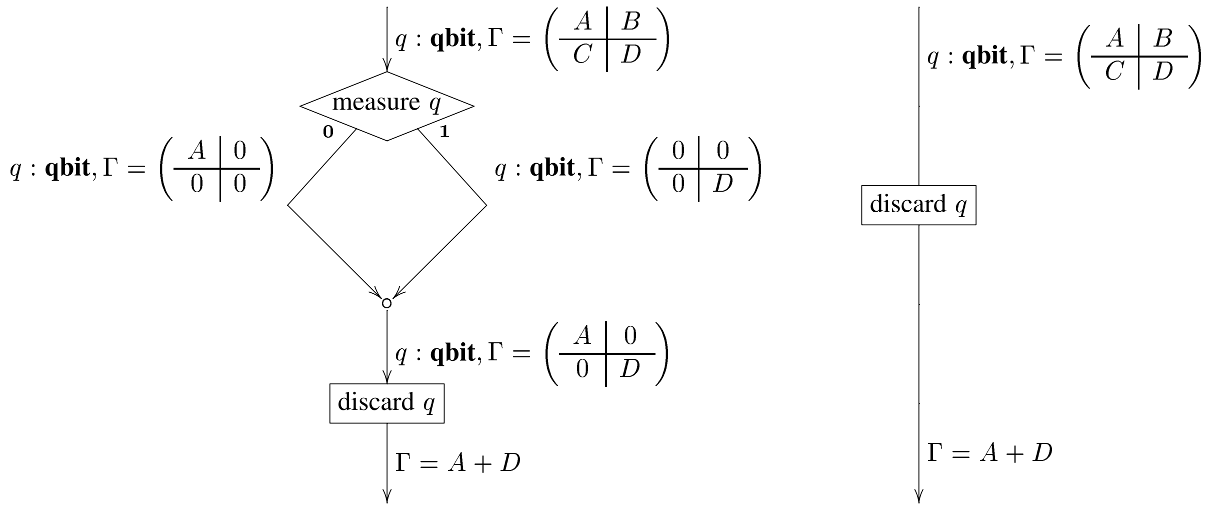
We give some examples of flow charts that can be built from the basic components.

Example 4.1. The first example shows how a probabilistic fair coin toss can be implemented, i.e., a branching statement which selects one of two possible outcomes with equal probability.



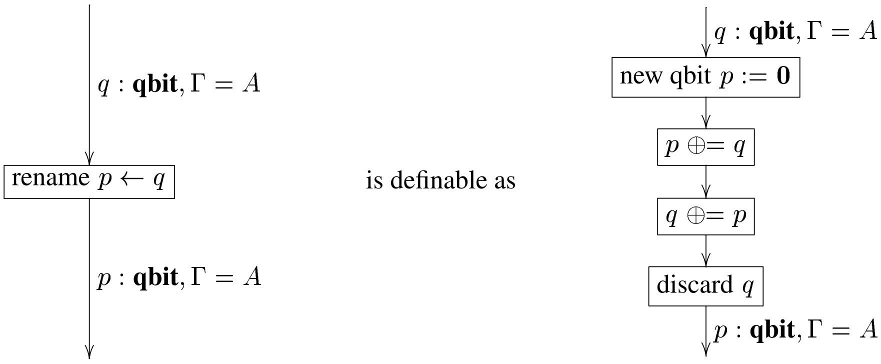
Here, H is the Hadamard matrix introduced in Section 3.3. Coin tosses with probabilities other than $\frac{1}{2}$ can be implemented by replacing H with some other appropriate unitary matrix.

Example 4.2. The next example shows the correctness of a program transformation: a measurement followed by deallocation is equivalent to a simple deallocation.



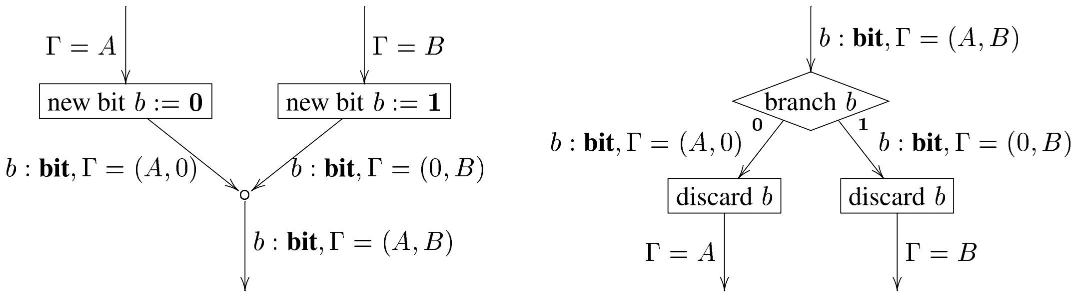
The correctness of this program transformation is of course due to the fact that the "discard" operation already has an implicit measurement built into it.

Example 4.3. This example shows how to define a "rename" operation for renaming a variable of type **qbit**. The given implementation is not very efficient, because it involves the application of a quantum gate. In practice, a compiler might be able to implement such renamings by a pointer operation with minimal runtime cost.

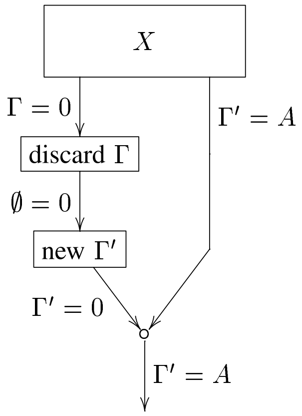


Of course, variables of type **bit** can be similarly renamed. We will from now on use the rename operation as if it were part of the language.

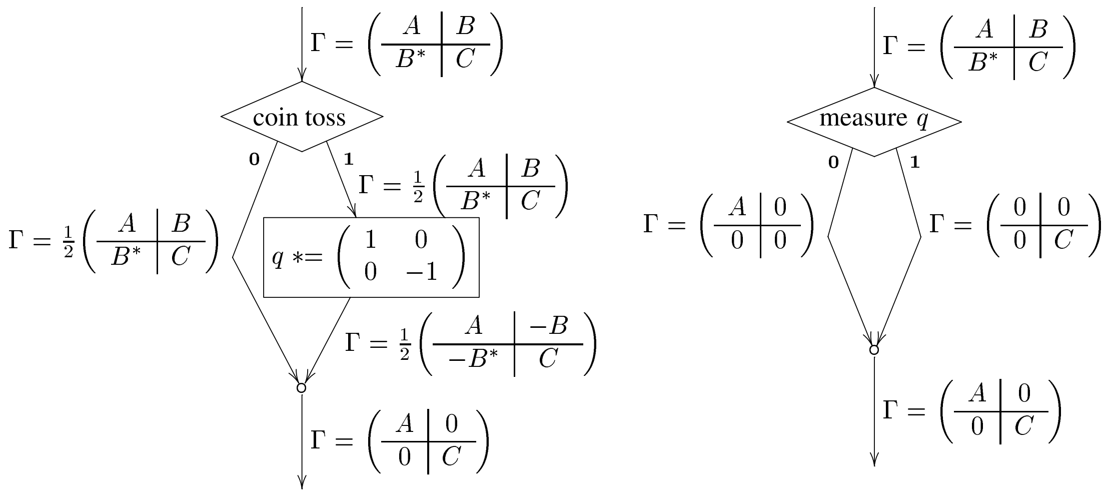
Example 4.4. This example formalizes a point that was made in Section 4.1: a control edge labeled with a classical bit is equivalent to two parallel control edges. In other words, the following two constructions are mutually inverse:



Example 4.5. This example shows that if a program fragment X has an outgoing edge which is reached with probability 0, then this edge can be eliminated. Here we have used an obvious abbreviation for multiple "discard" and "new" nodes.



Example 4.6. The next example shows something more interesting: it is possible to collapse a quantum bit q by means of a coin toss, without actually measuring q. The coin toss can be implemented as in Example 4.1. Let $\Gamma = q : \mathbf{qbit}, \Gamma'$.

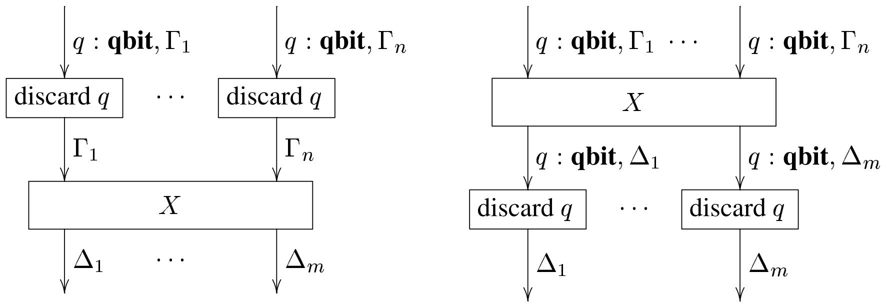


This example shows that it is possible for two programs to have the same observable behavior, despite the fact that their physical behavior is obviously different. In particular, the correctness of the left program depends critically on the fact that the outcome of the coin toss is "forgotten" when the two control paths are merged. Note that, if the outcome of the coin toss is known to some outside observer (e.g., to someone who has been eavesdropping), then it is possible, for this outside observer, to restore the initial state of the left program from its final state, simply by undoing the conditional unitary operation. On the other hand, the initial state of the right program is irretrievably lost after the measurement.

This apparent paradox is due to the fact, as discussed in Section 3.5, that a mixed state is a description of our *knowledge* of a physical state, rather than of a physical state itself. The two program fragments are equivalent in the sense that they will behave in the same way as part of any larger program, not in the sense that they cannot be distinguished by an outside observer with privileged knowledge.

It is precisely for this reason that the theory of *quantum communication*, i.e., of quantum programs which interactive input and output, is much more complicated than the theory of closed-world programs considered in this paper.

Example 4.7. This example shows that the discarding of a quantum bit can always be postponed. Thus, the following two flow charts are equivalent, provided that X is a flow chart not containing q;

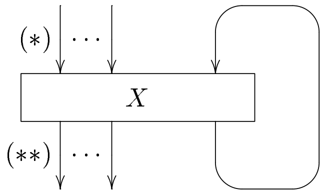


This can be easily shown by induction on flow charts. Note that this observation implies that there is effectively no difference between "discarding" a quantum bit and simply "forgetting" the quantum bit (by dropping any reference to it). In particular, it is not observable whether a "forgotten" quantum bit has been collapsed or not. This is true even if the quantum bit was entangled with other data in the computation — but only as long as no information about the collapsed quantum bit is leaked back to the program, not even via a third party such as the operating system. Because of the possibility of such unintended leaks, the discard operation should in practice always be implemented via an explicit collapse of the kind discussed at the end of Section 4.6.

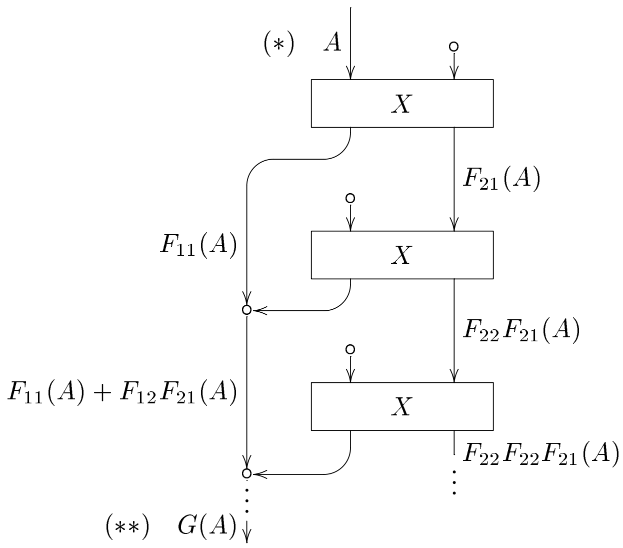
5 Loops, procedures, and recursion

5.1 Loops

A loop in a flow chart is constructed as in the following illustration:



Here X stands for an arbitrary flow chart fragment with $n+1$ incoming and $m+1$ outgoing edges. After adding the loop, there are n incoming and m outgoing edges left. The semantics of loops is initially defined by "infinite unwinding". The above loop is unwound as follows:



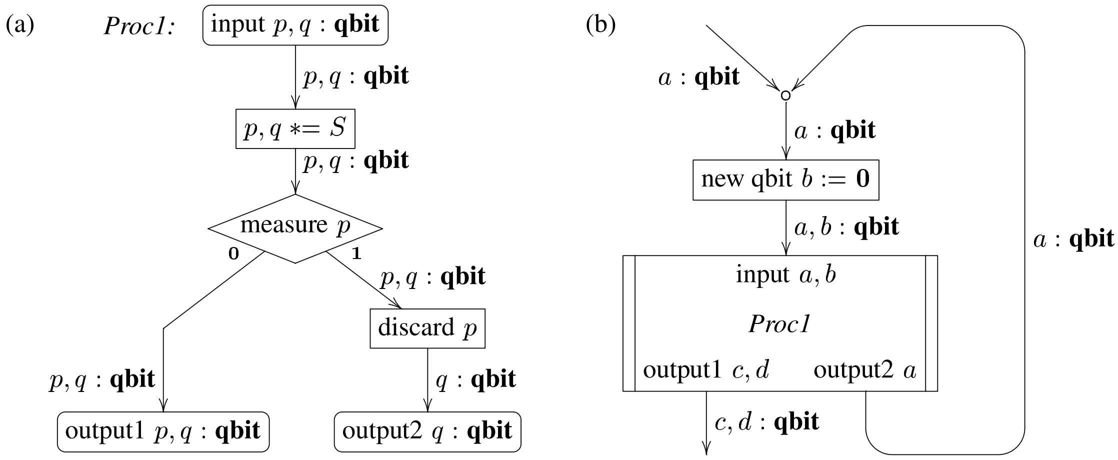


Figure 6: A procedure and a procedure call

Here, we have simplified the typesetting by representing potential multiple parallel control edges by a single line. Thus, $A = (A_1, \ldots, A_n)$ denotes a tuple of input matrices. We can decorate the unwound diagram with states in the usual top-down fashion. Specifically, suppose that the semantics of X is given by the linear function $F(A_1, \ldots, A_n, B) = (C_1, \ldots, C_m, D)$. We can split this function into four components F_{11}, F_{12}, F_{21}, and F_{22} such that $F(A, 0) = (F_{11}(A), F_{21}(A))$ and $F(0, B) = (F_{12}(B), F_{22}(B))$. Then we can label the states of the unwound loop diagram as shown in the illustration above. We find that the state at the edge (or tuple of edges) labeled $(**)$ is given by the infinite sum

$$G(A) = F_{11}(A) + \sum_{i=0}^{\infty} F_{12}(F_{22}^{i}(F_{21}(A))). \tag{1}$$

We will see in Section 6 that this sum indeed always converges. Furthermore, if A is positive, then so is $G(A)$, and $\mathrm{tr}\, G(A) \leqslant \mathrm{tr}\, A$.

