Quantum Arrows in Haskell

Juliana Kaizer Vizzotto[1] Antônio Carlos da Rocha Costa[1]

Federal University of Rio Grande do Sul, Brazil
Catholic University of Pelotas, Brazil

Amr Sabry[2]

Indiana University, USA

Abstract

We argue that a realistic model for quantum computations should be *general* with respect to measurements, and *complete* with respect to the information flow between the quantum and classical worlds. We discuss two alternative models for general and complete quantum computations based on probability distributions of quantum state vectors and on density matrices with classical outputs. We show that both models can be structured using a generalization of monads called arrows.

Keywords: quantum programming, density matrices, probabilities

1 Introduction

In recent work [13] we established that a *general* model of quantum computing (including measurements), based on density matrices and superoperators, is an instance of a generalization of monads called *arrows* [5]. That work is strictly based on quantum data (any classical value must be represented as quantum). The model cannot express the passage of information between the classical and quantum worlds.

However, various quantum algorithms are explained in terms of the *interchanging* of quantum and classical information [3]. For example, quantum teleportation is a traditional algorithm which is based on two quantum processes communicating via *classical data*. There is interest to consider *measurements* and the *information flow* between quantum and classical processes as essential components of quantum computations (for instance, see [11,6,9,4,12]).

[1] Email: {jkv,rocha}@atlas.ucpel.tche.br
[2] Email: sabry@indiana.edu
[3] By interchanging we mean, for instance, a measurement in the middle of the computation.

On the other hand, the finding of a representation that is suitable for representing both the results of unitary transformations and measurement operations should also be put into perspective, in order to uniformly fit with the requirements of generality and completeness.

That is, we would like that the same representational framework be able to take care of both: (1) the task of representing the *quantum state* resulting from a unitary operation applied to a given quantum state, and (2) the task of representing the pair of information coming out from a measurement, namely: (2a) that corresponding to the *measure* produced by the measurement (one of the eigen-values of the measurement operator), and (2b) the *quantum state* that results from the projection imposed on the original quantum state by the measurement (one of the eigen-vectors of the measurement operator).

The main problem introduced by the need of that uniformity is that measurement results (both value and state results) are of a probabilistic kind, needing *sets of possible results* for their representation. The usual alternative solution to such problem is the density matrix formalism.

However, there is a (possibly not minor) conceptual problem in the adoption of the density matrix formalism, namely: a density matrix is supposed to represent a set (*ensemble*) of quantum systems whose probability distribution of states the density matrix represents; however, from a programming theoretic point of view, one usually thinks of a quantum algorithm as being performed by one single quantum system, not an ensemble of quantum systems each possibly behaving in a different way according to a probability distribution.

We feel that the quantum programmer's intuition of programming one single quantum system at a time, while elaborating his algorithms, may happen to be not appropriately captured by the density matrix formalism. We feel (but we have no definite argument) that a representation modelled on the usual set-theoretic representation of states of non-deterministic machines, adjusted to explicitly represent the probability of occurrence of each deterministic state, may happen to capture in a better way the quantum programmer's intuition.

So, in the paper, we introduce two ways to deal with *combined* quantum and classical computations, which are based on different ways of representing states which result from measurements, one based on density matrices, the other based on explicit probability distributions over sets of quantum states.

The paper is organised as follows. In Section 2 we introduce *indexed monads* and *indexed arrows* in the context of Haskell. Section 3 briefly reviews our previous work [13] on modelling superoperators as arrows. We then show two alternative general and complete models for combined quantum and classical computations structured as indexed arrows in Section 4. Section 5 concludes.

2 Indexed Monads and Indexed Arrows

The mathematical concept of monads was introduced to computer science by Moggi [7] in the late 1980's as a way of structuring denotational semantics of programming languages. Several different language features, including nontermination, state, exceptions, continuations, and interaction can be viewed as monads. More recently, this construction has been internalised in the programming language Haskell as a tool to elegantly express computational effects within the context of a pure functional language.

Since the work of Moggi, several natural notions of computational effects were discovered which could only be expressed as generalisations of monads. Of particular importance to us is the generalisation of monads known as arrows [5] which is also internalised in the programming language Haskell. In this section, we briefly discuss a small variation of these notions in the context of the programming language Haskell, which we call *indexed* monads and *indexed* arrows. Those are the right notions needed to structure quantum computations.

2.1 Indexed Monads

A monad is used for formulating definitions and structuring *notions of computations* (possibly non-functional) in programming languages. In this context, a *program*, which features notions of computations, can be viewed as a *function from values to computations*. For instance a program with exceptions can be viewed as a function that takes a value and returns a *computation* that may succeed or may fail.

More precisely, to understand monadic computations one can consider a value category $\mathcal{C}$, as a model for functions, and build on top of that, notions of computation via an operator (*endofunctor*) T acting on objects of $\mathcal{C}$ - i.e., T maps an object B from $\mathcal{C}$, viewed as the *set of values of type* τ, to an object TB corresponding to *computations of type* τ. Then a program which takes an input of type A, and after performing a certain computation returns a value of type B, can be identified with a morphism from A to TB in $\mathcal{C}$ [8].

