Structuring General and Complete Quantum Computations in Haskell: The Arrows Approach

Juliana Kaizer Vizzotto[1], Orientador: Antônio Carlos da Rocha Costa[1 2], Co-orientador: Amr Sabry[3]

[1]PPGC – Instituto de Informática - Universidade Federal do Rio Grande do Sul
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil

[2]PPGI – Escola de Informática - Universidade Católica de Pelotas
Pelotas, RS

[3]Computer Science Department, Indiana University, Bloomington, USA

jkv@inf.ufrgs.br, rocha@atlas.ucpel.tche.br, sabry@cs.indiana.edu

Abstract. *In this thesis 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 thus structure general and complete quantum programming in Haskell using well known constructions from classical semantics and programming languages, like monads and arrows. The result connects "generic" and "complete" quantum features to well-founded semantics constructions and programming languages.*

1. Introduction

Quantum computation [Nielsen and Chuang 2000] can be understood as *transformation* of information encoded in the state of a *quantum* physical system. Its basic idea is to encode data using quantum bits (qubits). Differently from the classical bit, the qubit can be in a *superposition* of basic states leading to "quantum parallelism", which is an important characteristic of quantum computation since it can greatly increase the speed processing of algorithms. However, quantum data types are computationally very powerful not only due to superposition. There are other odd properties like *measurement* and *entangled*.

In this thesis 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 thus structure general and complete quantum programming in the functional programming language Haskell [Jones et al. 1999] using well known constructions from classical semantics and programming languages, like *monads* [Moggi 1989] and *arrows* [Hughes 2000]. In more detail, this thesis focuses on the following contributions: i) *understanding of quantum effects via monads and arrows*. Quantum parallelism, entanglement, and measurement certainly go beyond "pure" functional programming. We have shown that quantum parallelism can be modelled using a slightly generalisation of monads called *indexed monads*, or *Kleisli structures*. We have also shown that quantum measurement can be explained using a more radical generalisation of monads, the so-called *arrows*, more specifically, *indexed arrows*, which we define in this thesis. This result connects "generic" and "complete" quantum features to well-founded semantics constructions and programming languages; ii) *a computational interpretation of quantum mechanics*. In a thought experiment, Einsten, Podolsky, and Rosen demonstrate some counter-intuitive consequences of quantum mechanics [Bell 1987]. The basic idea is that two entangled particles appear to always communicate some information even when they are separated by arbitrarily large distances. There has been endless debate on this topic, but it is interesting that, as proposed by Amr Sabry [Sabry 2003], this strangeness can be essentially modelled by assignments to global variables. We build on that, and model entanglement using the general notions of computational effects embodied in monads and arrows.

The article is structured as follows. Section 2 describes a monadic approach for "pure" (without measurement) quantum programming in Haskell. In Section 3, after modelling density matrices and superoperators in Haskell, we structure this model for "general" quantum computations (including measurements) using a generalisation of monads called indexed arrows. In Section 4 we deal with "complete" quantum computations (including communication between quantum and classical data). Section 5 concludes. The thesis described in this article is available at `http://www.inf.ufrgs.br/~jkv/thesis.pdf`.

2. Modeling Quantum Effects I: State Vectors as Indexed Monads

The traditional model of quantum computing is based on vector spaces, with *normalized vectors* to model computational states and *unitary transformations* to model physically realizable quantum computations. The idea is that information processing is physically realized via a *closed quantum system*.

In a closed quantum system, the evolution is *reversible* (also called *strict* or *pure*), that is, it is only given by means of unitary gates; measurements, which model the *interaction* with external world, are not considered. Therefore, in this context, the quantum computational process is considered like a black box, where information can be input and then read at the end of the process.