An interesting point is that the inequality of traces may be strict, i.e., it is possible that $\mathrm{tr}\, G(A) < \mathrm{tr}\, A$. This can happen if there is a non-zero probability that the loop may not terminate. In this case, the probability that the program reaches state $(**)$ is strictly less than the probability that it reaches state $(*)$.

Note that the formula (1) allows us to calculate the semantics of the loop directly from the semantics of X, without the need to unwind the loop explicitly. This is an example of a compositional semantics, which we will explore in more detail in Section 6.

5.2 Procedures

A *procedure* is a flow chart fragment with a name and a type. Consider for example the procedure *Proc1* defined in Figure 6(a). This procedure has one entrance and two possible exits. The input to the procedure is a pair of qbits. The output is a pair of qbits when exiting through the first exit, or a single qbit when exiting through the second exit. The type of the procedure captures this information, and it is

$$\textit{Proc1} : \mathbf{qbit} \times \mathbf{qbit} \to \mathbf{qbit} \times \mathbf{qbit};\ \mathbf{qbit}.$$

Here, it is understood that "$\times$" binds more tightly than ";". In general, the type of a procedure is of the form $\bar{\Gamma} \to \bar{\Gamma}'$, where $\bar{\Gamma}, \bar{\Gamma}'$ are lists of products of basic types. Most procedures have a single entrance and a single exit, but there is no general reason why this should be so; we allow procedures with multiple entrances as well as multiple exits.

Figure 6(b) shows an example of a call to the procedure *Proc1* just defined. The example illustrates several points. The procedure call respects the type of the procedure, in the sense that it has as many incoming and outgoing edges as the procedure, and the number and type of parameters matches that of the procedure. The actual parameters are named inside the procedure call box. The order of the parameters is significant, and they are subject to one important restriction: the parameters corresponding to any one procedure entrance or exit must be *distinct*. Thus, we cannot for instance invoke *Proc1* with parameters (a, a).

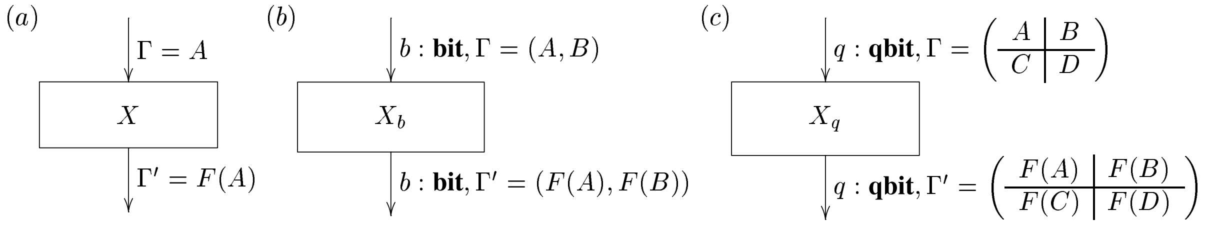
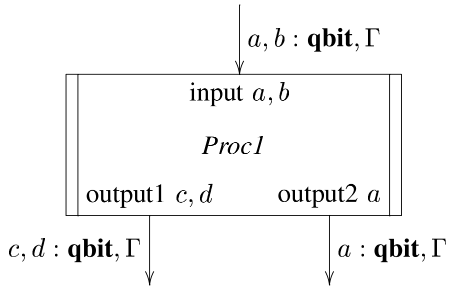


Figure 7: Weakening

We do not require that the names of the actual parameters match those of the formal parameters. For instance, in Figure 6, the actual parameters a, b correspond to the formal parameters p, q in the input of the procedure. We do not even require that the actual parameters must match the formal parameters consistently: for instance, the formal parameter q corresponds to the actual parameter b in the input, but to a in the second output. This is not important, as the compiler can implicitly insert renaming operations as in Example 4.3.

In general, a procedure may be called in a context which contains other variables besides those that are parameters to the procedure call. For instance, the procedure of Figure 6(a) can be invoked in the presence of an additional typing context Γ as follows:



Here, the set Γ of unused variables must be identical for all inputs and outputs of the procedure call. Intuitively, the variables in Γ are "not changed" by the procedure call; however, in reality, the behavior is more subtle because some of the variables from Γ might be quantum entangled with the procedure parameters, and thus may be indirectly affected by the procedure call. However, we will see that the semantics of procedure calls is nevertheless compositional; i.e., once the behavior of a procedure is known in the empty context, this uniquely determines the behavior in any other context.

5.3 Weakening

Before we can fully describe the semantics of procedure calls, we first need to explore the concept of weakening, i.e., the effect of adding dummy variables to a flow chart. Recall that the semantics of a flow chart X is given by a linear function F from matrix tuples to matrix tuples, as discussed in Section 4.7. This situation is shown schematically in Figure 7(a). In general, X may have several incoming and outgoing control edges, but for simplicity we consider the case where there is only one of each.

Now suppose that we modify the flow chart X by picking a fresh boolean variable b and adding it to the context of all the edges of X. The result is a new flow chart X_b, which is schematically shown in Figure 7(b). We claim that the semantics of the modified flow chart X_b is given by $G(A, B) = (F(A), F(B))$. This is easily proved by induction on flow charts: all the basic components have this property, and the property is preserved under horizontal and vertical composition and under the introduction of loops. Intuitively, since the variable b does not occur in X, its value is neither altered not does it affect the computation of F.

Analogously, we can modify X by adding a fresh quantum variable q to all its edges, as shown schematically in

Figure 7(c). Then the semantics of the modified chart X_q is given by the function

$$G\left(\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right) = \left(\begin{array}{c|c} F(A) & F(B) \\ \hline F(C) & F(D) \end{array}\right).$$

This, too, is easily proved by induction.

At this point, we should make the following interesting observation. Note that, if $\left(\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right)$ is a density matrix, then so are A and D, but not necessarily B and C. In fact, up to a scalar multiple, B may be completely arbitrary. If the function F was defined only on density matrices, then $F(B)$ and $F(C)$ would be in general undefined, and thus, G would be undefined. This is the reason why, in Section 4.7, we defined the semantics of a flow chart to be a function on arbitrary matrices, and not just on density matrices, as one might have expected.

However, the remark of the previous paragraph is only of notational, not of fundamental, importance. By Remark 2.2, the density matrices span $\mathbb{C}^{n\times n}$ as a complex vector space. Since F is a linear function, this implies that F is already determined by its value on density matrices. Thus, the fact that F is given as a function on all matrices conveys no additional information.

5.4 Semantics of non-recursive procedure calls

The intended semantics of a non-recursive procedure call is that of "inlining": a procedure call should behave exactly as if the body of the procedure was inserted in its place. Before the procedure body can be inserted, it needs to be transformed in two steps: first, appropriate renamings (as in Example 4.3) need to be inserted to match the formal parameters with the actual ones. Second, the procedure body needs to be weakened in the sense of Section 5.3, i.e., all variables in the context of the procedure call that are not parameters must be added as dummy variables to the procedure body. If necessary, the local variables of the procedure body must be renamed to avoid name clashes with these dummy variables.

The semantics of a procedure call can be computed compositionally, i.e., without having to do the actual inlining. Namely, the renaming step does not affect the semantics at all, and the semantics of the weakening step can be computed as in Section 5.3.

5.5 Recursive procedures

A procedure is recursive if it invokes itself, either directly or indirectly. An example of a recursive procedure is shown in Figure 8(a). Before reading on, the reader is invited to figure out what this procedure does.

The intended semantics of recursive procedures is given by infinite unwinding, similar to the way we treated loops. Unwinding the procedure X from Figure 8(a) yields the infinite flow chart shown in Figure 8(b). This example demonstrates that the unwinding of a recursive procedure may lead to a flow chart with an unbounded number of variables, as new local variables are introduced at each level of nesting. The typing conventions enforce that such such qbits will eventually be deallocated before the procedure returns.

To compute the semantics of a recursive procedure, we could, in principle, annotate its infinite unwinding with states just as we did for loops. However, since the number of variables keeps increasing with each nesting level, this would require writing an infinite number of larger and larger matrices, and the computation of the resulting limits would be rather cumbersome. Therefore, we skip the explicit annotation and move on to a more denotational (and more practical) approach to calculating the semantics of X.

To find a good description of the unwinding process, let us write $X(Y)$ for the flow chart which is the same as X, except that it has another flow chart Y substituted in place of the recursive call. We can then define the ith unwinding of X to be the flow chart Y_i, where Y_0 is a non-terminating program, and $Y_{i+1} = X(Y_i)$.

Now let us write F_i for the semantics of the flow chart Y_i just defined. By compositionality, the semantics of $X(Y)$ is a function of the semantics of Y. If Φ denotes this function, then we can recursively compute F_i for all i via the clauses $F_0 = 0$ and $F_{i+1} = \Phi(F_i)$. Finally, it is natural to define the semantics of X to be the limit of this sequence,

$$G = \lim_{i\to\infty} F_i. \tag{2}$$

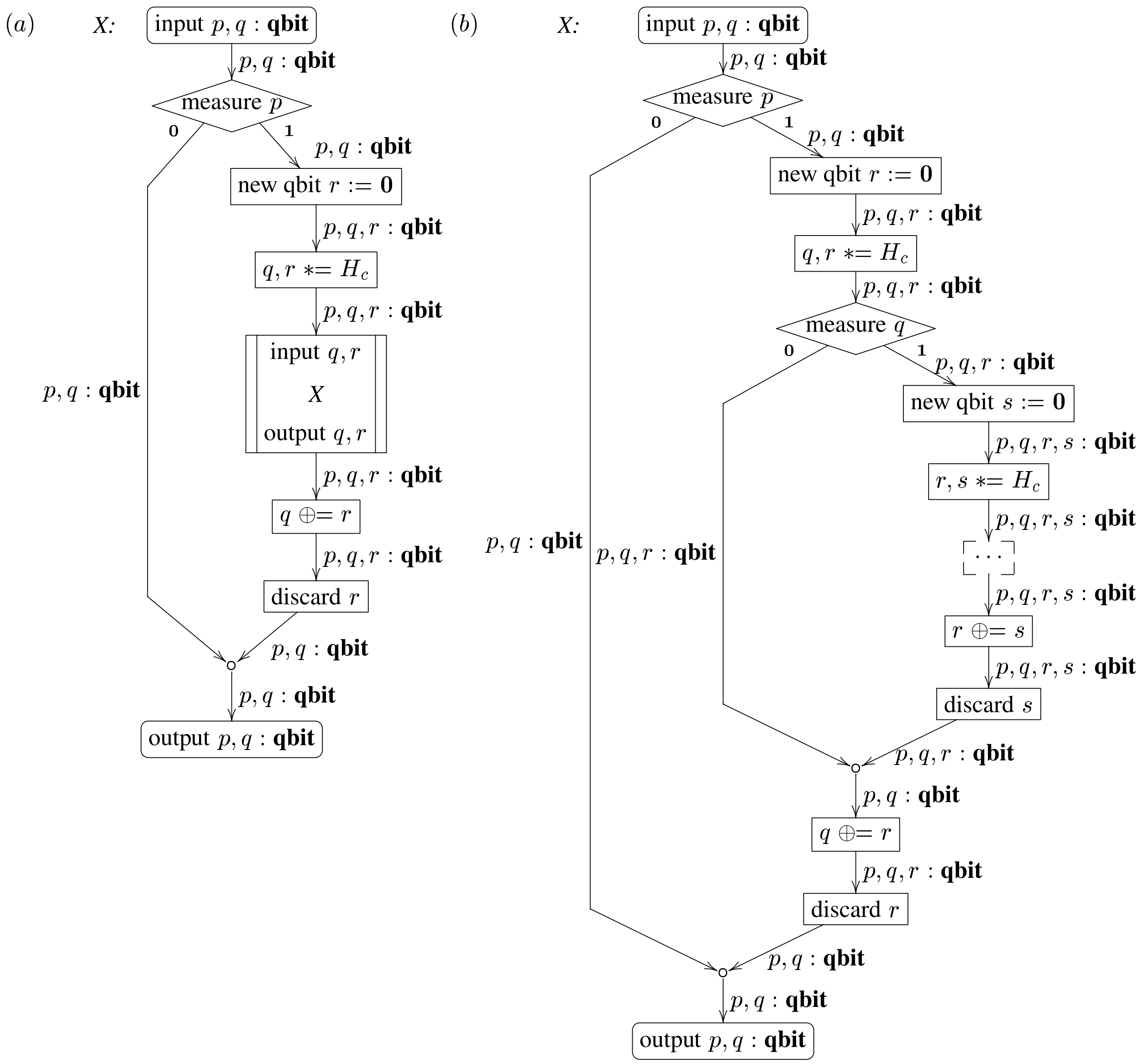


Figure 8: A recursive procedure and its unwinding

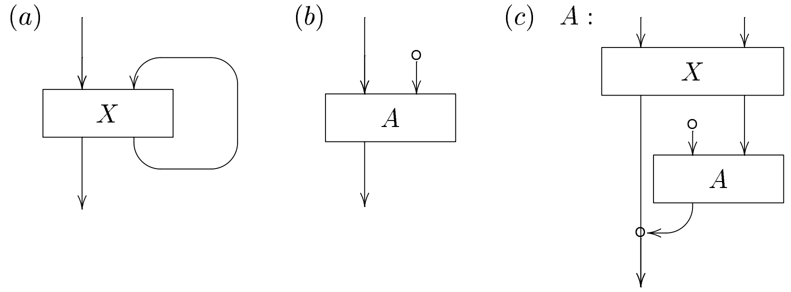


Figure 9: Loops from recursion

The existence of this limit will be justified in Section 6. For now, let us demonstrate the use of this method by computing the denotation of the sample flow chart from Figure 8. If $A = (a_{ij})_{ij}$, then we find

$$F_1(A) = \begin{pmatrix} a_{00} & a_{01} & 0 & 0 \\ a_{10} & a_{11} & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}, F_2(A) = \begin{pmatrix} a_{00} & a_{01} & 0 & 0 \\ a_{10} & a_{11} & 0 & 0 \\ 0 & 0 & a_{22} & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}, F_3(A) = \begin{pmatrix} a_{00} & a_{01} & 0 & 0 \\ a_{10} & a_{11} & 0 & 0 \\ 0 & 0 & a_{22} & 0 \\ 0 & 0 & 0 & \frac{1}{2}a_{33} \end{pmatrix},$$

$$F_4(A) = \begin{pmatrix} a_{00} & a_{01} & 0 & 0 \\ a_{10} & a_{11} & 0 & 0 \\ 0 & 0 & a_{22} + \frac{1}{4}a_{33} & 0 \\ 0 & 0 & 0 & \frac{1}{2}a_{33} \end{pmatrix}, F_5(A) = \begin{pmatrix} a_{00} & a_{01} & 0 & 0 \\ a_{10} & a_{11} & 0 & 0 \\ 0 & 0 & a_{22} + \frac{1}{4}a_{33} & 0 \\ 0 & 0 & 0 & \frac{1}{2}a_{33} + \frac{1}{8}a_{33} \end{pmatrix},$$

$$F_6(A) = \begin{pmatrix} a_{00} & a_{01} & 0 & 0 \\ a_{10} & a_{11} & 0 & 0 \\ 0 & 0 & a_{22} + \frac{1}{4}a_{33} + \frac{1}{16}a_{33} & 0 \\ 0 & 0 & 0 & \frac{1}{2}a_{33} + \frac{1}{8}a_{33} \end{pmatrix},$$

and so forth. The limit is

$$G(A) = \begin{pmatrix} a_{00} & a_{01} & 0 & 0 \\ a_{10} & a_{11} & 0 & 0 \\ 0 & 0 & a_{22} + \frac{1}{3}a_{33} & 0 \\ 0 & 0 & 0 & \frac{2}{3}a_{33} \end{pmatrix},$$

and this is the denotation of X. Note that, in this example, $\operatorname{tr} G(A) = \operatorname{tr} A$, which means this particular procedure terminates with probability 1. In general, it is possible that $\operatorname{tr} G(A) < \operatorname{tr} A$.