A monad is represented using a type constructor for computations m and two functions:

$$return :: forall\ a.a \rightarrow m\ a$$
$$\gg\!= :: forall\ a\ b.m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$$

The operation $\gg\!=$ (pronounced "bind") specifies how to sequence computations and $return$ specifies how to lift values to computations. Note the requirements of $forall$ in the definitions above. This is because T, as explained above, is an *endofunctor* in $\mathcal{C}$. Then, m is a type constructor acting on *all objects* from the value category.

However, sometimes we want to *select* some objects (sets) from $\mathcal{C}$ to apply the constructor T. This notion is slightly more general than monads, and it is captured by the definition of *Kleisli structure* [2]. Basically, for *indexed monads* (as we prefer to call Kleisli structures), the function T does not need be an endofunctor on $\mathcal{C}$. We can select some objects from $\mathcal{C}$ to apply the constructor. This is exactly the notion we need to model quantum state vectors [4] as monads. The constructor for a quantum vector can only act over the types which constitute a basis.

Now, the definitions of $return$ and $\gg\!=$ in Haskell should be rephrased as:

$$return :: forall\ a.F\ a \Rightarrow a \rightarrow m\ a$$
$$\gg\!= :: forall\ a\ b.F\ a, F\ b \Rightarrow m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$$

That is, for all a for which $F\ a$ holds we can apply the constructor m, and for all a and b for which $F\ a$ and $F\ b$ hold we can apply $\gg\!=$. Moreover, to construe a proper monad or *indexed* monad, the $return$ and $\gg\!=$ functions must work together according to the three monad laws [7].

2.2 Indexed Arrows

To handle situations where monads are inapplicable, Hughes [5] introduced a new abstraction generalising monads, called *arrows*. Indeed, in addition to defining a notion of pro-

[4] That is, a function which associates each basis element with a complex probability amplitude.

cedure which may perform computational effects, arrows may have a static component, or may accept more than one input.

Just as we think of a monadic type $m\ \ a$ as representing a *computation* delivering an a, so we think of an arrow type $a\ \ b\ \ c$ as representing a computation with input of type b delivering a c. Arrows make the dependence on input explicit.

$$\begin{array}{ll} arr & :: forall\ b\ c.(b \to c) \to a\ b\ c \\ (\ggg) & :: forall\ b\ c\ d.a\ b\ c \to a\ c\ d \to a\ b\ d \\ first & :: forall\ b\ c\ d.a\ b\ c \to a\ (b,d)\ (c,d) \end{array}$$

In other words, to be an arrow, a type a must support the three operations arr, $\ggg$, and *first* with the given types. The function *arr* allows us to lift "pure" functions to computations. The function $\ggg$ composes two computations. The function *first* allows us to apply an arrow in the context of other data.

Observe the requirements of *forall* in the definitions. They mean that we can build computations on top of *all* value functions. However, as with monads, we want to *select* some specific value functions. This is the case for quantum functions: we want to lift simple functions acting on the *basis* elements to functions acting on vectors over those basis. Hence we define *indexed arrows* [5]:

$$arr :: (I\ b, I\ c) \Rightarrow (b \to c) \to a\ b\ c$$
$$(\ggg) :: (I\ b, I\ c, I\ d) \Rightarrow a\ b\ c \to a\ c\ d \to a\ b\ d$$
$$first :: (I\ b, I\ c, I\ d) \Rightarrow a\ b\ c \to a\ (b,d)\ (c,d)$$

The operations for arrows or *indexed* arrows must satisfy the arrow laws [5], such that these operations are well-defined even with arbitrary permutations and change of associativity.

3 Review: Quantum Vectors as Indexed Monads and Superoperators as Indexed Arrows

3.1 Vectors as Indexed Monads

In this section we quickly review the work presented in [13]. Given a set a representing observable (classical) values, i.e. a *basis* set, a pure quantum state is a vector $a \to \mathbb{C}$ which associates each basis element with a complex probability amplitude. In Haskell, a finite set a can be represented as an instance of the class $Basis$, shown below, in which the constructor $basis :: [a]$ explicitly lists the basis elements. The basis elements must be distinguishable from each other, which explains the constraint $Eq\ \ a$ on the type of elements:

$$\mathbf{class}\ Eq\ a \Rightarrow Basis\ a\ \mathbf{where}\ basis :: [a]$$
$$\mathbf{type}\ K = Complex\ Double$$
$$\mathbf{type}\ Vec\ a = a \to K$$

The type K (notation from the base field) is the type of probability amplitudes.

The monadic functions for vectors are defined as:

$$return :: Basis\ a \Rightarrow a \to Vec\ a$$
$$return\ a\ b = \mathbf{if}\ a \equiv b\ \mathbf{then}\ 1.0\ \mathbf{else}\ 0.0$$
$$(\ggg=) :: (Basis\ a, Basis\ b) \Rightarrow Vec\ a \to (a \to Vec\ b) \to Vec\ b$$

[5] Categorically, the definition of arrows is captured by Freyd-categories [10]. *Indexed arrows* are *indexed* Freyd-categories.

$va \gg= f = \lambda b \to sum\ [(va\ \ a) * (f\ \ a\ \ b) \mid a \leftarrow basis]$

$return$ just lifts values to vectors, and $bind$, given a *unitary operator* (i.e., *unitary operator*) represented as a function $a \to Vec\ b$, and given a $Vec\ a$, returns a $Vec\ b$ (that is, it specifies how a $Vec\ a$ can be turned in a $Vec\ b$).