There are some intrinsic differences between classical and quantum programming due to the nature of quantum states and operations acting on these states. One can emphasize two main characteristics of quantum programming: i) quantum parallelism, which is caused by the quantum superposition phenomenon and expressed by *vector* states; ii) *global* (possible entangled) quantum state, which is why not all composed vectors, that model a quantum state, can be decomposed into their subparts. Each operation is global, yet in quantum circuits this global action is hidden. Abstractly, the application of a specific operation to a specific *subspace* of the vector space is achieved by the application of an operation to the whole space which carries the identity to the remaining subspaces. The semantics of any quantum programming language needs to take care of that.

In this section we present a monadic approach for quantum programming in Haskell and show how to structure quantum state vectors using monads, so that the application of unitary transformations to state vectors is modelled by the *bind* operation.

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.

In Haskell, a monad is represented using a type constructor for computations m and two functions:

$$\begin{array}{l} return :: forall\ a.a \rightarrow m\ a \\ \gg= ::forall\ a\ b.m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b \end{array}$$

The operation $\gg=$ (pronounced "bind") specifies how to sequence computations and $return$ specifies how to lift values to computations. The requirements of $forall$ in the definitions above state that the constructor is induced by an endofunctor T in some value category $\mathcal{C}$. 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* [Altenkirch and Reus 1999]. 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 [1] 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:

$$\begin{array}{l} return :: forall\ a.F\ a \Rightarrow a \rightarrow m\ a \\ \gg= ::forall\ a\ b.F\ a, F\ b \Rightarrow m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b \end{array}$$

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 $\gg=$. To construe a proper or *indexed* monad, the $return$ and $\gg=$ functions must work according to the monad laws [Moggi 1989].

2.2. Vectors

Given a set a representing observable (classical) values, i.e. a *basis* set, a pure quantum state is a vector $a \rightarrow \mathbb{C}$ 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:

$$\begin{array}{l} \mathbf{class}\ Eq\ a \Rightarrow Basis\ a\ \mathbf{where}\ basis :: [a] \\ \mathbf{type}\ K = Complex\ Double \\ \mathbf{type}\ Vec\ a = a \rightarrow K \end{array}$$

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

The monadic functions for vectors are defined as:

$$\begin{array}{l} return :: Basis\ a \Rightarrow a \rightarrow Vec\ a \\ return\ a\ b = \mathbf{if}\ a \equiv b\ \mathbf{then}\ 1.0\ \mathbf{else}\ 0.0 \\ (\gg=) :: (Basis\ a, Basis\ b) \Rightarrow Vec\ a \rightarrow (a \rightarrow Vec\ b) \rightarrow Vec\ b \\ va \gg= f = \lambda b \rightarrow sum\ [(va\ a) * (f\ a\ b) \mid a \leftarrow basis] \end{array}$$

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

$return$ 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 1 *The indexed monad* Vec *satisfies the required equations for monads.*

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

```
instance $Basis\ Bool$ where
  $basis = [False, True]$
$qFalse, qTrue, qFT, qFmT :: Vec\ Bool$
$qFalse = return\ False$
$qTrue = return\ True$
$qFT = (1\ /\ \sqrt{2})\ \$*\ (qFalse\ `mplus`\ qTrue)$
```

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 $|\,\mathit{False}\rangle$, $|\,\mathit{True}\rangle$, $\frac{1}{\sqrt{2}}(|\,\mathit{False}\rangle + |\,\mathit{True}\rangle)$, and $\frac{1}{\sqrt{2}}(|\mathit{False}\rangle - |\mathit{True}\rangle)$. The operations $\$*$, and $`mplus'$, are the usual scalar product, and sum of vectors, respectively.

Unitary operations can also be defined directly, for example:

```
type $Uni\ a\ b = a \to Vec\ b$
$hadamard :: Uni\ Bool\ Bool$
$hadamard\ False = qFT$
$hadamard\ True = qFmT$
```

3. Modeling Quantum Effects II: Superoperators as Indexed Arrows

While the state vector model of quantum computing is still widely considered as a convenient formalism to describe quantum algorithms, using measurements to deal with decoherence or noise, to make quantum computing an *interactive* process, and even to steer quantum computations has been considered a novel alternative, for instance see- [Aharonov et al. 1998, Raussendorf et al. 2003, Danos et al. 2005].