5.6 Recursion vs. loops

It is possible to encode loops in terms of recursion. Namely, the loop in Figure 9(a) can be expressed as the procedure call (b), where A is defined recursively as in (c). On the other hand, recursion cannot in general be encoded in terms of loops. This is because recursive procedures can allocate an unbounded number of variables.

6 Formal semantics

In this section, we give a more systematic and formal treatment of the semantics of quantum flow charts. We justify the well-definedness of the various constructions that were introduced informally in Sections 4 and 5, and in particular the existence of the limits in equations (1) and (2). We also consider a more abstract view of the semantics in terms of $\mathbf{CPO}$-enriched traced monoidal categories.

6.1 Signatures and matrix tuples

As outlined informally in Section 4.7, the denotation of a flow chart is given as a certain linear function from matrix tuples to matrix tuples. We begin our formal semantics by defining the spaces of such matrix tuples.

A *signature* is a list of non-zero natural numbers $\sigma = n_1, \ldots, n_s$. To each signature we associate a complex vector space

$$V_\sigma = \mathbb{C}^{n_1 \times n_1} \times \ldots \times \mathbb{C}^{n_s \times n_s}.$$

The elements of V_σ are tuples of matrices of the form $A = (A_1, \ldots, A_s)$, where the number and dimensions of the matrices A_i are determined by σ. As before, we often use the letters $A, B, \ldots$ to denote elements of V_σ. The notions of trace and norm are extended to matrix tuples as follows:

$$\begin{array}{rcl} \operatorname{tr} A & := & \sum_i \operatorname{tr} A_i, \\ |A|^2 & := & \sum_i |A_i|^2. \end{array}$$

We say that a matrix tuple $A \in V_\sigma$ is *hermitian* (respectively, *positive*) if A_i is hermitian (respectively, positive) for all i. We define the set $D_\sigma \subseteq V_\sigma$ of *density matrix tuples* to be the obvious generalization of the set D_n of density matrices:

$$D_\sigma = \{A \in V_\sigma \mid A \text{ positive hermitian and } \operatorname{tr} A \leqslant 1\}.$$

We define a partial order on matrix tuples by letting $A \sqsubseteq B$ if $B - A$ is positive. This makes D_σ into a complete partial order with least element 0. Completeness follows from Proposition 3.5, together with the fact that D_σ is a closed subset of $D_{n_1} \times \ldots \times D_{n_s}$.

Definition (Special signatures). Several signatures have special names; we write

$$\begin{array}{rcl} \textbf{bit} & = & 1, 1 \\ \textbf{qbit} & = & 2 \\ \mathbf{I} & = & 1 \\ \mathbf{0} & = & \epsilon \text{ (the empty list)} \end{array}$$

We call a signature *simple* if it is a singleton list. Thus, for example, **qbit** and **I** are simple, whereas **bit** and **0** are not.

6.2 The category V

Definition. The category $\mathbf{V}$ has as its objects signatures $\sigma = n_1, \ldots, n_s$. A morphism from σ to σ' is any complex linear function $F : V_\sigma \to V_{\sigma'}$.

Note that $\mathbf{V}$, as a category, is equivalent to the category of finite dimensional complex vector spaces. However, we will later use the additional, non-categorical structure on objects to define an interesting subcategory $\mathbf{Q}$ which has fewer isomorphic objects than $\mathbf{V}$.

Let $\sigma \oplus \sigma'$ denote concatenation of signatures. Then $\sigma \oplus \sigma'$ is a product and coproduct in $\mathbf{V}$, with the obvious injection and projection maps. The co-pairing map $[F, G] : \sigma \oplus \sigma' \to \tau$ is given by $[F, G](A, B) = F(A) + G(B)$, and the pairing map $\langle F, G \rangle : \sigma \to \tau \oplus \tau'$ is given by $\langle F, G \rangle(A) = (A, A)$. The neutral object for this biproduct is the empty signature $\mathbf{0}$.

Tensor product. If $\sigma = n_1, \ldots, n_s$ and $\tau = m_1, \ldots, m_t$ are signatures, their *tensor product* $\sigma \otimes \tau$ is defined as

$$\sigma \otimes \tau = n_1 m_1, \ldots, n_1 m_t, \ \ldots, \ n_s m_1, \ldots, n_s m_t.$$

Note that the components of $\sigma \otimes \tau$ are ordered lexicographically. The operation $\otimes$ extends to a symmetric monoidal structure on $\mathbf{V}$ with unit $I = 1$. The morphism part of the tensor product is defines as in the category of vector spaces; thus, if $F : \sigma \to \tau$ and $G : \sigma' \to \tau'$, then $F \otimes G : \sigma \otimes \sigma' \to \tau \otimes \tau'$ is defined on a basis element $A \otimes B$ via $(F \otimes G)(A \otimes B) = F(A) \otimes G(B)$, and extends to arbitrary elements by linearity. We note that this monoidal structure is strict (i.e., the associativity and unit morphisms are identity maps, rather than just isomorphisms). We also have the following strict distributivity law:

$$(\sigma \oplus \sigma') \otimes \tau = (\sigma \otimes \tau) \oplus (\sigma' \otimes \tau).$$

6.3 Superoperators

Definition (Completely positive operators, superoperators). Let $F : V_\sigma \to V_{\sigma'}$ be a linear function. We say that F is *positive* if $F(A)$ is positive for all positive $A \in V_\sigma$. We say that F is *completely positive* if $\mathrm{id}_\tau \otimes F : V_{\tau\otimes\sigma} \to V_{\tau\otimes\sigma'}$ is positive for all signatures τ. Finally, F is called a *superoperator* if it is completely positive and satisfies the following trace condition: $\mathrm{tr}\, F(A) \leqslant \mathrm{tr}\, A$, for all positive $A \in V_\sigma$.

Remark 6.1. In the physics literature, superoperators are usually assumed to be trace preserving, i.e., satisfying $\mathrm{tr}\, F(A) = \mathrm{tr}\, A$ for all A (see e.g. [10]). In our setting, it is appropriate to relax this condition in view of possible non-termination of programs.

Example 6.2. To illustrate the concept of a completely positive operator, consider the following three linear maps $F_i : V_2 \to V_2$.

$$F_1\begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} a & 2b \\ 2c & d \end{pmatrix}, \qquad F_2\begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} a & c \\ b & d \end{pmatrix}, \qquad F_3\begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} a & 0 \\ 0 & d \end{pmatrix}.$$

All three maps are linear, and thus morphisms in $\mathbf{V}$. Also, all three maps preserve trace and hermitian matrices. F_1 is not positive, because it maps a positive matrix to a non-positive matrix:

$$F_1\begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix}.$$

F_2 is positive, but not completely positive. While F_2 maps positive matrices to positive matrices, the same is not true for $\mathrm{id}_2 \otimes F_2$, for instance,

$$(\mathrm{id}_2 \otimes F_2)\begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

which is not positive. Finally, F_3 is completely positive.

As we will see in Section 6.9, superoperators are precisely the functions which arise as the denotations of quantum programs.

Lemma 6.3. *The following hold in the category* $\mathbf{V}$*:*

(a) Identity morphisms are superoperators, and superoperators are closed under composition.

(b) The canonical injections $\mathrm{in}_1 : \sigma \to \sigma \oplus \sigma'$ *and* $\mathrm{in}_2 : \sigma' \to \sigma \oplus \sigma'$ *are superoperators, and if* $F : \sigma \to \sigma'$ *and* $G : \tau \to \sigma'$ *are superoperators, then so is* $[F, G] : \sigma \oplus \tau \to \sigma'$*.*

(c) If $F : \sigma \to \sigma'$ *and* $G : \tau \to \tau'$ *are superoperators, then so are* $F \oplus G : \sigma \oplus \sigma' \to \tau \oplus \tau'$ *and* $F \otimes G : \sigma \otimes \sigma' \to \tau \otimes \tau'$*.*

(d) A morphism $F : \sigma \to \sigma'$ *is a superoperator if and only if* $\mathrm{id}_\tau \otimes F$ *is positive, for all* simple *signatures* τ*.*

(e) Let S *be a unitary* $n \times n$*-matrix. Then the morphism* $F : n \to n$ *defined by* $F(A) = SAS^*$ *is a superoperator.*

(f) Let S_1 *and* S_2 *be* $n \times n$*-matrices such that* $S_1^* S_1 + S_2^* S_2 = I$*. Then the morphism* $F : n \to n, n$ *defined by* $F(A) = (S_1 A S_1^*, S_2 A S_2^*)$ *is a superoperator.*

Proof. (a) and (b) are trivial. The first part of (c) follows from (b). For the second part of (c), note that $F \otimes G = (\mathrm{id}_{\sigma'} \otimes G) \circ (F \otimes \mathrm{id}_\tau)$. The two component maps are completely positive by definition, and they clearly satisfy the trace condition. For (d), only the right-to-left implication is interesting. Any object τ can be written as a sum $\tau = \tau_1 \oplus \ldots \oplus \tau_t$ of simple objects. Then by distributivity, $\mathrm{id}_\tau \otimes F = (\mathrm{id}_{\tau_1} \otimes F) \oplus \ldots \oplus (\mathrm{id}_{\tau_t} \otimes F)$, which is positive by assumption and (c). For (e), first note that if A is positive, then so is SAS^*, and $\mathrm{tr}\, SAS^* = \mathrm{tr}\, A$. Thus F is positive and satisfies the trace condition. To see that it is completely positive, note that for any n and identity $n \times n$-matrix I, $(\mathrm{id}_n \otimes F)(A) = (I \otimes S)A(I \otimes S)^*$. But $I \otimes S$ is unitary, thus $\mathrm{id}_n \otimes F$ is again of the same form as F, hence positive. By (d), it follows that F is a superoperator. For (f), note that F preserves positivity and trace, thus F is positive. The fact that it is completely positive follows as in (e).

6.4 The category $\mathbf{Q}$

Definition. The category $\mathbf{Q}$ is the subcategory of $\mathbf{V}$ which has the same objects of $\mathbf{V}$, and whose morphisms are the superoperators.

By Lemma 6.3(a)–(c), $\mathbf{Q}$ is indeed a subcategory of $\mathbf{V}$, and it inherits coproducts and the symmetric monoidal structure from $\mathbf{V}$. However, unlike $\mathbf{V}$, the category $\mathbf{Q}$ does not have finite products. This is because the diagonal morphism $[\mathrm{id}, \mathrm{id}] : \tau \to \tau \oplus \tau$ does not respect trace, and hence it is not a superoperator. However, the two projections $\pi_1 : \sigma \oplus \sigma' \to \sigma$ and $\pi_2 : \sigma \oplus \sigma' \to \sigma'$ are present in $\mathbf{Q}$.

Also note that the category $\mathbf{Q}$ distinguishes more objects than $\mathbf{V}$; for instance, the objects $\mathbf{bit} \oplus \mathbf{bit} = 1, 1, 1, 1$ and $\mathbf{qbit} = 2$ are isomorphic in $\mathbf{V}$, but not in $\mathbf{Q}$.

$\mathbf{CPO}$-enrichment. Recall that D_σ is the subset of V_σ consisting of density matrix tuples, i.e., positive matrix tuples A with $\operatorname{tr} A \leqslant 1$. Every superoperator $F : V_\sigma \to V_\tau$ restricts to a function $F : D_\sigma \to D_\tau$. We note that F respects the partial order $\sqsubseteq$: if $A \sqsubseteq B \in D_\sigma$, then $B = A + A'$ for some $A' \in D_\sigma$, and thus $F(B) = F(A) + F(A')$, which implies $F(A) \sqsubseteq F(B)$. Also, $F : D_\sigma \to D_{\sigma'}$ preserves least upper bounds of increasing sequences. This follows from Remark 3.6 and the fact that F, as a linear function on a finite-dimensional vector space, it continuous with respect to the usual Euclidean topology. Thus, we obtain a forgetful functor $D : \mathbf{Q} \to \mathbf{CPO}$, from $\mathbf{Q}$ to the category of complete partial orders, which maps σ to D_σ and F to itself.

If σ and σ' are objects of $\mathbf{Q}$, we can also define a partial order on the hom-set $\mathbf{Q}(\sigma, \sigma')$, by letting $F \sqsubseteq G$ if for all $A \in D_\sigma$, $F(A) \sqsubseteq G(A)$.