Proposition 3.1 *The indexed monad Vec satisfies the required equations for monads.*

Examples of vectors over the set of booleans may be defined as follows:

$\mathbf{instance}\ Basis\ Bool\ \mathbf{where}$
$\quad basis = [False, True]$

$qFalse, qTrue, qFT, qFmT :: Vec\ Bool$
$qFalse = return\ False$
$qTrue = return\ True$
$qFT = (1\ /\ \sqrt{2})\ \$*\ (qFalse\ `mplus`\ qTrue)$
$qFmT = (1\ /\ \sqrt{2})\ \$*\ (qFalse\ `mminus`\ qTrue)$

The first two are unit vectors corresponding to basis elements; the last two represent states which are in equal superpositions of $False$ and $True$. In the Dirac notation, these vectors would be respectively written as $|False\rangle$, $|True\rangle$, $\frac{1}{\sqrt{2}}(|False\rangle + |True\rangle)$, and $\frac{1}{\sqrt{2}}(|False\rangle - |True\rangle)$. The operations $\$*$, $`mplus'$, and $`mminus'$ are the usual scalar product, sum and subtraction of vectors, respectively.

Unitary operations can also be defined directly, for example:

$\mathbf{type}\ Uni\ \ a\ \ b = a \to Vec\ b$

$hadamard :: Uni\ Bool\ Bool$
$hadamard\ \ False = qFT$
$hadamard\ \ True = qFmT$

$zgate :: Uni\ Bool\ Bool$
$zgate\ \ False = qFalse$
$zgate\ \ True = -1\ \$*\ qTrue$

3.2 Superoperators as Indexed Arrows

Intuitively, density matrices can be understood as a statistical perspective of the state vector. In the density matrix formalism, a quantum state that used to be modelled by a vector v is now transformed in a matrix in such a way that the *amplitudes of the state vector turn into a kind of probability distributions over state vectors*.

$\mathbf{type}\ Dens\ \ b = Vec\ (b, b)$

Operations mapping density matrices to density matrices are called *superoperators*:

$\mathbf{type}\ Super\ \ b\ \ c = (b, b) \to Dens\ \ c$

We represent a superoperator mirroring a big matrix, so mapping values to density matrices (that is, $Super\ \ b\ \ c \equiv (b, b) \to (c, c) \to K$).

Just as the probability effect associated with vectors is a modelled by a *indexed monad* because of the $Basis$ constraint, the type $Super$ is modelled by a *indexed arrow* because the types include the additional constraint requiring the elements to form a set of basis values (the definition for arr, $\ggg$, and $first$ for $Super$ are in [13]).

Using this *general* model of quantum computations structured as arrows we can elegantly express quantum computations involving measurements. However, that work is strictly based on quantum data, we can not express algorithms with combined interactions of quantum and classical operations directly. Yet as noted in [4,12] a *complete* model for expressing quantum algorithms should accommodate both measurements and combined interactions of quantum and classical data.

4 Quantum Programs as Indexed Arrows

Based on the idea that fully expressible languages/models for quantum computation are supposed to include more than one final measurement operation, that is, they should accommodate both measurements and combined interactions of quantum and *classical data*, in this section we structure two alternative general (involving measurements) and complete (involving both quantum and classical data) approaches for *combined* quantum and classical computations as indexed arrows. The first one is based on a measurement approach for quantum programs. Basically, at each step (a part) of the density operator representing the global quantum state is measured, a *perspective* on the classical measurement results is returned, and the state is left in a new density operator. The second one is based on probability distributions over quantum state vectors. Attached to each vector in the distribution there is a list of classical values - the eigenvalues which are the output of performed measurements. Essentially, the idea is to model the behavior of a single, well determined quantum system, where at any time one can express the information flow from the quantum to the classical level, and vice-versa.

4.1 Programs with Density Operators and Classical Outputs

We present a model for quantum computations based on a measurement approach. The idea is to have a density operator representing the global quantum state, and a probability distribution of classical values representing the classical part of the state. A quantum program acting on this state is interpreted by a special *tracing superoperator*, which in the general case traces out part of the state, returning a classical output, and leaving the system in a new state (possibly in a space with reduced dimension).

4.1.1 Programs with Density Matrices

Because the tracing superoperator in general *forgets* part of the state, we define a relation between bases which we call Dec (from *decomposition*):

$$\mathbf{class}\ (Basis\ a, Basis\ b, Basis\ o) \Rightarrow Dec\ a\ b\ o\ \mathbf{where}$$
$$dec :: [a] \to [(b, o)]$$

specifying that a basis a can be written as (b, o). Then, a quantum program from a to b, parameterised by i, the type of the input classical probability distribution, and o, the part to be measured, is represented by a superoperator from a to b, delivering a classical probability distribution over o.

$$\mathbf{type}\ DProb\ c = [(c, Prob)]$$
$$\mathbf{type}\ QProgram\ i\ o\ a\ b = (DProb\ i, (a, a)) \to (DProb\ o, Dens\ b)$$

Note that our quantum programs should satisfy the restriction $Dec\ a\ b\ o$, and that $DProb\ i$ is used in classical operations or quantum operations controlled by classical data.