In this section we review the general model of quantum computations, including measurements, based on density matrices and superoperators. After expressing it in Haskell, we establish that the superoperators used to express all quantum computations and measurements are an instance of the concept of *indexed arrows*, a generalisation of monads. The material presented on this section was published in [Vizzotto et al. 2006a].

3.1. Indexed Arrows

To handle situations where monads are inapplicable, Hughes [Hughes 2000] introduced a new abstraction generalising monads, called *arrows*. Indeed, in addition to defining a notion of procedure 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.

$$\begin{aligned}
&arr \;\; :: forall\ b\ c.(b \rightarrow c) \rightarrow a\ b\ c\\
&(\ggg) :: forall\ b\ c\ d.a\ b\ c \rightarrow a\ c\ d \rightarrow a\ b\ d\\
&first \;\; :: forall\ b\ c\ d.a\ b\ c \rightarrow a\ (b,d)\ (c,d)
\end{aligned}$$

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*:

$$\begin{aligned}
&arr :: (I\ b, I\ c) \Rightarrow (b \rightarrow c) \rightarrow a\ b\ c\\
&(\ggg) :: (I\ b, I\ c, I\ d) \Rightarrow a\ b\ c \rightarrow a\ c\ d \rightarrow a\ b\ d\\
&first :: (I\ b, I\ c, I\ d) \Rightarrow a\ b\ c \rightarrow a\ (b,d)\ (c,d)
\end{aligned}$$

The operations for arrows or *indexed* arrows must satisfy the arrow laws [Hughes 2000], such that these operations are well-defined even with arbitrary permutations and change of associativity.

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*.

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

Mappings between density matrices are called *superoperators*:

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

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

Just as the probability effect associated with vectors is modelled by a *indexed monad* because of the $Basis$ constraint, the type $Super$ is modelled by an *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 [Vizzotto et al. 2006a]).

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 [Gay and Nagarajan 2005, Unruh 2005] a *complete* model for expressing quantum algorithms should accommodate both measurements and combined interactions of quantum and classical data.

4. Modeling Quantum Effects III: Mixed Programs with Density Operators and Classical Outputs as Indexed Arrows

The model presented in last section is purely quantum. However, various quantum algorithms are explained in terms of the *interchanging* of quantum and classical information [2]. For instance, quantum teleportation is a traditional example of an algorithm which is based on two quantum process communicating via *classical data*. There is interest to consider a *mixed* model for quantum computations involving *measurements* and the *information flow* between quantum and classical processes (for instance, see [Raussendorf et al. 2003, Gay and Nagarajan 2005, Unruh 2005]).

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.

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 *measurement value* 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.

Hence, in this section we present a model for *mixed* or *combined* quantum computations based on a measurement approach over density matrices. We call mixed or combined quantum computation any computation transforming a combined state, with classical and quantum data. Essentially, the idea is to have a density operator representing the (global) quantum part, and a probability distribution of classical values representing the classical part of the state. A quantum program acting on this combined 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). The material presented on this section has been published in [Vizzotto et al. 2006b].

4.1. 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*):

```
class $(Basis\ a, Basis\ b, Basis\ o) \Rightarrow Dec\ a\ b\ o$ where
  $dec :: [a] \rightarrow [(b, o)]$
```

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

[2]By interchanging we mean, for instance, a measurement in the middle of the computation.

part to be measured, is represented by a superoperator from a to b, delivering a classical probability distribution over o.

$$\begin{array}{l}\textbf{type } \mathit{DProb}\ c = [(c, \mathit{Prob})]\\ \textbf{type } \mathit{QProgram}\ i\ o\ a\ b = (\mathit{DProb}\ i, (a, a)) \rightarrow (\mathit{DProb}\ o, \mathit{Dens}\ b)\end{array}$$

Note that the programs should satisfy the restriction $\mathit{Dec}\ a\ b\ o$, and that $\mathit{DProb}\ i$ 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.