Lemma 6.4. *The poset $\mathbf{Q}(\sigma, \sigma')$ is a complete partial order.*

Proof. Let $F_0 \sqsubseteq F_1 \sqsubseteq \ldots$ be an increasing sequence of morphisms in $\mathbf{Q}(\sigma, \sigma')$. Define $F : D_\sigma \to D_{\sigma'}$ as the pointwise limit: $F(A) = \bigvee_i F(A_i)$. By Remark 3.6, $F(A)$ is also the topological limit $F(A) = \lim_{i\to\infty} F(A_i)$, and it follows by continuity that F is linear on the convex subset $D_\sigma \subseteq V_\sigma$. Since, by Remark 2.2, D_σ spans V_σ, F can be extended to a unique linear function $F : V_\sigma \to V_{\sigma'}$, i.e., to a morphism of $\mathbf{V}$. F satisfies the trace condition and is positive by construction. To see that it is completely positive, note that for any object τ and any $B \in D_{\tau\otimes\sigma}$, $(\tau \otimes F)(B) = \lim_{i\to\infty}(\tau \otimes F_i)(B)$, and hence $\tau \otimes F$ is positive for the same reason as F. Thus, $F : \sigma \to \sigma'$ is a morphism of $\mathbf{Q}$, and hence the desired least upper bound of $(F_i)_i$.

Also, the categorical operations (composition, co-pairing, and tensor) are Scott-continuous, i.e., they preserve least upper bounds of increasing sequences. This makes $\mathbf{Q}$ into a $\mathbf{CPO}$-enriched category.

Trace. A *monoidal trace* on a monoidal category $(\mathbf{Q}, \oplus)$ is a natural family of operations

$$\mathrm{Tr}^{\tau}_{\sigma,\sigma'} : \mathbf{Q}(\sigma \oplus \tau, \sigma' \oplus \tau) \to \mathbf{Q}(\sigma, \sigma'),$$

subject to a number of equations [7, 6, 12]. A monoidal category with a monoidal trace is called a *traced monoidal category*. A monoidal trace is usually just called "trace", but we add the adjective "monoidal" here to to avoid confusion with the trace of a matrix as in Section 2.2.

The category $\mathbf{Q}$ is equipped with a monoidal trace for the monoid which is given by coproducts $\oplus$ (not for the tensor product $\otimes$). In fact, the construction of this monoidal trace is an instance of a general construction which works in any $\mathbf{CPO}$-enriched category with coproducts.

To define the monoidal trace of a morphism $F : \sigma \oplus \tau \to \sigma' \oplus \tau$, we construct a family of morphisms $H_i : \sigma \oplus \tau \to \sigma'$ as follows. We let $H_0 = 0$, the constant zero function. For all i, we define $H_{i+1} = [\mathrm{id}_{\sigma'}, H_i \circ \mathrm{in}_2] \circ F$. Then $H_0 \sqsubseteq H_1$, because H_0 is the least element in the given partial order. By monotonicity of the categorical operations (a consequence of $\mathbf{CPO}$-enrichment), it follows that $H_i \sqsubseteq H_{i+1}$ for all i. Hence $(H_i)_i$ is an increasing sequence. Let $H = \bigvee_i H_i : \sigma \oplus \tau \to \sigma'$ be the least upper bound. Finally, define $\operatorname{Tr} F = H \circ \mathrm{in}_1 : \sigma \to \sigma'$. It is standard to check that this construction indeed defines a monoidal trace, i.e., that it satisfies all the necessary equations [6].

In more concrete terms, suppose that $F : \sigma \oplus \tau \to \sigma' \oplus \tau$ has been decomposed into components $F_{11} : \sigma \to \sigma'$, $F_{12} : \sigma \to \tau$, $F_{21} : \tau \to \sigma'$, and $F_{22} : \tau \to \tau$ as in Section 5.1. Then we have

$$\begin{aligned}
&H_0(A, 0) = 0, \\
&H_1(A, 0) = F_{11}(A), \\
&H_2(A, 0) = F_{11}(A) + F_{12}F_{21}(A), etc,
\end{aligned}$$

so that

$$(\mathrm{Tr}\, F)(A) = H(A,0) = F_{11}(A) + \sum_{i=0}^{\infty} F_{12}(F_{22}^{i}(F_{21}(A))).$$

Comparing this to equation (1) of Section 5.1, we find that the monoidal trace is precisely the construction we want for the interpretation of loops. In particular, this justifies the convergence of the infinite sum in equation (1).

We also note that the monoidal trace is related to the tensor $\otimes$ by the following property: if $F : \sigma \oplus \tau \to \sigma' \oplus \tau$, and ρ is any object, then

$$\mathrm{Tr}(F \otimes \rho) = (\mathrm{Tr}\, F) \otimes \rho.$$

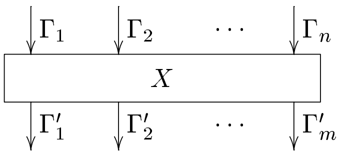
Here it is understood that we identify the objects $(\sigma \oplus \tau) \otimes \rho$ and $(\sigma \otimes \rho) \oplus (\tau \otimes \rho)$ (which happen to be identical anyway). We can summarize this property together with distributivity by saying that for any ρ, the functor $(-) \otimes \rho$ is a traced monoidal functor. We call a traced monoidal category with this additional structure a *traced monoidal category with weakening*.

6.5 The interpretation of flow charts

To each type A, we associate an object $[\![A]\!]$ of the category $\mathbf{Q}$. There are only two types, and their interpretations are suggested by the names of the corresponding objects: $[\![\mathbf{bit}]\!] = \mathbf{bit}$ and $[\![\mathbf{qbit}]\!] = \mathbf{qbit}$. To each typing context $\Gamma = x_1{:}A_1, \ldots, x_n{:}A_n$, we associate an object $[\![\Gamma]\!]$ as follows:

$$[\![\Gamma]\!] = [\![A_1]\!] \otimes \ldots \otimes [\![A_n]\!]$$

Further, if $\bar{\Gamma} = \Gamma_1; \ldots; \Gamma_n$ is a list of typing contexts, we define $[\![\bar{\Gamma}]\!] = [\![\Gamma_1]\!] \oplus \ldots \oplus [\![\Gamma_n]\!]$. Each quantum flow chart



is interpreted as a morphism

$$[\![X]\!] : [\![\Gamma_1]\!] \oplus \ldots \oplus [\![\Gamma_n]\!] \to [\![\Gamma'_1]\!] \oplus \ldots \oplus [\![\Gamma'_m]\!]$$

in the category $\mathbf{Q}$. The interpretation is defined by induction on the construction of the flow chart.

Atomic charts. The basic flow charts from Figures 3 and 5, with context Γ empty, are interpreted as the following morphisms.

$[\![\text{new bit } b := \mathbf{0}]\!]$	$=$	*newbit* :	$\mathbf{I} \to \mathbf{bit}$:	$\textit{newbit}(a) = (a, 0)$
$[\![\text{new qbit } q := \mathbf{0}]\!]$	$=$	*newqbit* :	$\mathbf{I} \to \mathbf{qbit}$:	$\textit{newqbit}(a) = \begin{pmatrix} a & 0 \\ 0 & 0 \end{pmatrix}$
$[\![\text{discard } b]\!]$	$=$	*discardbit* :	$\mathbf{bit} \to \mathbf{I}$:	$\textit{discardbit}(a,b) = a + b$
$[\![\text{discard } q]\!]$	$=$	*discardqbit* :	$\mathbf{qbit} \to \mathbf{I}$:	$\textit{discardqbit}\begin{pmatrix} a & b \\ c & d \end{pmatrix} = a + d$
$[\![b := \mathbf{0}]\!]$	$=$	*set*$_0$:	$\mathbf{bit} \to \mathbf{bit}$:	$\textit{set}_0(a,b) = (a+b, 0)$
$[\![b := \mathbf{1}]\!]$	$=$	*set*$_1$:	$\mathbf{bit} \to \mathbf{bit}$:	$\textit{set}_1(a,b) = (0, a+b)$
$[\![\bar{q} \mathrel{*}= S]\!]$	$=$	*unitary*$_S$:	$\mathbf{qbit}^n \to \mathbf{qbit}^n$:	$\textit{unitary}_S(A) = SAS^*$
$[\![\text{branch } b]\!]$	$=$	*branch* :	$\mathbf{bit} \to \mathbf{bit} \oplus \mathbf{bit}$:	$\textit{branch}(a,b) = (a, 0, 0, b)$
$[\![\text{measure } q]\!]$	$=$	*measure* :	$\mathbf{qbit} \to \mathbf{qbit} \oplus \mathbf{qbit}$:	$\textit{measure}\begin{pmatrix} a & b \\ c & d \end{pmatrix} = (\begin{pmatrix} a & 0 \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 \\ 0 & d \end{pmatrix})$
$[\![\text{merge}]\!]$	$=$	*merge* :	$\mathbf{I} \oplus \mathbf{I} \to \mathbf{I}$:	$\textit{merge}(a,b) = a + b$
$[\![\text{initial}]\!]$	$=$	*initial* :	$\mathbf{0} \to \mathbf{I}$:	$\textit{initial}(0) = 0$
$[\![\text{permute } \phi]\!]$	$=$	*permute*$_\phi$:	$A_1 \otimes \ldots \otimes A_n \to A_{\phi(1)} \otimes \ldots \otimes A_{\phi(n)}$	.

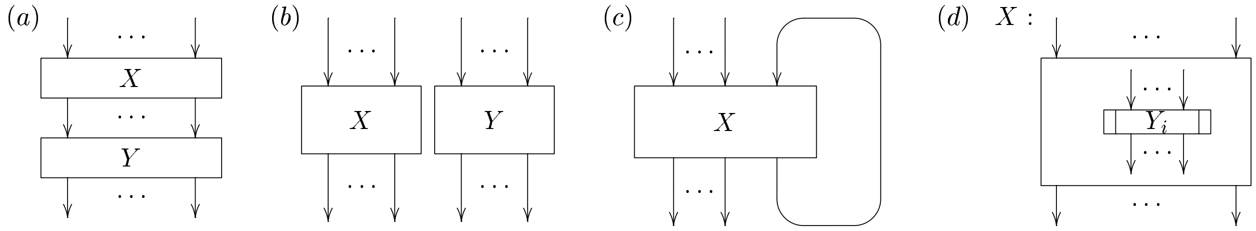


Figure 10: Some composite flow charts

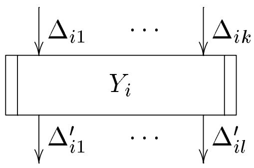
Here, *permute*$_\phi$ is the natural permutation map based on the symmetric tensor $\otimes$. There is also the trivial flow chart, consisting of one edge only, which is naturally interpreted as the identity map, and the flow chart consisting of two wires crossing each other, which is interpreted as the symmetry map for coproducts, $[\mathrm{in}_2, \mathrm{in}_1] : \sigma \oplus \tau \to \tau \oplus \sigma$.

Composite charts. Complex flow charts are built by combining simpler ones in one of the following ways:

1. Adding variables (weakening): If Y is obtained from X by adding an additional context Γ to all the edges of X, then
$$[\![Y]\!] = [\![X]\!] \otimes [\![\Gamma]\!].$$
2. Vertical composition: If A is the vertical composition of X and Y, as in Figure 10(a), then
$$[\![A]\!] = [\![Y]\!] \circ [\![X]\!].$$
3. Horizontal composition: If B is the horizontal composition of X and Y, as in Figure 10(b), then
$$[\![B]\!] = [\![X]\!] \oplus [\![Y]\!]$$
4. Loops: If C is obtained from X by introducing a loop, as in Figure 10(c), then
$$[\![C]\!] = \mathrm{Tr}([\![X]\!])$$

Two important results from the theory of traced monoidal categories ensure that this interpretation is well-defined: first, every possible flow chart (not containing procedure calls) can be build up from basic flow charts and the operations in Figure 10(a)-(c). Second, if there is more than one way of constructing a given flow chart from smaller pieces, the equations of traced monoidal categories with weakening guarantee that the resulting interpretations coincide, i.e., the interpretation is independent of this choice.

Procedures and recursion. For dealing with procedures, we formally augment the flow chart language with a set $Y_1, \ldots, Y_n$ of flow chart variables, each with a type $Y_i : \bar{\Delta}_i \to \bar{\Delta}'_i$. If Y_i is such a variable, then we allow the flow chart



to be used as a basic component. We write $X = X(Y_1, \ldots, Y_n)$ for a flow chart X which depends on $Y_1, \ldots, Y_n$, a situation which is shown schematically in Figure 10(d). The interpretation of such an X is given relative to an *environment* ρ, which is an assignment which maps each variable $Y_i : \bar{\Delta}_i \to \bar{\Delta}'_i$ to a morphism $\rho(Y_i) : \sigma_i \to \sigma'_i$ of the appropriate type in $\mathbf{Q}$. The interpretation $[\![X]\!]_\rho$ of X with respect to an environment ρ is given inductively, which base case $[\![Y_i]\!]_\rho = \rho(Y_i)$, and inductive cases as before. In this way, each flow chart $X(Y_1, \ldots, Y_n)$ defines a function

$$\Phi_X : \mathbf{Q}(\sigma_1, \sigma'_1) \times \ldots \times \mathbf{Q}(\sigma_n, \sigma'_n) \to \mathbf{Q}(\tau, \tau'),$$

which maps an environment $\rho = (f_1, \ldots, f_n)$ to $[\![X]\!]_\rho$. An easy induction shows that this function is Scott-continuous; in fact, it is given by a term in the language of traced monoidal categories with weakening.

The interpretation of recursive procedures is then given by the solution of fixpoint equations. In concrete terms, if the procedures $Y_1, \ldots, Y_n$ are defined by mutual recursion by a set of equations $Y_i = X_i(Y_1, \ldots, Y_n)$, for $i = 1, \ldots, n$, then their joint interpretation is given as the least fixpoint of the Scott-continuous function

$$\langle \Phi_{X_1}, \ldots, \Phi_{X_n} \rangle : \mathbf{Q}(\sigma_1, \sigma_1') \times \ldots \times \mathbf{Q}(\sigma_n, \sigma_n') \to \mathbf{Q}(\sigma_1, \sigma_1') \times \ldots \times \mathbf{Q}(\sigma_n, \sigma_n').$$

As a special case, let us consider the case of a single recursive procedure Y, defined by $Y = X(Y)$ for some flow chart X. In this case, X defines a Scott-continuous function

$$\Phi_X : \mathbf{Q}(\sigma, \sigma') \to \mathbf{Q}(\sigma, \sigma').$$

The interpretation $[\![Y]\!]$ will be given as the least fixpoint of Φ_X. This fixpoint can be calculated as the limit of an increasing sequence $F_0 \sqsubseteq F_1 \sqsubseteq \ldots$, where $F_0 = 0$, then constant zero function, and $F_{i+1} = \Phi_X(F_i)$. We find that

$$[\![Y]\!] = \bigvee_i F_i = \lim_i F_i.$$

Comparing this to equation (2) of Section 5.5, we find that this least fixpoint is precisely the required interpretation of the recursively defined procedure $Y = X(Y)$. In particular, since least fixpoints of Scott-continuous endofunctions on pointed complete partial orders always exist, this justifies the convergence of the limit in equation (2).