As any type can be decomposed by the *unit* $()$, and can be decomposed by itself, and also can be decomposed into one of its parts, we have the following instances:

$$
\begin{array}{l}
\textbf{instance}\ (\mathit{Basis}\ a) \Rightarrow \mathit{Dec}\ a\ a\ ()\ \textbf{where} \\
\quad \mathit{dec}\ [\,] = [\,] \\
\quad \mathit{dec}\ (x : l) = (x, ()) : \mathit{dec}\ l \\
\textbf{instance}\ (\mathit{Basis}\ a) \Rightarrow \mathit{Dec}\ a\ ()\ a\ \textbf{where} \\
\quad \mathit{dec}\ [\,] = [\,] \\
\quad \mathit{dec}\ (x : l) = ((), x) : \mathit{dec}\ l \\
\textbf{instance}\ (\mathit{Basis}\ a, \mathit{Basis}\ b) \Rightarrow \mathit{Dec}\ (a, b)\ a\ b\ \textbf{where} \\
\quad \mathit{dec}\ l = l
\end{array}
$$

Any unitary operator can be lifted to a quantum program which traces out ().

$$
\begin{array}{l}
\mathit{uni2qprog} :: (\mathit{Basis}\ a, \mathit{Basis}\ b, \mathit{Basis}\ i, \mathit{Dec}\ a\ b\ ()) \Rightarrow \\
\qquad\qquad\qquad \mathit{Lin}\ a\ b \rightarrow \mathit{QProgram}\ i\ ()\ a\ b \\
\mathit{uni2qprog}\ f\ (dp, (a_1, a_2)) = \\
\qquad \textbf{let}\ d = \mathit{uni2vec}\ (f\ a_1)\rangle{*}\langle (f\ a_2) \\
\qquad \textbf{in}\ ([\,], d) \\
\qquad\quad \textbf{where}\ \mathit{uni2vec}\ f = [((a, b), f\ \ a\ \ b) \mid (a, b) \leftarrow \mathit{basis}] \\
\qquad\qquad\quad v_1\rangle{*}\langle v_2 = \lambda a_1 \rightarrow [(a_2, v_1\ a_1 * \mathit{conjugate}\ (v_2\ a_2)) \mid \\
\qquad\qquad\qquad\qquad a_2 \leftarrow \mathit{basis}]
\end{array}
$$

The function *uni2qprog* constructs a quantum program, acting on a combined state, from a unitary operator. The idea is to apply the default construction to build a superoperator from a unitary transformation. Note that the classical input is ignored and the classical output is empty: there is no interaction with the classical world when considering unitary transformations. For instance:

$$
\begin{array}{l}
\mathit{hadamardP} :: \mathit{QProgram}\ i\ ()\ \mathit{Bool}\ \mathit{Bool} \\
\mathit{hadamardP} = \mathit{uni2qprog}\ \mathit{hadamard}
\end{array}
$$

lifts the unitary operator *hadamard* to a quantum program acting on a combined state.

Given, a quantum state over a basis set (a, b), the quantum program *trR* forgets the *right* component, returning a new state over b. The subspace is measured before being discharged outputting a classical probability distribution over the basis which forms that subspace. In this case, the input classical data is just ignored.

$$
\begin{array}{l}
\mathit{trR} :: (\mathit{Basis}\ a, \mathit{Basis}\ b, \mathit{Dec}\ (a, b)\ a\ b) \Rightarrow \mathit{QProgram}\ i\ b\ (a, b)\ a \\
\mathit{trR}\ (dp, ((a_1, b_1), (a_2, b_2))) = \textbf{let}\ d = \textbf{if}\ b_1 \equiv b_2\ \textbf{then}\ \mathit{return}\ (a_1, a_2) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ \mathit{vzero} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad p = [(b_1, 1) \mid b_1 \equiv b_2] \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ (p, d) \\
\mathit{trA} :: (\mathit{Basis}\ a, \mathit{Basis}\ i, \mathit{Dec}\ a\ ()\ a) \Rightarrow \mathit{QProgram}\ i\ a\ a\ () \\
\mathit{trA}\ (dp, (a_1, a_2)) = \textbf{let}\ d = \textbf{if}\ a_1 \equiv a_2\ \textbf{then}\ \mathit{return}\ ((), ())\ \textbf{else}\ \mathit{vzero} \\
\qquad\qquad\qquad\qquad\quad p = [(a_1, 1) \mid a_1 \equiv a_2] \\
\qquad\qquad\qquad\quad \textbf{in}\ (p, d)
\end{array}
$$

Similarly, the program *trA* forgets (measures) all quantum state returning only a classical probability distribution as the result. To construe the classical probability distribution we consider that any value from the type being measured *can* appear in the output quantum state. Hence each value from the basis is attached to the probability 1. The real probability to appear in the final state is calculate by the function *app* below, which given a program and a *combined* state calculates the new density matrix and the classical result (if there is some).

