Math. Struct. in Comp. Science

Structuring Quantum Effects: Superoperators as Arrows

JULIANA VIZZOTTO[1], THORSTEN ALTENKIRCH[2] and AMR SABRY[3]

[1] *Institute of Informatics, Federal University of Rio Grande do Sul, Porto Alegre, Brazil.*
[2] *School of Computer Science and IT, The University of Nottingham, UK.*
[3] *Department of Computer Science, Indiana University, USA.*

Received January 2005

We show that the model of quantum computation based on density matrices and superoperators can be decomposed in a pure classical (functional) part and an effectful part modelling probabilities and measurement. The effectful part can be modelled using a generalisation of monads called arrows. We express the resulting executable model of quantum computing in the programming language Haskell using its special syntax for arrow computations. The embedding in Haskell is however not perfect: a faithful model of quantum computing requires type capabilities which are not directly expressible in Haskell.

1. Introduction

A newcomer to the field of quantum computing is immediately overwhelmed with many apparent differences with classical computing that suggest that quantum computing might require radically new semantic models and programming languages. In some sense this is true for two reasons: (1) quantum computing is based on a kind of parallelism caused by the non-local wave character of quantum information which is qualitatively different from the classical notion of parallelism, and (2) quantum computing has a peculiar notion of observation in which the observed part of the quantum state and every other part that is entangled with it immediately lose their wave character. Interestingly it seems that none of the other differences that are often cited between quantum and classical computing are actually relevant semantically. For example, even though we do not often think of classical computation as "reversible," it is just as reversible as quantum computing. Both can be implemented by a set of reversible universal gates (see (Nielsen and Chuang 2000), section 1.4.1), but in neither model should the user be required to reason about reversibility.

The two properties of quantum computing discussed above certainly go beyond "pure" classical programming and it has been suspected earlier that they might correspond to some notion of computational effect. Following Moggi's influential paper (Moggi 1989), computational effects like assignments, exceptions, non-determinism, could all be modelled using the categorical construction of a *monad*. 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 (Hughes 2000) which is also internalised in the programming language Haskell.

Mu and Bird (2001) showed that the simple model of quantum computing based on vectors and linear operators is almost a monad. We expand and build on this observation as follows. First the model built using vectors and linear operators cannot even express measurements. There exists however a more general model based on density matrices and superoperators which can express measurements. After expressing this more general model in Haskell, we establish that the superoperators used to express all quantum computations and measurements are indeed an instance of the concept of arrows. In particular the construction clarifies the crucial need for some form of linear typing: arrow computations must be required to use every quantum value or else the computations produce results that are inconsistent with quantum mechanics.

In summary, our construction relates "unusual" quantum features to well-founded semantic constructions and programming languages. We hope it will serve as a useful tool to further understand the nature and structure of quantum computation. The remainder of the paper is organised as follows. Section 2 presents the model of quantum computing using vectors and linear operators and its implementation in Haskell, focusing on the possibility of structuring the effects using monads. Section 3 discusses the limitations of this simple model as a complete model of quantum computation which should include measurement. Section 4 introduces a more general model of quantum based on density matrices and superoperators. Our main result is discussed in Section 5 where we show that general quantum computations including measurement can be structured using the generalisation of monads called arrows. Section 6 gives two complete examples implementing a Toffoli circuit and the teleportation experiment: both examples use the arrow notation to express the structure of the computation elegantly. Section 7 discusses the limitations of our Haskell embedding and its connection to the functional quantum programming language QML (Altenkirch and Grattage 2005). Section 8 concludes. The entire Haskell code is available online (Vizzotto *et al.* 2005).

2. The Traditional Model of Quantum Computing

We present the traditional model of quantum computing in this section.

2.1. *Vectors*

A finite set a can be represented in Haskell as an instance of the class *Basis* below, which has the constructor $basis :: [a]$ explicitly listing the basis elements. Given such a set a representing observable (classical) values, a pure quantum value is a vector $a \to \mathbb{C}$ which associates each basis element with a complex probability amplitude. The basis elements must be distinguishable from each other which explains the constraint $Eq\ a$ on the type of elements below:

```
class Eq a => Basis a where basis :: [a]
type K = Complex Double
type Vec a = a → K
```