6.6 Structural and denotational equivalence

The interpretation of quantum flow charts can be generalized from the category $\mathbf{Q}$ to any category which has the requisite structure.

Definition. An *elementary quantum flow chart category* is a symmetric monoidal category with traced finite coproducts, such that $A \otimes (-)$ is a traced monoidal functor for every object A, together with a distinguished object **qbit** and morphisms $\iota : I \oplus I \to \mathbf{qbit}$ and $p : \mathbf{qbit} \to I \oplus I$, such that $p \circ \iota = \mathrm{id}$. Here I is the unit of the symmetric monoidal structure.

In an elementary quantum flow chart category, we define an object **bit** $:= I \oplus I$. Then the morphisms *newbit*, *discardbit*, *set*$_0$, *set*$_1$, *branch*, *merge*, and *initial*, needed in the interpretation of atomic quantum flow charts, are definable from the finite coproduct structure. Furthermore, the morphisms *newqbit*, *discardqbit*, and *measure* can be defined in terms of ι and p. The only additional piece of information needed to interpret quantum flow charts in an elementary quantum flow chart category is an interpretation of built-in unitary operators.

Consider a flow chart language with loops, no recursion, and a certain set of built-in unitary operator symbols. Let $\mathbf{C}$ be an elementary quantum flow chart category, and let η be an assignment which maps each built-in n-ary operator symbol S to a morphism $\eta_S : \mathbf{qbit}^n \to \mathbf{qbit}^n$ in $\mathbf{C}$. Then there is an evident interpretation of quantum flow charts, which maps each $X : \bar{\Gamma} \to \bar{\Gamma}'$ to a morphism $[\![X]\!]_\eta : [\![\bar{\Gamma}]\!] \to [\![\bar{\Gamma}']\!]$, defined inductively as in Section 6.5. Further, if the category $\mathbf{C}$ is $\mathbf{CPO}$-enriched, then we can also interpret recursively defined flow charts in it.

Definition (Structural equivalence, denotational equivalence). Two quantum flow charts $X, Y : \bar{\Gamma} \to \bar{\Gamma}'$ are said to be *structurally equivalent* if for every elementary quantum flow chart category $\mathbf{C}$ and every interpretation η of basic operator symbols, $[\![X]\!]_\eta = [\![Y]\!]_\eta$. Further, X and Y are said to be *denotationally equivalent* if $[\![X]\!] = [\![Y]\!]$ for the canonical interpretation in the category $\mathbf{Q}$ of signatures and completely positive operators.

Clearly, structural equivalence implies denotational equivalence, and the converse is not true. Structural equivalence is essentially a syntactic notion: if two flow charts are structurally equivalent, then one can be obtained from the other by purely symbolic manipulations, without any assumptions about the behavior of the built-in unitary operators. For instance, the two flow charts in Example 4.2 are structurally equivalent, as are those in Example 4.7. On the other hand, the two flow charts in Example 4.6 are not structurally equivalent. Structural equivalence is probably the minimal reasonable equivalence which one might want to consider on flow charts.

Denotational equivalence, on the other hand, is a semantic notion. It captures precisely our concept of "behavior" of quantum programs, fully taking into account the meaning of the built-in operators. We should remark that, like any denotational notion of "behavior", denotational equivalence abstracts from some aspects of the actual physical behavior of a system; for instance, issues like the running time or space usage of an algorithm are not modeled. Denotational equivalence is only concerned with the probabilistic input-output relationship of programs. It is the *largest* possible equivalence on quantum flow charts in the following sense: if $X, Y : \bar{\Gamma} \to \bar{\Gamma}'$ are not denotationally equivalent, then there exists a context $C[-]$ (a flow chart with a "hole") such that $C[X]$ and $C[Y]$ are of type $I \to$ **bit**, and $C[X]$ evaluates to **0** with a different probability than $C[Y]$.

In light of this result, the denotational semantics of flow charts can be proven to be *fully abstract* with respect to a suitable operational semantics. Details will be given elsewhere.

6.7 Characterizations of completely positive operators and superoperators

We will now give some basic and well-known characterizations of superoperators. These results will be used in Section 6.9 to prove that every superoperator arises as the denotation of a quantum flow chart.

As before, let e_i denote the ith canonical unit column vector. The space $\mathbb{C}^{n\times n}$ of $n \times n$-matrices has a canonical basis (as a vector space), consisting of the matrices $E_{ij} = e_i e_j^*$. Any linear function $F : \mathbb{C}^{n\times n} \to \mathbb{C}^{m\times m}$ is uniquely determined by its action on the basis elements.

Definition. The *characteristic matrix* of a linear function $F : \mathbb{C}^{n\times n} \to \mathbb{C}^{m\times m}$ is the matrix $\chi_F \in \mathbb{C}^{nm\times nm}$ defined by

$$\chi_F = \left(\begin{array}{c|c|c} F(E_{11}) & \cdots & F(E_{1n}) \\ \hline \vdots & \ddots & \vdots \\ \hline F(E_{n1}) & \cdots & F(E_{nn}) \end{array}\right).$$

More generally, let $\sigma = n_1, \ldots, n_s$ and $\tau = m_1, \ldots, m_t$ be signatures, and let $F : V_\sigma \to V_\tau$ be a linear function. We define the ij-component of F to be the function $F_{ij} = \pi_j \circ F \circ \mathrm{in}_i : \mathbb{C}^{n_i \times n_i} \to \mathbb{C}^{m_j \times m_j}$. The *characteristic matrix tuple* of F is

$$\chi_F = (\chi_{F_{11}}, \ldots, \chi_{F_{1t}}, \quad \ldots \quad, \chi_{F_{s1}}, \ldots, \chi_{F_{st}}).$$

Note that if $F : V_\sigma \to V_\tau$ is a linear function, its characteristic matrix tuple is an element $\chi_F \in V_{\sigma\otimes\tau}$. Moreover, F and χ_F determine each other uniquely.

One might ask whether it is possible to characterize the completely positive operators, or respectively the superoperators, in terms of a property of their characteristic matrices. This is indeed possible. In the following theorems, we start by considering the simple case, i.e., the case of operators $F : \mathbb{C}^{n\times n} \to \mathbb{C}^{m\times m}$. The general non-simple case is treated afterwards.

Theorem 6.5. *Let $F : \mathbb{C}^{n\times n} \to \mathbb{C}^{m\times m}$ be a linear operator, and let $\chi_F \in \mathbb{C}^{nm\times nm}$ be its characteristic matrix.*

(a) F is of the form $F(A) = UAU^*$, for some $U \in \mathbb{C}^{n\times m}$, if and only if χ_F is pure.

(b) The following are equivalent:

(i) F is completely positive.

(ii) χ_F is positive.

(iii) F is of the form $F(A) = \sum_i U_i A U_i^*$, for some finite sequence of matrices $U_1, \ldots, U_k \in \mathbb{C}^{n\times m}$.

Proof. For part (a), observe that the matrix χ_F is pure iff it is of the form $\chi_F = uu^*$, for some $u \in \mathbb{C}^{nm}$. We can write

$$u = \left(\begin{array}{c} v_1 \\ \hline \vdots \\ \hline v_n \end{array}\right),$$

for some vectors $v_i \in \mathbb{C}^m$, and let $U = (v_1 | \ldots | v_n) \in \mathbb{C}^{n\times m}$. Then $F(E_{ij}) = v_i v_j^* = UE_{ij}U^*$, for all i, j, and thus $F(A) = UAU^*$ for all A. Conversely if $F(A) = UAU^*$, then $\chi_F = uu^*$ with u constructed from U as above.

For part (b), to show (i) $\Rightarrow$ (ii), it suffices to observe that the matrix

$$E = \left(\begin{array}{c|c|c} E_{11} & \cdots & E_{1n} \\ \hline \vdots & \ddots & \vdots \\ \hline E_{n1} & \cdots & E_{nn} \end{array}\right)$$

is positive, and that $\chi_F = (\mathrm{id}_n \otimes F)(E)$. To prove that (ii) $\Rightarrow$ (iii), assume that χ_F is positive. Then χ_F can be written as a sum of pure matrices, say, $\chi_F = B_1 + \ldots + B_k$. For each $i = 1, \ldots, k$, let F_i be the linear operator whose characteristic matrix is $\chi_{F_i} = B_i$. By part (a), $F_i(A) = U_i A U_i^*$, hence $F(A) = \sum_i U_i A U_i^*$ as desired. Finally, the implication (iii) $\Rightarrow$ (i) is trivial. $\square$

Corollary 6.6. *Two linear functions $F, G : \mathbb{C}^{n\times n} \to \mathbb{C}^{m\times m}$ satisfy $F \sqsubseteq G$ if and only if $\chi_F \sqsubseteq \chi_G$. Here $F \sqsubseteq G$ is defined as in Section 6.1 to mean $F(A) \sqsubseteq G(A)$ for all positive A.* $\square$

Next, we wish to characterize superoperators, i.e., completely positive operators which satisfy the trace condition. We start with some preliminary observations. First, note that for any signature σ, the *trace operator* $\mathrm{tr}_\sigma : V_\sigma \to V_1$ is a superoperator. We also call it the *erasure* map, as it corresponds to an erasure of (quantum and classical) information.

Definition. The *trace characteristic matrix tuple* of a linear function $F : V_\sigma \to V_\tau$ is defined to be $\chi_F^{(\mathrm{tr})} = \chi_{\mathrm{tr}_\tau \circ F} \in V_\sigma$, i.e., the characteristic matrix tuple of $\mathrm{tr}_\tau \circ F$. Note that $\chi_F^{(\mathrm{tr})}$ is easily calculated from χ_F by taking a "componentwise trace", i.e., $\chi_F^{(\mathrm{tr})} = (\mathrm{id}_\sigma \otimes \mathrm{tr}_\tau)(\chi_F)$.

Theorem 6.7. *Let $F : \mathbb{C}^{n\times n} \to \mathbb{C}^{m\times m}$ be a completely positive operator. The following are equivalent:*

(i) F is a superoperator.

(ii) $\chi_F^{(\mathrm{tr})} \sqsubseteq I_n$, where $I_n \in \mathbb{C}^{n\times n}$ is the identity matrix.

(iii) F is of the form $F(A) = \sum_i U_i A U_i^*$, for matrices $U_1, \ldots, U_k$ with $\sum_i U_i^* U_i \sqsubseteq I_n$.

Proof. For the equivalence of (i) and (ii), note that F is a superoperator iff $\mathrm{tr}\, F(A) \leqslant \mathrm{tr}\, A$, for all positive A. This is the case iff $\mathrm{tr}_m \circ F \sqsubseteq \mathrm{tr}_n$, and by Corollary 6.6, iff $\chi_{\mathrm{tr}_m \circ F} \sqsubseteq \chi_{\mathrm{tr}_n}$. But $\chi_{\mathrm{tr}_n} = I_n$, and thus this is equivalent to $\chi_F^{(\mathrm{tr})} \sqsubseteq I_n$. For the equivalence of (ii) and (iii), first note that by Theorem 6.5, F can be written as $F(A) = \sum_i U_i A U_i^*$, for some matrices $U_1, \ldots, U_k$. But then, $\chi_F^{(\mathrm{tr})} = \overline{\sum_i U_i^* U_i}$, the complex conjugate of $\sum_i U_i^* U_i$. Thus the equivalence follows. $\square$

The equivalence (i) $\iff$ (iii) is known as the *Kraus Representation Theorem*. Note that F is trace preserving, i.e., $\mathrm{tr}\, F(A) = \mathrm{tr}\, A$ for all A, iff $\chi_F^{(\mathrm{tr})} = I_n$ iff $\sum_i U_i^* U_i = I_n$. Theorems 6.5 and 6.7 can be straightforwardly generalized to the non-simple case, as summarized in the next theorem.

Theorem 6.8. *Let $\sigma = n_1, \ldots, n_s$ and $\tau = m_1, \ldots, m_t$ be signatures, and let $F : V_\sigma \to V_\tau$ be a linear function.*

(a) F is completely positive iff χ_F is positive.

(b) F is a superoperator iff χ_F is positive and $\chi_F^{(\mathrm{tr})} \sqsubseteq I_\sigma$, where $I_\sigma \in V_\sigma$ is the tuple consisting of identity matrices.

(c) F is a superoperator iff it can be written of the form $F(A_1, \ldots, A_s) = (\sum_{il} U_{i1l} A_i U_{i1l}^*, \ldots, \sum_{il} U_{itl} A_i U_{itl}^*)$, for matrices $U_{ijl} \in \mathbb{C}^{n_i \times m_j}$ where $\sum_{jl} U_{ijl}^* U_{ijl} \sqsubseteq I_{n_i}$ for all i. Here, l ranges over some finite index set.

Proof. All three parts follow from straightforward componentwise arguments. Let $F_{ij} = \pi_j \circ F \circ \mathrm{in}_i : V_{n_i} \to V_{m_j}$ be the ij-component of F as before, and let $F_i = F \circ \mathrm{in}_i : V_{n_i} \to V_\tau$. For (a), observe that F is completely positive iff each F_{ij} is completely positive. For (b), note that F satisfies the trace condition iff each F_i satisfies it. This is the case iff $\mathrm{tr}_\tau \circ F_i \sqsubseteq \mathrm{tr}_{n_i}$, or equivalently, $\chi_{F_i}^{(\mathrm{tr})} \sqsubseteq I_{n_i}$, for all i. The latter is equivalent to $\chi_F^{(\mathrm{tr})} \sqsubseteq I_\sigma$. For (c), first note that by Theorem 6.5, each F_{ij} can be written as $F_{ij}(A) = \sum_l U_{ijl} A_i U_{ijl}^*$, where $l = 1, \ldots, k_{ij}$. By setting $k = \max_{ij} k_{ij}$ and $U_{ijl} = 0$ if $l > k_{ij}$, we may assume that l ranges uniformly over $1, \ldots, k$. Thus, F can be written in the desired form; further $\mathrm{tr}\, F_i(A) = \mathrm{tr} \sum_{jl} U_{ijl} A_i U_{ijl}^*$, and hence, by Theorem 6.7(iii), F is a superoperator iff $\sum_{jl} U_{ijl}^* U_{ijl} \sqsubseteq I_{n_i}$ for all i. $\square$

Remark 6.9 *(Compact closed structure).* Recall that $\mathbf{V}$ and $\mathbf{Q}$ are symmetric monoidal categories whose objects are signatures, and whose morphisms are, respectively, linear functions, and superoperators. The characteristic matrix determines a one-to-one correspondence between hom-sets $\mathbf{V}(\rho \otimes \sigma, \tau) \cong \mathbf{V}(\rho, \sigma \otimes \tau)$ (they both correspond to matrix tuples in $V_{\rho \otimes \sigma \otimes \tau}$). Moreover, this one-to-one correspondence is natural in ρ and τ. A category with this property is called *compact closed*.