$$
\begin{aligned}
& app :: (Basis\ a, Basis\ b, Basis\ i, Basis\ o, Dec\ a\ b\ o) \Rightarrow \\
& \qquad QProgram\ i\ o\ a\ b \rightarrow (DProb\ i, Dens\ a) \rightarrow (DProb\ o, Dens\ b) \\
& app\ p\ (di, da) = \mathbf{let}\ dbf = [(b, sum\ [\mathbf{let}\ (po, db) = p\ (di, a) \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad p_2 = db\ b \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad p_1 = da\ a \\
& \qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{in}\ p_1 * p_2 \mid a \leftarrow basis]) \mid b \leftarrow basis] \\
& \qquad\qquad\qquad\quad po = [(o, p_1 * p_2 * p_3) \mid a \leftarrow basis, (b, o) \leftarrow dec\ [a], \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{let}\ (po, db) = p\ (di, (a, a)), \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{let}\ p_1 = lookup\ o\ po, \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{let}\ p_2 = da\ (a, a), \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{let}\ p_3 = dbf\ (b, b)] \\
& \qquad\qquad\quad \mathbf{in}\ (po, dbf)
\end{aligned}
$$

The output density matrix is calculated by simple matrix multiplication: the superoperator matrix by the input density matrix. Note that the overall operation may depend of the classical state. The probability distribution of classical values is calculated by the multiplication of the probability of the observable *value* in the operator by the probability of the respective observable *value* in the input density matrix. Also we take into account the output density matrix in the calculation as it may be the case that a unitary operation is applied before the measurement.

4.1.2 Programs with Density Matrices as Indexed Arrows

We define the three functions, arr, $\ggg$, and *first*, over $QProgram\ i\ o$ as follows:

$$
\begin{aligned}
& arr :: (Basis\ b, Basis\ c, Dec\ b\ c\ ()) \Rightarrow (b \rightarrow c) \rightarrow QProgram\ i\ ()\ b\ c \\
& arr = uni2qprog.fun2uni \\
& \qquad\qquad \mathbf{where}\ fun2uni\ f = return.f \\
& (\ggg) :: (Basis\ a, Basis\ b, Basis\ c, Basis\ o_1, Basis\ o_2, \\
& \qquad\qquad\qquad\qquad\qquad Dec\ a\ b\ o_1, Dec\ b\ c\ o_2) \Rightarrow \\
& \qquad QProgram\ i\ o_1\ a\ b \rightarrow QProgram\ o_1\ o_2\ b\ c \rightarrow QProgram\ i\ o_2\ a\ c \\
& (f \ggg g)\ (dpi, (a_1, a_2)) = app\ g\ (f\ (dpi, (a_1, a_2))) \\
& first :: (Basis\ a, Basis\ b, Basis\ c, Basis\ o, \\
& \qquad\qquad\qquad\qquad Dec\ a\ b\ a_2, Dec\ (a, c)\ (b, c)\ a_2) \Rightarrow \\
& \qquad\qquad QProgram\ i\ o\ a\ b \rightarrow QProgram\ i\ o\ (a, c)\ (b, c) \\
& first\quad p\ (dpi, ((b_1, d_1), (b_2, d_2))) = \\
& \qquad \mathbf{let}\ (dc, dpo) = p\ (dpi, (b_1, b_2)) \\
& \qquad\quad vdd = return\ (d_1, d_2) \\
& \qquad\quad dbd = [(((b_1, d_{12}), (b_2, d_{22})), db\ (b_1, b_2)\ dc * vdd\ (d_{12}, d_{22})) \mid \\
& \qquad\qquad\qquad\qquad\qquad ((b_1, d_{12}), (b_2, d_{22})) \leftarrow basis] \\
& \qquad \mathbf{in}\ (dpo, dbd)
\end{aligned}
$$

Proposition 4.1 *The indexed arrow* $QProgram\ \ i\ \ o$ *satisfies the required equations for arrows.*

4.2 Programs with Probability Distributions of Quantum Vectors States

The idea is to have a combined state, where the classical part is as before (i.e. a probability distribution of classical values), and the quantum part is represented by a explicit probability distribution over quantum states. A program acting on this combined state can act on the quantum part, on the classical part, or on both parts.

Combined programs acting only on quantum data are of two kinds: i) the unitary transformations, which reversibly transform the state vector and nothing happens to the classical probability; and ii) measurements, which probabilistically yield one of the *eigenvalues* of the observable being measured, and *throws* the system into the correspondent *eigenstate*. Yet one can have quantum operations controlled by classical values as well as purely classical operations.

4.2.1 Programs with Probability Distributions

The probabilistic quantum programming model is based on a data type to represent *explicit probability distributions of quantum state vectors*:

$$\begin{array}{l}\textbf{type } EV = Double \\ \textbf{type } Prob = Double \\ \textbf{newtype } PDQst\ a = PDQ\{ unPDQ :: [([EV], Vec\ a, Prob)]\}\end{array}$$

More specifically, a probability distribution over a basis set a is represented by a pair formed by: a list of real values EV, the eigenvalues which are the outputs of previously performed measurements, and a state vector, $Vec\ a$. We chose to keep a list of eigenvalues EV to maintain a history of measurements. For now this list does not include information about the source of eigenvalues, i.e., about the position of the qubit which was measured in the global state.