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

$$\begin{array}{l}\mathit{uni2qprog} :: (\mathit{Basis}\ a, \mathit{Basis}\ b, \mathit{Basis}\ i, \mathit{Dec}\ a\ b\ ()) \Rightarrow\\ \qquad\qquad\qquad \mathit{Lin}\ a\ b \rightarrow \mathit{QProgram}\ i\ ()\ a\ b\end{array}$$

The function $\mathit{uni2qprog}$ 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 $\mathit{hadamard}$ 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.

$$\begin{array}{l}\mathit{trR} :: (\mathit{Basis}\ a, \mathit{Basis}\ b, \mathit{Dec}\ (a, b)\ a\ b) \Rightarrow \mathit{QProgram}\ i\ b\ (a, b)\ a\\ \mathit{trA} :: (\mathit{Basis}\ a, \mathit{Basis}\ i, \mathit{Dec}\ a\ ()\ a) \Rightarrow \mathit{QProgram}\ i\ a\ a\ ()\end{array}$$

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.

We define the three functions, arr, $\ggg$, and first, over $\mathit{QProgram}\ i\ o$ leaving to the following proposition:

Proposition 2 *The indexed arrow* $\mathit{QProgram}\ i\ o$ *satisfies the required equations for arrows.*

5. Conclusion

We have presented a general and complete model for combined (quantum and classical) computations structured as arrows. The work is a stepping stone to develop a language in which the classical, probabilistic, and quantum layers are separate.

References

Aharonov, D., Kitaev, A., and Nisan, N. (1998). Quantum circuits with mixed states. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 20–30. New York: ACM Press.

Altenkirch, T. and Reus, B. (1999). Monadic presentations of lambda terms using generalized inductive types. In *Proc. Computer Science Logic*.

Bell, J. S. (1987). On the Einstein-Podolsky-Rosen paradox. In *[**?**]*, pages 14–21. Cambridge University Press.

Danos, V., Hondt, E. D. ., Kashefi, E., and Panangaden, P. (2005). Distributed measurement-based quantum computation. In Selinger, P., editor, *Proceedings of the 3rd International Workshop on Quantum Programming Languages*, Electronic Notes in Theoretical Computer Science, Chicago, USA. [S.l.] Elsevier Science.

Gay, S. J. and Nagarajan, R. (2005). Communicating quantum processes. In *Proceedings of the 32nd ACM Symposium on Principles of Programming Languages*.

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

Jones, S. P., Hughes, J., Augustsson, L., Barton, D., Boutel, B., Burton, W., Fasel, J., Hammond, K., Hinze, R., Hudak, P., Johnsson, T., Jones, M., Launchbury, J., Meijer, E., Peterson, J., Reid, A., Runciman, C., and Wadler, P. (1999). *Haskell 98: A Non-strict, Purely Functional Language*. `http://www.haskell.org/onlinereport/`.

Moggi, E. (1989). Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in computer science*, pages 14–23. IEEE Press.

Nielsen, M. A. and Chuang, I. L. (2000). *Quantum Computation and Quantum Information*. Cambridge University Press.

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

Sabry, A. (2003). Modeling quantum computing in Haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 39–49. ACM Press.

Unruh, D. (2005). Quantum programs with classical output streams. *Electronic Notes in Theoretical Computer Science*. 3rd International Workshop on Quantum Programming Languages, to be published.

Vizzotto, J. K., Altenkirch, T., and Sabry, A. (2006a). Structuring quantum effects: Superoperators as arrows. *Journal of Mathematical Structures in Computer Science: special issue in quantum programming languages*, 16:453–468.

Vizzotto, J. K., Costa, A. C. R., and Sabry, A. (2006b). Quantum arrows in haskell. In *Proc. 4th International Workshop on Quantum Programming Languages*, Oxford. to appear in Electronic Notes in Theoretical Computer Science (ENTCS).