Let $\mathbf{W}$ be the category whose objects are signatures and whose morphisms are completely positive operators. Thus $\mathbf{Q} \subseteq \mathbf{W} \subseteq \mathbf{V}$. Theorem 6.8(a) implies that $\mathbf{W}$ inherits the compact closed structure from $\mathbf{V}$. However, by Theorem 6.8(b), the category $\mathbf{Q}$ of superoperators is *not* compact closed; indeed, if $F : V_\sigma \to V_\tau$ is a superoperator, then is characteristic matrix χ_F is not in general a density matrix, because in general, $\operatorname{tr} \chi_F > 1$. However, $\lambda \chi_F$ is a density matrix for some scalar λ with $0 < \lambda \leqslant 1$. In this sense, we may say that $\mathbf{Q}$ possesses a compact closed structure *up to scalar multiples*.

6.8 Normal form for superoperators

As a consequence of the in Theorems 6.5 and 6.7, we obtain a normal form for superoperators: any superoperator can be expressed as a sub-unitary transformation, followed by an erasure and a measurement. This normal form is not unique.

Definition. A matrix $U \in \mathbb{C}^{n \times m}$ is said to be *sub-unitary* if U is a submatrix of some unitary matrix U', i.e., if there exist matrices U_1, U_2, U_3 (not necessarily of the same dimensions as U) such that

$$U' = \left(\begin{array}{c|c} U & U_1 \\ \hline U_2 & U_3 \end{array}\right)$$

is unitary. A linear function $F : \mathbb{C}^{n \times n} \to \mathbb{C}^{m \times m}$ is called sub-unitary if it is of the form $F(A) = UAU^*$, for some sub-unitary matrix $U \in \mathbb{C}^{n \times m}$. More generally, a linear function $F : V_\sigma \to V_\tau$ is called sub-unitary if it is of the form $F(A_1, \ldots, A_s) = (U_1 A_1 U_1^*, \ldots, U_s A_s U_s^*)$, for sub-unitary matrices $U_i \in \mathbb{C}^{n_i \times m_i}$, where $\sigma = n_1, \ldots, n_s$ and $\tau = m_1, \ldots, m_s$.

Lemma 6.10. *A matrix $U \in \mathbb{C}^{n \times m}$ is sub-unitary iff $UU^* \sqsubseteq I_m$ iff $U^*U \sqsubseteq I_n$.*

Proof. Clearly, U is sub-unitary iff there exists a matrix U_1 such that the rows of $(U|U_1)$ form an orthonormal set. This is the case iff $UU^* + U_1 U_1^* = I_m$, and by Remark 2.3, iff $UU^* \sqsubseteq I_m$. The second equivalence is similar. □

Definition. Let $\sigma = n_1, \ldots, n_s$ be a signature, and let $\tilde{\sigma} = n_1 + \ldots + n_s$, an integer regarded as a simple signature. The *measurement operator* $\mu_\sigma : V_{\tilde{\sigma}} \to V_\sigma$ is defined as

$$\mu_\sigma \left(\begin{array}{c|c|c} A_{11} & \cdots & A_{1s} \\ \hline \vdots & \ddots & \vdots \\ \hline A_{s1} & \cdots & A_{ss} \end{array}\right) = (A_{11}, A_{22}, \ldots, A_{ss}),$$

where $A_{ij} \in \mathbb{C}^{n_i \times n_j}$. An *erasure operator* is any operator of the form $(\operatorname{tr}_\sigma \otimes \operatorname{id}_\tau) : V_{\sigma \otimes \tau} \to V_\tau$.

Note that sub-unitary transformations, measurements and erasures are superoperators; the following theorem states that any superoperator is a combination of these three basic ones.

Theorem 6.11. *(a) Every superoperator $F : \mathbb{C}^{n \times n} \to \mathbb{C}^{m \times m}$ can be factored as $F = E \circ G$, where G is sub-unitary and E is an erasure operator.*

(b) Every superoperator $F : V_\sigma \to V_\tau$ can be factored as $F = M \circ E \circ G$, where G is sub-unitary, E is an erasure operator, and M is a measurement operator.

Proof. (a) By Theorem 6.7, there exist matrices $U_1, \ldots, U_k \in \mathbb{C}^{n \times m}$ such that $F(A) = \sum_i U_i A U_i^*$ and $\sum_i U_i^* U_i \sqsubseteq I_n$. Let U be the vertical stacking of the matrices $U_1, \ldots, U_k$,

$$U = \left(\begin{array}{c} U_1 \\ \hline \vdots \\ \hline U_k \end{array}\right),$$

and define $G : \mathbb{C}^{n\times n} \to \mathbb{C}^{km\times km}$ by $G(A) = UAU^*$. Since $U^*U = \sum_i U_i^*U_i$, the matrix U is sub-unitary by Lemma 6.10. Also, let $E = (\mathrm{tr}_k \otimes \mathrm{id}_m) : \mathbb{C}^{km\times km} \to \mathbb{C}^{m\times m}$. Then $E(G(A)) = (\mathrm{tr}_k \otimes \mathrm{id}_m)(UAU^*) = \sum_i U_iAU_i^* = F(A)$, as claimed.

(b) Suppose $\sigma = n_1, \ldots, n_s$ and $\tau = m_1, \ldots, m_t$. By Theorem 6.8(c), F can be written as $F(A_1, \ldots, A_s) = (\sum_{il} U_{i1l}A_iU_{i1l}^*, \ldots, \sum_{il} U_{itl}A_iU_{itl}^*)$, for matrices $U_{ijl} \in \mathbb{C}^{n_i\times m_j}$ where $\sum_{jl} U_{ijl}^*U_{ijl} \sqsubseteq I_{n_i}$ for all i, and where l ranges over $1, \ldots, k$, for some k. For each i, let $U_i \in \mathbb{C}^{n_i\times k(m_1+\ldots+m_t)}$ be the vertical stacking of the matrices $U_{i11}, \ldots, U_{it1}, \ldots, U_{i1k}, \ldots, U_{itk}$. Let $\sigma' = k, \ldots, k$, a list of length s. Define $G : V_\sigma \to V_{\sigma'\otimes\tilde{\tau}}$ by $G(A_1, \ldots, A_s) = (U_1A_1U_1^*, \ldots, U_sA_sU_s^*)$. Clearly, $U_i^*U_i = \sum_{jl} U_{ijl}^*U_{ijl} \sqsubseteq I_{n_i}$ for all i, and thus G is sub-unitary. Let $E = (\mathrm{tr}_{\sigma'} \otimes \mathrm{id}_{\tilde{\tau}}) : V_{\sigma'\otimes\tilde{\tau}} \to V_{\tilde{\tau}}$, and let $M = \mu_\tau : V_{\tilde{\tau}} \to V_\tau$. An easy calculation shows that $M \circ E \circ G(A) = F(A)$, as desired. □

6.9 Fullness of the interpretation

We defined the interpretation of a quantum flow chart X to be a morphism $[\![X]\!]$ in the category **Q**, i.e., a superoperator. We now want to show that every superoperator is definable in this way. More precisely, we want to show that the interpretation is *full*: whenever $\bar{\Gamma}$ and $\bar{\Gamma}'$ are lists of typing contexts and $F : [\![\bar{\Gamma}]\!] \to [\![\bar{\Gamma}']\!]$ is a superoperator, then there exists a quantum flow chart $X : \bar{\Gamma} \to \bar{\Gamma}'$ such that $[\![X]\!] = F$.

For the purpose of this section, we consider the flow chart language which has loops, and which contains all unitary operators as built-in operators. In a more realistic setting, one would only have a finite, but complete set of built-in operators (in the sense of Proposition 3.2); in this case, fullness is true *up to epsilon*, i.e., for every $\epsilon > 0$, one can find X such that $\|[\![X]\!] - F\| < \epsilon$.

Lemma 6.12. *(a) If $\bar{\Gamma}$ and $\bar{\Gamma}'$ are lists of typing contexts such that $[\![\bar{\Gamma}]\!] = [\![\bar{\Gamma}']\!] = \sigma$, then there exists a flow chart $X : \bar{\Gamma} \to \bar{\Gamma}'$ such that $[\![X]\!] = \mathrm{id}_\sigma$.*

(b) Suppose $n = 2^k$, $m = 2^l$, and $F : \mathbb{C}^{n\times n} \to \mathbb{C}^{m\times m}$ is sub-unitary. Then there exists a quantum flow chart $X : \mathbf{qbit}^k \to \mathbf{qbit}^l$ such that $[\![X]\!] = F$.

Proof. For part (a), first note that if neither $\bar{\Gamma}$ nor $\bar{\Gamma}'$ contain the type **bit**, then $\bar{\Gamma} = \bar{\Gamma}'$ and there is nothing to show. Further, all occurrences of the type **bit** can be removed by repeated application of the transformation from Example 4.4.

For (b), we have $F(A) = UAU^*$ for some sub-unitary matrix $U \in \mathbb{C}^{n\times m}$. Then there exist matrices U_1, U_2, U_3, not necessarily of the same dimensions as U, such that

$$U' = \left(\begin{array}{c|c} U & U_1 \\ \hline U_2 & U_3 \end{array}\right) \in \mathbb{C}^{p\times p}$$

is unitary. Without loss of generality, we may assume that $p = 2^r$ is a power of two. Then $F = [\![X]\!]$, where X is the flow chart shown in Figure 11. Here we have used obvious abbreviations for multiple “new”, “measure” and “discard” operations. Note that all but one branch of the measurements lead into an infinite loop; this is due to the fact that F may not be trace preserving. □

Theorem 6.13 (Fullness). *For given lists of typing contexts $\bar{\Gamma}$, $\bar{\Gamma}'$, if $F : [\![\bar{\Gamma}]\!] \to [\![\bar{\Gamma}']\!]$ is a superoperator, then there exists a quantum flow chart $X : \bar{\Gamma} \to \bar{\Gamma}'$ such that $[\![X]\!] = F$.*

Proof. This is an almost trivial consequence of Theorem 6.11 and Lemma 6.12. First, by Lemma 6.12(a), it suffices to consider the case where $\bar{\Gamma} = \mathbf{qbit}^{k_1}; \ldots; \mathbf{qbit}^{k_s}$ and $\bar{\Gamma}' = \mathbf{qbit}^{l_1}; \ldots; \mathbf{qbit}^{l_t}$. Second, let $l \geqslant l_i$ for all i, and let $2^r > t$. Then F can be factored as $F_2 \circ F_1$, where $F_1 : [\![\bar{\Gamma}]\!] \to [\![\mathbf{qbit}^l \times \mathbf{bit}^r]\!]$ is a superoperator and $F_2 : [\![\mathbf{qbit}^l \times \mathbf{bit}^r]\!] \to [\![\bar{\Gamma}']\!]$ is a canonical projection. F_2 is clearly definable. Let $\sigma = [\![\bar{\Gamma}]\!]$ and $\tau = [\![\mathbf{qbit}^l \times \mathbf{bit}^r]\!]$. Then $\tilde{\tau} = [\![\mathbf{qbit}^{l+r}]\!]$. By the proof of Theorem 6.11, F_1 can be factored as $M \circ E \circ G$, where $G : \sigma \to \sigma' \otimes \tilde{\tau}$ is sub-unitary, $E : \sigma' \otimes \tilde{\tau} \to \tilde{\tau}$ is the canonical erasure operator, and $M : \tilde{\tau} \to \tau$ is the canonical measurement operator. Moreover, $\sigma' = k, \ldots, k$ is a list of length s, and without loss of generality, we may assume that $k = 2^p$ is a power of two, so that $\sigma' = [\![\mathbf{qbit}^p; \ldots; \mathbf{qbit}^p]\!]$. Now M is definable by Lemma 6.12(b), E is definable by a sequence of “discard” and “merge” operations, and $M : [\![\mathbf{qbit}^{l+r}]\!] \to [\![\mathbf{qbit}^l \times \mathbf{bit}^r]\!]$ is definable by r measurements. □

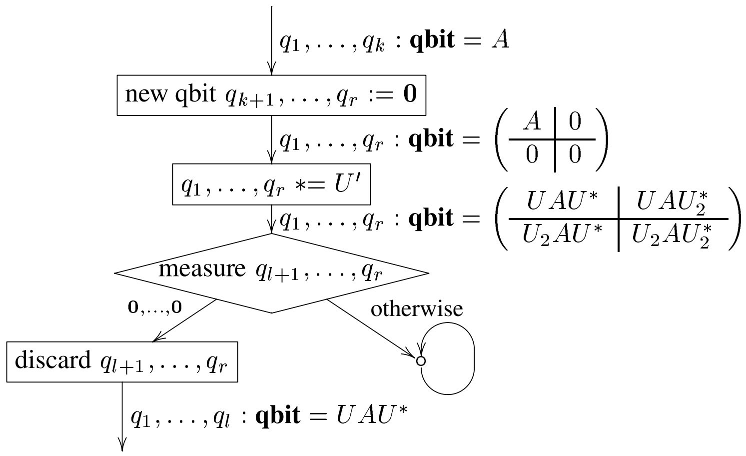


Figure 11: Flow chart realizing a sub-unitary transformation

7 Towards a structured syntax

In previous sections, we have presented a view of quantum programming in terms of flow charts. The reasons were partly pedagogical, because flow charts explicitly highlight the "atomic" concepts of control flow, which are often left implicit in more structured programming languages. Particularly the "merge" operation, with its associated erasure of classical information, is of fundamental importance to quantum computing because it causes the passage from pure to impure states. Another reason for presenting the language in terms of flow charts was semantical: flow charts, because of their close connection with traced monoidal categories, provide a convenient setting for describing the semantics of quantum programs.

However, for actual programming, flow charts are generally too cumbersome as a notation. The reasons for this are the same as in classical programming language theory: their graphical nature makes flow charts difficult to manipulate, and they also discourage a structured approach to programming. We now present a more "textual" syntax for quantum programs, which is also more "structured" in the sense of structured programming languages such as Pascal.