The *dynamics* of a quantum system is represented by two kinds of transformations of probability distribution over state vectors:

$$\begin{array}{l}\textbf{data } PDQTrans\ b\ c = Transform\ ((PDQst\ b) \rightarrow (PDQst\ c)) \\ \qquad\qquad\qquad\qquad\quad \mid\ Meas\ ((PDQst\ b) \rightarrow (PDQst\ c))\end{array}$$

We made the difference explicit because the semantics of applying unitary transformations is different from the semantics of applying measurements.

A simple unitary transformation can be defined in such a way that the transformation is applied to all vectors in the distribution. The list of eigenvalues and the probabilities are preserved.

Measurements are the operations which produce eigenvalues as *classical outputs* and return a new classical probability distribution of eigenstates of the observable according to *each* vector in the distribution.

4.2.2 Programs with Probability Distributions as Indexed Arrows

We define the three functions, arr, $\ggg$, and $first$, over $PDQTrans$ as follows:

$$\begin{array}{l}arr :: (Basis\ b, Basis\ c) \Rightarrow (b \rightarrow c) \rightarrow PDQTrans\ b\ c \\ arr\ f = Transform\ (\lambda x \rightarrow PDQ\ [(e_1, v_2, p) \mid (e_1, v_1, p) \leftarrow unPDQ\ x, \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{let } fv = fun2vecfun\ f, \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{let } v_2 = fv\ v_1])\end{array}$$

$$\begin{array}{l}(\ggg) :: (Basis\ b, Basis\ c, Basis\ d) \Rightarrow \\ \qquad PDQTrans\ b\ c \rightarrow PDQTrans\ c\ d \rightarrow PDQTrans\ b\ d \\ (Transform\ f) \ggg (Transform\ g) = Transform\ (\lambda x \rightarrow \textbf{let } d = f\ x \textbf{ in } g\ d) \\ (Meas\ f) \ggg (Transform\ g) = Transform\ (\lambda x \rightarrow \textbf{let } d = f\ x \textbf{ in } g\ d) \\ (Transform\ f) \ggg (Meas\ g) = Meas\ (\lambda x \rightarrow \textbf{let } d = f\ x \textbf{ in } g\ d)\end{array}$$

```
(Meas f) ⋙ (Meas g) = Meas (λx → let d = f x in g d)

first :: (Basis b, Basis c, Basis d) ⇒
                          PDQTrans b c → PDQTrans (b, d) (c, d)
first (Transform f) =
Transform (λx → let fg = getvbs (Transform f)
                    fext = helper_first fg
                in PDQ [(le, v, p) | (le1, v1, p1) ← unPDQ x,
                         let (le, v, p) =   (le1, [((c, d), k1 ∗ k2) |
                         ((b, d), k1) ← v1, let d2 = fext (b, d),
                         (le2, v2, p2) ← unPDQ d2,
                         ((c, d), k2) ← v2], p1)])
first (Meas f) =
Meas (λx → let   fg = getvbs (Meas f)
                 fext = helper_first fg
           in zipqd (PDQ [(le, v, p) | (le1, v1, p1) ← unPDQ x,
                   ((b, d), k1) ← v1, let d2 = fext (b, d),
                   (le2, v2, p2) ← unPDQ d2,
                   let (le, v) = (le2 ++ le1,
                             [((c, d), k1 ∗ k2) | ((c, d), k2) ← v2]),
                   let p = p1 ∗ p2 ∗ (((∗∗2).magnitude) k1)]))
```

The first two functions are straightforward: *arr* constructs a *reversible* transformation from a basic function, *fun2vecfun* converts a "matrix" to a function mapping vectors to vectors, and ⋙ just composes two *PDQTrans*. The function *first* is a bit more subtle, the idea is to transform a function which acts in *part* of a quantum state (say $Vec\ b$) to a function which acts in the *global* state (say $Vec\ (b, d)$). The implementation is based in the following two functions:

```
getvbs :: PDQTrans a b → (a → PDQst b)
getvbs (Transform f) = λa → let d = dreturn a in f d
getvbs (Meas f) = λa → let d = dreturn a in f d

helper_first :: (a → PDQst b) → (a, c) → PDQst (b, c)
helper_firstl f (a, c) =   let   db = f a
                                 dc = dreturn c
                           in PDQ [(le, v2, p ∗ q) |
                              (le, vb, p) ← unPDQ db,
                              (_, vc, q) ← unPDQ dc,
                                 let v2 = [((b, c),
                                         vb b ∗ vc c) | (b, c) ← basis]]
```

Given a *PDQTrans*, *getvbs* determines how that behaves for basic vectors. Then, given the basis' elements, *helper_first* extends the transformation. Essentially, what *first* does is to calculate the *extended* function for the input *PDQTrans* using *firstbs*, and then to calculate the output, correctly applying the extended *PDQTrans* to the input probability distribution of state vectors. The trick for *first* is that we have made an explicit difference between measurements and unitary transformations. If the input function is *not* a measurement the calculation is standard, but if that *is* a measurement then the number of states vectors in the distribution is augmented and we need to use the function *zipqd*, which combines all state vectors that are tagged with the same eigenvalue.

Proposition 4.2 *The indexed arrow $PDQTrans$ satisfies the required equations for arrows.*

4.3 Example: Quantum Teleportation

Quantum teleportation [3] is one of the most traditional examples of quantum algorithms which require the interchanging between quantum and classical data. T hat enables the transmission, *using a classical communication channel*, of an unknown quantum state via a previously shared epr pair. In this section we faithfully express the teleportation algorithm using the two models presented above.

In this section we use Paterson (2001)'s *arrow notation*. Arrow notation is an extension to Haskell with an improved syntax for writing computations using arrows. Here is a simple example to illustrate the notation:

$$
\begin{aligned}
&e_1 :: \mathit{Super}\ (\mathit{Bool}, a)\ (\mathit{Bool}, a)\\
&e_1 = \mathit{proc}\ (a, b) \rightarrow \mathbf{do}\\
&\qquad\qquad r \leftarrow \mathit{lin2super}\ \mathit{hadamard} \prec a\\
&\qquad\qquad \mathit{returnA} \prec (r, b)
\end{aligned}
$$

The $\mathbf{do}$-notation simply sequences the actions in its body. The function $returnA$ is the equivalent for arrows of the monadic function $return$. The two additional keywords are:

- the *arrow abstraction* $proc$ which constructs an arrow instead of a regular function.
- the *arrow application* $\prec$ which feeds the value of an expression into an arrow.

4.3.1 Teleportation with Density Matrices and Classical Outputs

The main procedure receives no classical data and three entangled qubits; then passes a qubit of the epr pair and the qubit to be teleported to Alice, which realizes some quantum operations and measures its two qubits, returning only classical values to the main procedure, which will be communicated to Bob.

$$
\begin{aligned}
&\mathit{teleportation} :: \mathit{QProgram}\ ()\ ()\ (\mathit{Bool}, \mathit{Bool}, \mathit{Bool})\ \mathit{Bool}\\
&\mathit{teleportation} = \mathit{proc}\ (\mathit{eprL}, \mathit{eprR}, q) \rightarrow \mathbf{do}\\
&\qquad\qquad\qquad\qquad \mathit{cs} \leftarrow \mathit{alice} \prec (\mathit{eprL}, q)\\
&\qquad\qquad\qquad\qquad q' \leftarrow \mathit{bob} \prec (\mathit{eprR}, \mathit{cs})\\
&\qquad\qquad\qquad\qquad \mathit{returnA} \prec q'\\
&\mathit{alice} :: \mathit{QProgram}\ ()\ (\mathit{Bool}, \mathit{Bool})\ (\mathit{Bool}, \mathit{Bool})\ ()\\
&\mathit{alice} = \mathit{proc}\ (\mathit{eprL}, q) \rightarrow \mathbf{do}\\
&\qquad\qquad (q_1, e_1) \leftarrow \mathit{qcnotP} \prec (q, \mathit{eprL})\\
&\qquad\qquad q_2 \leftarrow \mathit{hadamardP} \prec q_1\\
&\qquad\qquad \mathit{cs} \leftarrow \mathit{trA} \prec (e_1, q_2)\\
&\qquad\qquad \mathit{returnA} \prec \mathit{cs}
\end{aligned}
$$

where $qcnotP$ is the program:

$$
\begin{aligned}
&\mathit{qcnotP} :: \mathit{QProgram}\ ()\ ()\ (\mathit{Bool}, \mathit{Bool})\ (\mathit{Bool}, \mathit{Bool})\\
&\mathit{qcnotP} = \mathit{uni2qprog}\ (\mathit{controlled}\ \mathit{arr}\ \neg)\\
&\qquad\qquad \mathbf{where}\ \mathit{controlled}\ f\ (b, a) = (\mathit{return}\ b)\langle * \rangle\\
&\qquad\qquad\qquad\qquad\qquad\qquad (\mathbf{if}\ b\ \mathbf{then}\ f\ a\ \mathbf{else}\ \mathit{return}\ a)
\end{aligned}
$$

which acts as a controlled quantum not over the quantum data.

Bob is a procedure which receives a classical data over $(Bool, Bool)$ and a qubit. The procedure analyses the classical data and depending on its value applies or not a certain quantum operation to the input qubit.

$$
\begin{array}{l}
bob :: QProgram\ (Bool, Bool)\ ()\ ((), Bool)\ Bool \\
bob = \lambda(pbb, db) \rightarrow \textbf{let}\ (p_1, d_1) = \textbf{if}\ (lookup\ True\ (unzipL\ pbb)\ pbb > 0) \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{then}\ (qnotP\ ([((), 1)], db)) \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ ([((), 1)], vreturn\ db) \\
\qquad\qquad\qquad\qquad\quad (p_2, d_2) = \textbf{if}\ (lookup\ True\ (unzipR\ pbb) > 0) \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{then}\ (zgateP\ ([((), 1)], db)) \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ st1 \\
\qquad\qquad\qquad\quad \textbf{in}\ (p_2, d_2)
\end{array}
$$

Again we are using a program version of a unitary operation:

$$
\begin{array}{l}
zgateP :: QProgram\ ()\ ()\ Bool\ Bool \\
zgateP = uni2prog\ zgate
\end{array}
$$

The functions $unzipL$ and $unzipR$ take a list of tuples and return a list with the left elements of the tuples and a list with the right elements of the tuples, respectively.