It is worth emphasizing, once again, that we are describing a *functional* programming language, despite the fact that its syntax superficially looks imperative. Each statement acts as a function from an explicitly identified set of inputs to outputs, rather than operating on an implicitely defined global state.

7.1 The language QPL

We assume a countable set of *variables*, denoted $x, y, b, q, \ldots$. We also assume a countable set of *procedure variables* $X, Y, \ldots$.

Types. A *type* t, s is either **bit** or **qbit**. A *procedure type* T is an expression of the form $s_1, \ldots, s_n \to t_1, \ldots, t_m$, where s_i and t_j are types. A *typing context* Γ is a finite list of pairs of a variable and a type, such that no variable occurs more than once. Typing contexts are written in the usual way as $x_1{:}t_1, \ldots, x_n{:}t_n$. A *procedure context* Π is defined similarly, except that it consists of procedure variables and procedure types. We use the notation $x{:}t, \Gamma$ and $X{:}T, \Pi$ for extension of contexts, and in using this notation, we always implicitly assume that the resulting context is well-defined, i.e., that x does not already occur in Γ and X does not already occur in Π.

Terms. The set of *QPL terms* is defined by the following abstract syntax:

$$
\begin{array}{lll}
\text{QPL Terms} \quad P, Q & ::= & \textbf{new bit}\ b := \mathbf{0} \;\big|\; \textbf{new qbit}\ q := \mathbf{0} \;\big|\; \textbf{discard}\ x \\
 & \big| & b := \mathbf{0} \;\big|\; b := \mathbf{1} \;\big|\; q_1, \ldots, q_n \mathrel{*}= S \\
 & \big| & \textbf{skip} \;\big|\; P; Q \\
 & \big| & \textbf{if}\ b\ \textbf{then}\ P\ \textbf{else}\ Q \;\big|\; \textbf{measure}\ q\ \textbf{then}\ P\ \textbf{else}\ Q \;\big|\; \textbf{while}\ b\ \textbf{do}\ P \\
 & \big| & \textbf{proc}\ X : \Gamma \to \Gamma'\ \{\ P\ \}\ \textbf{in}\ Q \;\big|\; y_1, \ldots, y_m = X(x_1, \ldots, x_n)
\end{array}
$$

Here, S denotes a built-in unitary transformation of arity n, and Γ, Γ' denote typing contexts. The intended meaning of the basic terms is the same as that of the corresponding atomic flow chart components. $P; Q$ denotes sequential composition of terms, and the **skip** command does nothing. $(\textbf{proc}\ X : \Gamma \to \Gamma'\ \{\ P\ \}\ \textbf{in}\ Q)$ defines a procedure X with body P and scope Q; Γ and Γ' are bindings of the formal parameters for input and output. The term $\bar{y} = X(\bar{x})$ denotes a procedure call. In writing programs, it is common to use certain derived terms, writing for instance $b := c$ as an abbreviation for ($\textbf{if}\ c\ \textbf{then}\ b := \mathbf{0}\ \textbf{else}\ b := \mathbf{1}$), or ($\textbf{if}\ b\ \textbf{then}\ P$) for ($\textbf{if}\ b\ \textbf{then}\ P\ \textbf{else skip}$).

Typing judgments. A *typing judgment* is an expression of the form

$$\Pi \vdash \langle\Gamma\rangle\ P\ \langle\Gamma'\rangle,$$

where Π is a procedure context and Γ, Γ' are typing contexts. The intended meaning is that under the procedure binding Π, P is a well-typed term which transforms a set of variables Γ into a set of variables Γ'. The typing rules are shown in Figure 12.

Note that the typing rules enforce that in the term $(q_1, \ldots, q_n \mathrel{*}= S)$, the variables $q_1, \ldots, q_n$ are *distinct*. Similarly, in a procedure call $\bar{y} = X(\bar{x})$, each of $\bar{x}$ and $\bar{y}$ is a list of distinct variables (although it is possible that $x_i = y_j$). Also note that each term has explicit inputs and outputs; for instance, the term $\bar{y} = X(\bar{x})$ has inputs $\bar{x}$ and outputs $\bar{y}$, whereas the term $b := \mathbf{0}$ has input b and output b. The latter term should be thought of as consuming a variable b, then creating a new one with the same name. A more "functional" way of expressing this would be to write $b = \mathbf{0}(b)$ instead of $b := \mathbf{0}$.

Semantics. The language QPL can be immediately translated into the flow chart language of Section 4. The semantics of a QPL term is simply the semantics of the corresponding flow chart. Alternatively, the semantics can be defined directly by induction on typing judgments; the rules for doing so are obvious and we omit them here. It suffices to say that each typing judgment

$$X_1{:}\bar{s}_1 \to \bar{t}_1, \ldots, X_n{:}\bar{s}_n \to \bar{t}_n \vdash \langle\Gamma\rangle\ P\ \langle\Gamma'\rangle$$

is interpreted as a Scott-continuous function

$$[\![P]\!] : \mathbf{Q}([\![\bar{s}_1]\!], [\![\bar{t}_1]\!]) \times \ldots \times \mathbf{Q}([\![\bar{s}_n]\!], [\![\bar{t}_n]\!]) \to \mathbf{Q}([\![\Gamma]\!], [\![\Gamma']\!]).$$

The language QPL differs from the flow chart language in some minor ways. Specifically, QPL contains the following restrictions:

- branchings and loops must be properly nested,
- merge operations can only occur in the context of a branching or loop,
- each program fragment has a unique incoming and a unique outgoing control path. In particular, procedures have only one entry and exit.

Note that the typing rules for QPL allow procedure definitions to be recursive; however, we have omitted a facility for defining two or more *mutually* recursive procedures. However, this is not a genuine restriction, because mutually recursive procedures can be easily expanded into simply recursive ones. Where desired, one can augment the syntax in the standard way to allow mutual recursion to be expressed directly.

7.2 Block QPL

The language QPL imposes a block structure on control constructs such as **if** and **while**, but not on memory management. Unlike in block-oriented languages such as Pascal, we have allowed variables to be allocated and deallocated anywhere, subject only to the typing rules. This is not unsafe, because the type system is there to ensure that variables are deallocated when they are no longer used. However, allowing non-nested allocations and deallocations carries an implementation cost, because it means that variables must be allocated from a heap rather than a stack. It is therefore interesting to investigate an alternate version of QPL in which a stricter block structure is imposed. We call this alternative language "Block QPL".

Thus, in Block QPL, we want to restrict allocations and deallocations to occur in properly nested pairs. Moreover, this nesting should also respect the nesting of the control constructs **if**, **measure**, and **while**. We thus introduce the (*newbit*) $$\frac{}{\Pi \vdash \langle \Gamma \rangle\ \textbf{new bit}\ b := \mathbf{0}\ \langle b{:}\textbf{bit}, \Gamma \rangle}$$

(*newqbit*) $$\frac{}{\Pi \vdash \langle \Gamma \rangle\ \textbf{new qbit}\ q := \mathbf{0}\ \langle q{:}\textbf{qbit}, \Gamma \rangle}$$

(*discard*) $$\frac{}{\Pi \vdash \langle x{:}t, \Gamma \rangle\ \textbf{discard}\ x\ \langle \Gamma \rangle}$$

(*assign*$_0$) $$\frac{}{\Pi \vdash \langle b{:}\textbf{bit}, \Gamma \rangle\ b := \mathbf{0}\ \langle b{:}\textbf{bit}, \Gamma \rangle}$$

(*assign*$_1$) $$\frac{}{\Pi \vdash \langle b{:}\textbf{bit}, \Gamma \rangle\ b := \mathbf{1}\ \langle b{:}\textbf{bit}, \Gamma \rangle}$$

(*unitary*) $$\frac{S \text{ is of arity } n}{\Pi \vdash \langle q_1{:}\textbf{qbit}, \ldots, q_n{:}\textbf{qbit}, \Gamma \rangle\ \bar{q} \mathrel{*}= S\ \langle q_1{:}\textbf{qbit}, \ldots, q_n{:}\textbf{qbit}, \Gamma \rangle}$$

(*skip*) $$\frac{}{\Pi \vdash \langle \Gamma \rangle\ \textbf{skip}\ \langle \Gamma \rangle}$$

(*compose*) $$\frac{\Pi \vdash \langle \Gamma \rangle\ P\ \langle \Gamma' \rangle \qquad \Pi \vdash \langle \Gamma' \rangle\ Q\ \langle \Gamma'' \rangle}{\Pi \vdash \langle \Gamma \rangle\ P; Q\ \langle \Gamma'' \rangle}$$

(*if*) $$\frac{\Pi \vdash \langle b{:}\textbf{bit}, \Gamma \rangle\ P\ \langle \Gamma' \rangle \qquad \Pi \vdash \langle b{:}\textbf{bit}, \Gamma \rangle\ Q\ \langle \Gamma' \rangle}{\Pi \vdash \langle b{:}\textbf{bit}, \Gamma \rangle\ \textbf{if}\ b\ \textbf{then}\ P\ \textbf{else}\ Q\ \langle \Gamma' \rangle}$$

(*measure*) $$\frac{\Pi \vdash \langle q{:}\textbf{qbit}, \Gamma \rangle\ P\ \langle \Gamma' \rangle \qquad \Pi \vdash \langle q{:}\textbf{qbit}, \Gamma \rangle\ Q\ \langle \Gamma' \rangle}{\Pi \vdash \langle q{:}\textbf{qbit}, \Gamma \rangle\ \textbf{measure}\ q\ \textbf{then}\ P\ \textbf{else}\ Q\ \langle \Gamma' \rangle}$$

(*while*) $$\frac{\Pi \vdash \langle b{:}\textbf{bit}, \Gamma \rangle\ P\ \langle b{:}\textbf{bit}, \Gamma \rangle}{\Pi \vdash \langle b{:}\textbf{bit}, \Gamma \rangle\ \textbf{while}\ b\ \textbf{do}\ P\ \langle b{:}\textbf{bit}, \Gamma \rangle}$$

(*proc*) $$\frac{X{:}\bar{t} \to \bar{s}, \Pi \vdash \langle \bar{x}{:}\bar{t} \rangle\ P\ \langle \bar{y}{:}\bar{s} \rangle \qquad X{:}\bar{t} \to \bar{s}, \Pi \vdash \langle \Gamma \rangle\ Q\ \langle \Gamma' \rangle}{\Pi \vdash \langle \Gamma \rangle\ \textbf{proc}\ X : \bar{x}{:}\bar{t} \to \bar{y}{:}\bar{s}\ \{\ P\ \}\ \textbf{in}\ Q\ \langle \Gamma' \rangle}$$

(*call*) $$\frac{}{X{:}\bar{t} \to \bar{s}, \Pi \vdash \langle \bar{x}{:}\bar{t}, \Gamma \rangle\ \bar{y} = X(\bar{x})\ \langle \bar{y}{:}\bar{s}, \Gamma \rangle}$$

(*permute*) $$\frac{\Pi \vdash \langle \Gamma \rangle\ P\ \langle \Delta \rangle, \qquad \Pi', \Gamma', \Delta' \text{ permutations of } \Pi, \Gamma, \Delta}{\Pi' \vdash \langle \Gamma' \rangle\ P\ \langle \Delta' \rangle}$$

Figure 12: Typing rules for QPL

notion of a *block*, which is a program fragment enclosed in curly brackets $\{\ P\ \}$. The convention is that the scope of any variable declaration extends only to the end of the current block; moreover, the bodies of conditional statements, loops, and procedures are implicitly regarded as blocks in this sense.

In the presence of such a block structure, the explicit **discard** command is no longer needed, so we remove it from the language. Also, we note that with these changes, the incoming and outgoing variables of any procedure must necessarily be the same. Thus, we also modify the syntax of procedure calls, writing **call** $X(x_1,\ldots,x_n)$ instead of $x_1,\ldots,x_n = X(x_1,\ldots,x_n)$. This leaves us with the following syntax for Block QPL:

$$
\begin{array}{lll}
\text{Block QPL Terms} \quad P,Q & ::= & \textbf{new bit}\ b := \mathbf{0} \ \big|\ \textbf{new qbit}\ q := \mathbf{0} \\
 & \big| & b := \mathbf{0} \ \big|\ b := \mathbf{1} \ \big|\ q_1,\ldots,q_n \mathrel{*}= S \\
 & \big| & \textbf{skip} \ \big|\ P;Q \ \big|\ \{\ P\ \} \\
 & \big| & \textbf{if}\ b\ \textbf{then}\ P\ \textbf{else}\ Q \ \big|\ \textbf{measure}\ q\ \textbf{then}\ P\ \textbf{else}\ Q \ \big|\ \textbf{while}\ b\ \textbf{do}\ P \\
 & \big| & \textbf{proc}\ X : \Gamma \to \Gamma\ \{\ P\ \}\ \textbf{in}\ Q \ \big|\ \textbf{call}\ X(x_1,\ldots,x_n)
\end{array}
$$

The typing rules remain unchanged, except for a new rule (*block*), and appropriate changes to the rules for branchings and procedures.

$$
(\textit{block}) \qquad \frac{\Pi \vdash \langle\Gamma\rangle\ P\ \langle\Gamma'\rangle}{\Pi \vdash \langle\Gamma\rangle\ \{\ P\ \}\ \langle\Gamma\rangle}
$$

$$
(\textit{if}) \qquad \frac{\Pi \vdash \langle b{:}\mathbf{bit},\Gamma\rangle\ P\ \langle\Gamma'\rangle \qquad \Pi \vdash \langle b{:}\mathbf{bit},\Gamma\rangle\ Q\ \langle\Gamma''\rangle}{\Pi \vdash \langle b{:}\mathbf{bit},\Gamma\rangle\ \textbf{if}\ b\ \textbf{then}\ P\ \textbf{else}\ Q\ \langle b{:}\mathbf{bit},\Gamma\rangle}
$$

$$
(\textit{measure}) \qquad \frac{\Pi \vdash \langle q{:}\mathbf{qbit},\Gamma\rangle\ P\ \langle\Gamma'\rangle \qquad \Pi \vdash \langle q{:}\mathbf{qbit},\Gamma\rangle\ Q\ \langle\Gamma''\rangle}{\Pi \vdash \langle q{:}\mathbf{qbit},\Gamma\rangle\ \textbf{measure}\ q\ \textbf{then}\ P\ \textbf{else}\ Q\ \langle b{:}\mathbf{bit},\Gamma\rangle}
$$

$$
(\textit{proc}) \qquad \frac{X{:}\bar{t} \to \bar{t},\Pi \vdash \langle\bar{x}{:}\bar{t}\rangle\ P\ \langle\Gamma'\rangle \qquad X{:}\bar{t} \to \bar{t},\Pi \vdash \langle\Gamma\rangle\ Q\ \langle\Gamma''\rangle}{\Pi \vdash \langle\Gamma\rangle\ \textbf{proc}\ X : \bar{x}{:}\bar{t} \to \bar{x}{:}\bar{t}\ \{\ P\ \}\ \textbf{in}\ Q\ \langle\Gamma\rangle}
$$

$$
(\textit{call}) \qquad \frac{}{X{:}\bar{t} \to \bar{t},\Pi \vdash \langle\bar{x}{:}\bar{t},\Gamma\rangle\ \textbf{call}\ X(\bar{x})\ \langle\bar{x}{:}\bar{t},\Gamma\rangle}
$$

Note that, for any valid typing judgment $\Pi \vdash \langle\Gamma\rangle\ P\ \langle\Gamma'\rangle$, we necessarily have $\Gamma \subseteq \Gamma'$; thus the rule (*block*) has a similar effect as the QPL rule (*discard*).