$$
\begin{array}{l}
unzipL :: [((a, b), p)] \rightarrow [a] \\
unzipL\ l = \textbf{let}\ (lb, lp) = unzip\ l \\
\qquad\qquad\qquad (las, lbs) = unzip\ lb \\
\qquad\qquad \textbf{in}\ las \\
unzipR :: [((a, b), p)] \rightarrow [b] \\
unzipR\ l = \textbf{let}\ (lb, lp) = unzip\ l \\
\qquad\qquad\qquad (las, lbs) = unzip\ lb \\
\qquad\qquad \textbf{in}\ lbs
\end{array}
$$

4.3.2 Teleportation with Probability Distribution of Quantum State Vectors

Using a reasoning as above, we model the algorithm for teleportation using explicit probability distributions as arrows:

$$
\begin{array}{l}
alice :: PDQTrans\ (Bool, Bool)\ () \\
alice = proc\ (eprL, q) \rightarrow \textbf{do} \\
\qquad\qquad (q_1, e_1) \leftarrow controlled\_notD \prec (q, eprL) \\
\qquad\qquad q_2 \leftarrow hadamardD \prec q_1 \\
\qquad\qquad u_1 \leftarrow discqD \prec q_2 \\
\qquad\qquad e_2 \leftarrow simplqD \prec (u_1, e_1) \\
\qquad\qquad u_2 \leftarrow discqD \prec e_2 \\
\qquad\qquad returnA \prec u_2
\end{array}
$$

where the function

$$discqD :: PDQTrans\ Bool\ ()$$

discards a qubit, which physically corresponds to measuring it, returning a real value for the probability distribution; $simplqD$ just simplifies unity ().

$$
\begin{array}{l}
bob :: PDQTrans\ Bool\ Bool \\
bob = PDQTrans\ (\lambda x \rightarrow PDQ\ [((le_1, v_3), p_1) \mid ((le_1, v_1), p_1) \leftarrow unPDQ\ x, \\
\qquad \textbf{let}\ v_2 = \textbf{if}\ ((head\ le_1) \equiv 1)\ \textbf{then}\ v_1 \gg\!= qnot\ \textbf{else}\ v_1, \\
\qquad \textbf{let}\ v_3 = \textbf{if}\ ((head\ (tail\ le_1)) \equiv 1)\ \textbf{then}\ v_2 \gg\!= z\ \textbf{else}\ v_2]) \\
teleportation :: PDQTrans\ (Bool, Bool, Bool)\ Bool \\
teleportation = proc\ (eprL, eprR, q) \rightarrow \textbf{do}
\end{array}
$$

$$
\begin{aligned}
&u_1 \leftarrow alice \prec (eprL, q)\\
&q' \leftarrow bob \prec eprR\\
&returnA \leftarrow q'
\end{aligned}
$$

5 Conclusions and Future Work

We have presented two general and complete models for combined (quantum and classical) computations structured as arrows. The presentation is a stepping stone to develop a language in which the classical, probabilistic, and quantum layers are separate, which would simplify reasoning about quantum programs. The implementation is a prototype of the ideas in Haskell. We hope to integrate the results in some quantum programming language like QML [1].

References

[1] Altenkirch, T. and J. Grattage, *A functional quantum programming language*, in: *Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science, LICS 2005*, IEEE Computer Society Press, 2005, pp. 249–258.

[2] Altenkirch, T. and B. Reus, *Monadic presentations of lambda terms using generalized inductive types*, in: *Computer Science Logic*, 1999.

[3] Bennett, C. H., G. Brassard, C. Crepeau, R. Jozsa, A. Peres and W. Wootters, *Teleporting an unknown quantum state via dual classical and EPR channels*, Phys. Rev. Lett. (1993), pp. 1895–1899.

[4] Gay, S. J. and R. Nagarajan, *Communicating quantum processes*, in: *Proceedings of the 32nd ACM Symposium on Principles of Programming Languages*, 2005.

[5] Hughes, J., *Generalising monads to arrows*, Science of Computer Programming **37** (2000), pp. 67–111.

[6] Kashefi, E., P. Panangaden and V. Danos, *The measurement calculus* (2004).

[7] Moggi, E., *Computational lambda-calculus and monads*, in: *Proceedings of the Fourth Annual Symposium on Logic in computer science*, IEEE Computer Society Press (1989), pp. 14–23.

[8] Moggi, E., *Notions of computation and monads*, Information and Computation **93** (1991), pp. 55–92.

[9] Nielsen, M. A., *Universal quantum computation using only projective measurement, quantum memory, and preparation of the 0 state*, Phys. Lett. **A. 308 (2-3)** (2003), pp. 96–100.

[10] Paterson, R., *A new notation for arrows*, in: *International Conference on Functional Programming* (2001), pp. 229–240.

[11] Raussendorf, R., D. Browne and H. Briegel, *Measurement-based quantum computation with cluster states*, Phys. Rev. A **68** (2003).

[12] Unruh, D., *Quantum programs with classical output streams*, in: *Proceedings of the 3rd International Workshop on Quantum Programming Languages (QPL 2005)*, Electronic Notes in Theoretical Computer Science (2006).

[13] Vizzotto, J. K., T. Altenkirch and A. Sabry, *Structuring quantum effects: Superoperators as arrows*, Mathematical Structures in Computer Science, special issue on Quantum Programming Languages (2006), to appear.