The advantage of having a strict block structure as in Block QPL is that allocation follows a stack discipline, thus potentially simplifying implementations. However, there seems to be little added benefit for the programmer, particularly since the QPL type system already prevents memory leaks. In fact, the restrictions of Block QPL seem to encourage an unnatural programming style; on balance, it is probably not worth having these restrictions.

7.3 Extensions of the type system

So far, the only data types we have considered are **bit** and **qbit**, because this is the bare minimum needed to discuss the interaction of quantum data, classical data, and classical control. In practice, such a finitary type system is much too restrictive; for instance, the full power of loops and recursion does not manifest itself unless programs can operate on variable size data. In this section, we briefly discuss how the QPL type system can be extended with more complex types, and particularly infinitary types. It is remarkable that these extensions work seamlessly, even when mixing classical and quantum types in a data structure. In discussing possible extensions to the type system, we keep in mind the semantic framework of Section 6, as well as the potential physical realizability of the resulting types.

Tuples. We extend the type system of QPL adding a new type $t_1 \otimes \ldots \otimes t_n$, whenever $t_1, \ldots, t_n$ are types. We introduce statements for constructing and deconstructing tuples:

$$(tuple) \quad \frac{}{\Pi \vdash \langle x_1{:}t_1, \ldots, x_n{:}t_n, \Gamma\rangle \; x = (x_1, \ldots, x_n) \; \langle x : t_1 \otimes \ldots \otimes t_n, \Gamma\rangle}$$

$$(untuple) \quad \frac{}{\Pi \vdash \langle x : t_1 \otimes \ldots \otimes t_n, \Gamma\rangle \; (x_1, \ldots, x_n) = x \; \langle x_1{:}t_1, \ldots, x_n{:}t_n, \Gamma\rangle}$$

The semantics of such tuples can be given in the framework of Section 6 without any changes; one simply adds an equation $[\![t \otimes s]\!] = [\![t]\!] \otimes [\![s]\!]$, and interprets the basic tupling and untupling operations as the semantic identity map.

Tuples can be used to encode fixed-length classical or quantum integers. For instance, the type of 4-bit classical integers is defined as $\mathbf{int}_4 = \mathbf{bit} \otimes \mathbf{bit} \otimes \mathbf{bit} \otimes \mathbf{bit}$. If desired, one may add appropriate built-in functions to facilitate the manipulation of such integers.

Sums. We further extend the type system by introducing a sum type $t_1 \oplus \ldots \oplus t_n$, whenever $t_1, \ldots, t_n$ are types. A sum type expresses a choice between several alternatives. Note that the selection of alternatives is *classical*; for instance, the type of classical booleans is definable as $\mathbf{bit} = \mathbf{I} \oplus \mathbf{I}$. Elements of sum types are constructed and deconstructed via injection and case statements, much as in other functional programming languages:

$$(inj) \quad \frac{}{\Pi \vdash \langle x{:}t_i, \Gamma\rangle \; y = \mathrm{in}_i \, x : t_1 \oplus \ldots \oplus t_n \; \langle y : t_1 \oplus \ldots \oplus t_n, \Gamma\rangle}$$

$$(case) \quad \frac{\Pi \vdash \langle x_1{:}t_1, \Gamma\rangle \; P_1 \; \langle \Gamma'\rangle \qquad \ldots \qquad \Pi \vdash \langle x_n{:}t_n, \Gamma\rangle \; P_n \; \langle \Gamma'\rangle}{\Pi \vdash \langle y : t_1 \oplus \ldots \oplus t_n, \Gamma\rangle \; \mathbf{case} \; y \; \mathbf{of} \; \mathrm{in}_1 \, x_1 \Rightarrow P_1 \mid \ldots \mid \mathrm{in}_n \, x_n \Rightarrow P_n \; \langle \Gamma'\rangle}$$

Note that by adding sum types to the language QPL, it is possible to encode procedures with multiple points of entry and exit.

Infinite types. The semantics of Section 6 cannot directly handle infinite types, since it is based on finite dimensional vector spaces. However, it is not difficult to adapt the semantics to the infinite-dimensional case. This allows us, for instance, to accommodate an infinite type of classical integers, which is defined as the countable sum $\mathbf{int} = \mathbf{I} \oplus \mathbf{I} \oplus \ldots$. The semantics of infinite types is based on positive linear operators of bounded trace; details will be given elsewhere.

Perhaps more controversial than infinite sums are infinite tensor products. For instance, a naive implementation of "arbitrary size quantum integers" would be as the infinite tensor $\mathbf{qbit} \otimes \mathbf{qbit} \otimes \ldots$. While infinite tensor products create no particular problem from the point of view of denotational semantics, a sensible implementation can only use finitely many quantum bits at any given time. This can be achieved by imposing a semantic "zero tail state" condition, which means that only finitely many non-zero bits are allowed to occur at any given time in the computation. The compiler or the operating system has to implement a mechanism by which the zero tail state condition is enforced. This requires some overhead, but might be a useful abstraction for programmers.

Structured types. A particularly useful class of infinite sum types is the class of structured recursive types, such as lists, trees, etc. For example, the type of lists of quantum bits can be defined recursively as $L := \mathbf{I} \oplus (\mathbf{qbit} \otimes L)$. Note that, because the use of the $\oplus$ operator implies a classical choice, quantum data occurs only "at the leaves" of a structured type, while the structure itself is classical. Thus, the length of a list of quantum bits is a classically known quantity. This view of structured quantum types fits well with our paradigm of "classical control".

Lists of quantum bits are a good candidate for an implementation of a type of "variable-size quantum integers". With this implementation, each quantum integer has a classically determined size. One can thus write programs which operate on quantum integers of arbitrary, but known, size. This seems to be precisely what is required by many currently known number-theoretic quantum algorithms, such as Shor's factoring algorithm or the Quantum Fourier Transform. Moreover, representing quantum integers as lists, rather than as arrays as is usually done [9, 2], means that no out-of-bounds checks or distinctness checks are necessary at run-time; the syntax automatically quarantees

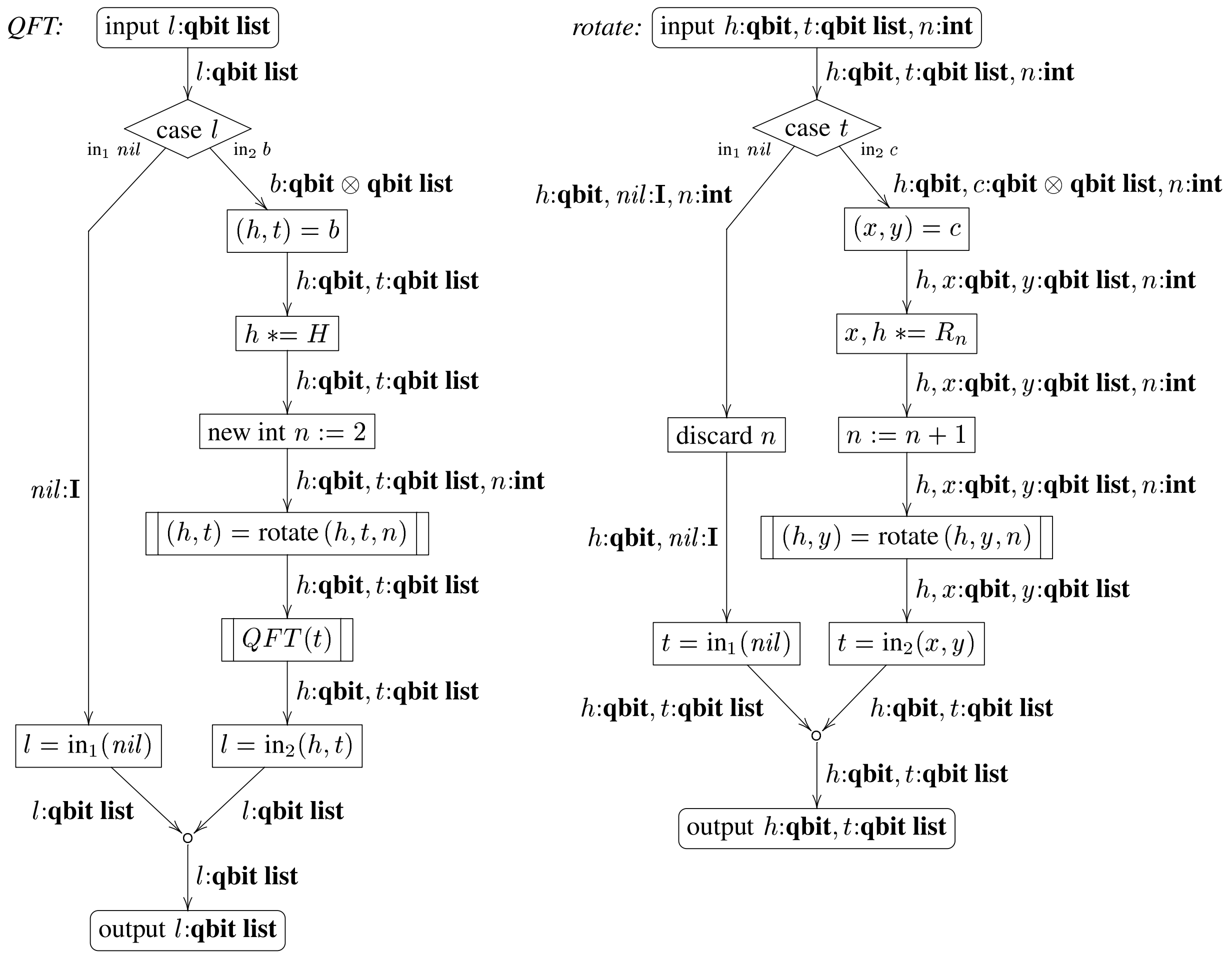


Figure 13: The Quantum Fourier Transform

that distinct identifiers refer to distinct objects at run-time. An example of a quantum algorithm using lists is given in Section 7.4.

On a more speculative note, one might ask whether it is possible to have structured types whose very structure is "quantum" (for instance, a quantum superposition of a list of length 1 and a list of length 2). Such types do not readily fit into our "classical control" paradigm. It is an interesting question whether there is a physically feasible representation of such types, and whether they can be manipulated in an algorithmically useful way.

Higher-order types Unlike typical functional programming languages, the language QPL does not presently incoporate any higher-order features. There is currently no mechanism for abstracting procedures and considering them as data to be manipulated. It is an interesting question whether it is possible to augment the language with functional closures in the style of a typed linear lambda calculus. Possible semantic models of such a language might be drawn from the recent work of Abramsky and Coecke [1], or from the theory of "quantum coherence spaces" recently defined by Girard [4].

7.4 Example: The Quantum Fourier Transform.

We given an example of the use of recursive types. Let **qbit list** be the type of lists of quantum bits, defined by the recursive equation **qbit list** $:= \mathbf{I} \oplus ($**qbit** $\otimes$ (**qbit list**)). Figure 13 shows an implementation of the Quantum Fourier Transform (QFT) [13, 10], which is of type **qbit list** $\rightarrow$ **qbit list**. The algorithm shown differs from the standard

Quantum Fourier Transform in that we have omitted the final step which reverses the order of the bits in the output. Note that the procedure *QFT* uses recursion to traverse its input list; it also uses an auxiliary procedure *rotate* which is recursive in its own right. For simplicity, we have augmented the language by a classical integer type **int** with built-in addition, and we have added an obvious **case** statement of type $A \oplus B \to A; B$. We also use the Hadamard operator H, as well as a parameterized family of unitary operators R_n, which are defined by

$$R_n = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{2\pi i/2^n} \end{pmatrix}.$$

References

[1] S. Abramsky and B. Coecke. Physical traces: Quantum vs. classical information processing. arXiv:cs.CG/0207057 v1, July 2002.

[2] S. Bettelli, T. Calarco, and L. Serafini. Toward an architecture for quantum programming. arXiv:cs.PL/0103009 v2, Nov. 2001.

[3] R. Cleve. An introduction to quantum complexity theory. In C. Macchiavello, G. Palma, and A. Zeilinger, editors, *Collected Papers on Quantum Computation and Quantum Information Theory*, pages 103–127. World Scientific, 2000.

[4] J.-Y. Girard. Entre logique et quantique. Manuscript, Institut de Mathématiques de Luminy, Marseille, Oct. 2002.

[5] J. Gruska. *Quantum Computing*. McGraw Hill, 1999.

[6] M. Hasegawa. *Models of Sharing Graphs: A Categorical Semantics of let and letrec*. PhD thesis, Department of Computer Science, University of Edinburgh, July 1997.

[7] A. Joyal, R. Street, and D. Verity. Traced monoidal categories. *Mathematical Proceedings of the Cambridge Philosophical Society*, 119:447–468, 1996.

[8] E. H. Knill. Conventions for quantum pseudocode. LANL report LAUR-96-2724, 1996.

[9] B. Ömer. A procedural formalism for quantum computing. Master's thesis, Department of Theoretical Physics, Technical University of Vienna, July 1998. http://tph.tuwien.ac.at/˜oemer/qcl.html.

[10] J. Preskill. Quantum information and computation. Lecture Notes for Physics 229, California Institute of Technology, 1998.

[11] J. W. Sanders and P. Zuliani. Quantum programming. In *Mathematics of Program Construction*, Springer LNCS 1837, pages 80–99. 2000.

[12] P. Selinger. Categorical structure of asynchrony. In *Proceedings of MFPS 15, New Orleans*, Electronic Notes in Theoretical Computer Science 20, 1999.

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