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

From a programming perspective, a monad is a way to structure *computations* in terms of values and sequences of computations using those values. The type constructor *Vec* is technically not a monad: it corresponds to a *Kleisli structure* (Altenkirch and Reus 1999). Yet as noted by Mu and Bird (2001), the probabilities introduced by vector spaces constitute a computational effect which can be structured using a slight generalisation of monads in Haskell (Moggi 1989). A monad is represented using a type constructor for computations m and two functions: $return :: a \to m\ a$ and $\ggg\!= :: m\ a \to (a \to m\ b) \to m\ b$. The operation $\ggg\!=$ (pronounced "bind") specifies how to sequence computations and *return* specifies how to lift values to computations:

```
return :: Basis a => a → Vec a
return a b = if a≡b then 1.0 else 0.0

(>>=) :: Basis a => Vec a → (a → Vec b) → Vec b
va >>= f = λ b → sum [ (va a) * (f a b) | a ∈ basis]
```

Because of the additional constraint that our computations must be over specified bases whose elements must be comparable, the types of our operations are more restricted than strictly desired for a monad. However *return* and $\ggg\!=$ satisfy the three monad laws.

Proposition 2.1. Vector spaces satisfy the required equations for monads.

Vector spaces have additional properties abstracted in the Haskell class *MonadPlus*. Instances of this class support two additional methods: *mzero* and *mplus* which provide a "zero" computation and an operation to "add" computations:

```
mzero :: Vec a
mzero = const 0.0

mplus :: Vec a → Vec a → Vec a
mplus v1 v2 a = v1 a + v2 a
```

We also have *mminus* which is defined analogously.

For convenience, it is also possible to define various kinds of products over vectors: the *scalar* product $\$*$, the *tensor* product $\langle * \rangle$, and the *dot* product $\langle \cdot \rangle$:

```
($*) :: K → Vec a → Vec  a
pa $* v = λa → pa * v a

(<*>) :: Vec a → Vec b → Vec (a,b)
v1 <*> v2 = λ (a,b) → v1 a * v2 b

(<.>) :: Basis a => Vec a → Vec a → K
v1 <.> v2 = sum [ conjugate (v1 a) * (v2 a) | a ∈ basis ]
```

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

```
instance Basis Bool where basis = [False,True]
```

```
qFalse,qTrue :: Vec Bool   qFT,qFmT :: Vec Bool
qFalse = return False      qFT = (1/sqrt 2) $* (qFalse `mplus` qTrue)
qTrue = return True        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 $|\mathit{False}\rangle$, $|\mathit{True}\rangle$, $\frac{1}{\sqrt{2}}(|\mathit{False}\rangle + |\mathit{True}\rangle)$, and $\frac{1}{\sqrt{2}}(|\mathit{False}\rangle - |\mathit{True}\rangle)$.

Vectors over several values can be easily described using the tensor product on vectors or the Cartesian product on the underlying bases:

```
instance (Basis a, Basis b) => Basis(a, b) where
  basis = [(a, b) | a ∈ basis, b ∈ basis ]

  p1,p2,p3:: Vec (Bool,Bool)    epr :: Vec (Bool,Bool)
  p1  =  qFT <*> qFalse         epr (False,False)  =  1 / sqrt 2
  p2  =  qFalse <*> qFT         epr (True,True)    =  1 / sqrt 2
  p3  =  qFT <*> qFT            epr _              =  0
```

In contrast to the first three vectors, the last vector describes an *entangled* quantum state which cannot be separated into the product of independent quantum states. The name of the vector "*epr*" refers to the initials of Einstein, Podolsky, and Rosen who used such a vector in a thought experiment to demonstrate some counterintuitive consequences of quantum mechanics (Einstein *et al.* 1935).

2.2. *Linear Operators*

Given two base sets A and B a linear operator $f \in A \multimap B$ is a function mapping vectors over A to vectors over B. We represent such operators as functions mapping values to vectors which is similar to the representation used by Karczmarczuk (2003):

```
type Lin a b = a → Vec b

fun2lin :: (Basis a, Basis b) => (a → b) → Lin a b
fun2lin f a = return (f a)
```

The function *fun2lin* converts a classical (reversible) function to a linear operator. For example, the quantum version of the boolean negation is: `qnot = fun2lin not`. Linear operations can also be defined directly, for example:

```
phase :: Lin Bool Bool                      hadamard :: Lin Bool Bool
phase False = return False                  hadamard False = qFT
phase True = (0 :+ 1) $* (return True)      hadamard True = qFmT
```

The definition of a linear operation specifies its action on one individual element of the basis. To apply a linear operation f to a vector v, we use the *bind* operation to calculate $v \gg= f$. For example $(qFT \gg= hadamard)$ applies the operation *hadamard* to the vector qFT which one can calculate produces the vector *qFalse* as a result.

It is possible to write higher-order functions which consume linear operators and produce new linear operators. An important example of such functions produces the so-called *controlled operations*:

```
controlled :: Basis a => Lin a a → Lin (Bool,a) (Bool,a)
controlled f (b1,b2) = (return b1) <*> (if b1 then f b2 else return b2)
```

The linear operator f is transformed to a new linear operator controlled by a quantum boolean value. The modified operator returns a pair whose first component is the input control value. The second input is passed to f only if the control value is true, and is otherwise left unchanged. For example, $(qFT\ \langle * \rangle\ qFalse) \gg= (controlled\ qnot)$ applies the familiar *controlled-not* gate to a vector over two values: the control value is a superposition of *False* and *True* and the data value is *False*. As one may calculate the result of this application is the *epr* vector.

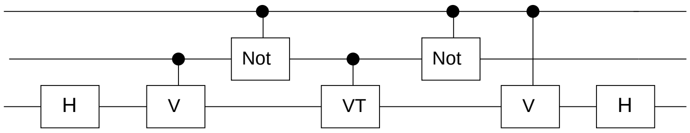
Linear operations can be combined and transformed in several ways which we list below. The function $\rangle{*}\langle$ produces the linear operator corresponding to the *outer product* of two vectors. The functions *linplus* and *lintens* are the functions corresponding to the sum and tensor product on vectors. Finally the function o composes two linear operators.

```
adjoint :: Lin a b → Lin b a
adjoint f b a = conjugate (f a b)

(>*<) :: (Basis a, Basis b) => Vec a → Vec b → Lin a b
(v1 >*< v2) a b = v1 a * conjugate (v2 b)

linplus :: (Basis a, Basis b) => Lin a b → Lin a b → Lin a b
linplus f g a = f a `mplus` g a

lintens :: (Basis a, Basis b, Basis c, Basis d) =>
                Lin a b → Lin c d → Lin (a,c) (b,d)
lintens f g (a,c) = f a <*> g c

o :: (Basis a, Basis b, Basis c) => Lin a b → Lin b c → Lin a c
o f g a = (f a >>= g)
```

2.3. *Example: A Toffoli Circuit*

The circuit diagram uses the de-facto standard notation for specifying quantum computations. Each line carries one quantum bit (*qubit*); we refer to the three qubits in the circuit as *top*, *middle*, and *bottom*. The values flow from left to right in steps corresponding to the alignment of the boxes which represent quantum gates. The gates labeled H, V, VT, and *Not* represent the quantum operations *hadamard*, *phase*, *adjoint phase*, and *qnot* respectively. Gates connected via a bullet to another wire are *controlled* operations.

In general all three qubits in the circuit may be entangled and hence the state vector



representing them cannot be separated into individual state vectors. This means that, despite the appearance to the contrary, it is not possible to operate on any of the lines individually. Instead the circuit defines a linear operation on the entire state:

```
toffoli :: Lin (Bool,Bool,Bool) (Bool,Bool,Bool)
toffoli (top,middle,bottom) =
   let cnot = controlled qnot
       cphase = controlled phase
       caphase = controlled (adjoint phase)
   in hadamard bottom >>= λ b1 →
      cphase (middle,b1) >>= λ (m1,b2) →
      cnot (top,m1) >>= λ (t1,m2) →
      caphase (m2,b2) >>= λ (m3,b3) →
      cnot (t1,m3) >>= λ (t2,m4) →
      cphase (t2,b3) >>= λ (t3,b4) →
      hadamard b4 >>= λ b5 →
      return (t3,m4,b5)
```

3. Measurement

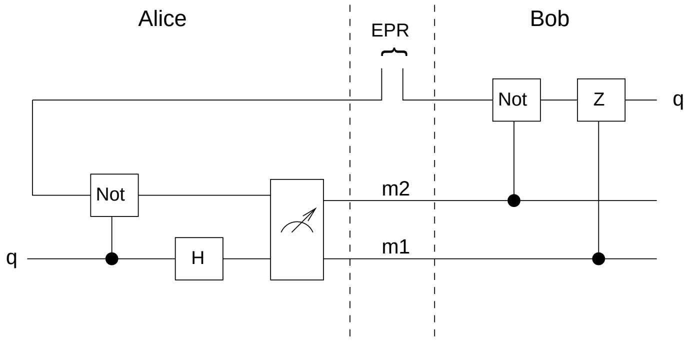
The use of monads to structure the probability effects reveals an elegant underlying structure for quantum computations. This structure can be studied in the context of category theory and exploited in the design of calculi for quantum computation (van Tonder 2003; van Tonder 2004; Valiron 2004; Altenkirch and Grattage 2005).

Unfortunately in the simple model of quantum computing we have used so far, it is difficult or impossible to deal formally with another class of quantum effects, including measurements, decoherence, or noise. We first give one example where such effects are critical, and then discuss various approaches in the literature on how to deal with such effects.

3.1. *Teleportation*

The idea of teleportation is to disintegrate an object in one place making a perfect replica of it somewhere else. Indeed quantum teleportation (Bennett *et al.* 1993) enables the transmission, *using a classical communication channel*, of an unknown quantum state via a previously shared *epr* pair.

In the following diagram, Alice and Bob initially have access to one of the qubits of an entangled *epr* pair, and Alice aims to teleport an unknown qubit q to Bob:



The calculation proceeds as follows. First Alice interacts with the unknown qubit q and her half of the *epr* state. Then Alice performs a measurement collapsing her quantum state and getting two classical bits m_1 and m_2 that she transmits to Bob using a classical channel of communication.

Upon receiving the two classical bits of information, Bob interacts with his half of the *epr* state with gates controlled by the classical bits. The circuit in the figure can be shown to re-create the quantum state q which existed at Alice's site before the experiment.

Our main interest in this circuit is that it is naturally expressed using a sequence of operations on quantum values which include a non-unitary *measurement* in the middle. Using the model developed in the previous section, it is not possible to describe this algorithm as stated. In the next section, we briefly discuss several possible ways to deal with this problem.

3.2. *Dealing with Measurement*

The literature includes several approaches to dealing with the problem of measurement. We characterize such approaches in three broad categories: deferring measurements, using classical control with pointers and side-effects, and using density matrices and superoperators. We discuss the first two approaches in the remainder of this section, and expand on the latter approach in the next section.

3.2.1. *Deferring measurements.* The first approach used for example by Mu and Bird (2001), Van Tonder (2003; 2004) and Karczmarczuk (2003) relies on the *principle of deferred measurement* (Nielsen and Chuang 2000). This principle can be used to transform computations to *always* defer measurements to the end. Using this idea one can focus entirely on managing the probability effects and perform the measurements outside the formalism. The drawback of this approach is clear: programs that interleave quantum operations with measurements cannot be expressed naturally. For example, transforming the teleportation circuit above to defer the measurements until after Bob's computation completely changes the character of the experiment, because no *classical* information is transmitted from Alice to Bob.

3.2.2. *Classical Control and Side-effects.* In general, this category of models is based on the quantum random access machine (QRAM) model of Knill (1996), which is summarized by the slogan "quantum data, classical control" (Selinger 2004). In this context, a quantum computer can be seen as a classical computer with a quantum device attached to it. The classical control sends instructions for the quantum machine to execute unitary operations and measurements. The most common approach to computationally realize this hybrid architecture is via manipulating what are effectively *pointers* to a *global shared quantum state* (Ömer 1998; Sabry 2003; Selinger and Valiron 2005; Skibinski 2001). The use of pointers and sharing to model the side-effect of measurement is in some sense adequate. However by doing so, we completely lose the monadic structure and the direct connections to categorical semantics.

4. Density Matrices and Superoperators

Fortunately the above model of quantum computing can be generalised to solve the problem of modelling measurements in a better way. In the generalised model, the state of the computation is represented using a *density matrix* and the operations are represented using *superoperators* (Aharonov *et al.* 1998). Using these notions, the *projections* necessary to express measurements become expressible within the model. We review this model in this section.

4.1. *Density Matrices*

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 modelled by its outer product.

```
type Dens a = Vec (a,a)

pureD :: Basis a => Vec a → Dens a
pureD v = lin2vec (v >*< v)

lin2vec :: (a → Vec b) → Vec (a,b)
lin2vec = uncurry
```

The function *pureD* embeds a state vector in its density matrix representation. For convenience, we uncurry the arguments to the density matrix so that it looks more like a "matrix." For example, the density matrices corresponding to the vectors *qFalse*, *qTrue*, and *qFT* can be visually represented as follows:

$$\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \qquad \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \qquad \begin{pmatrix} 1/2 & 1/2 \\ 1/2 & 1/2 \end{pmatrix}$$

The appeal of density matrices is that they can represent states other than the pure ones above. In particular if we perform a measurement on the state represented by qFT, we should get *False* with probability $1/2$ or *True* with probability $1/2$. This information, which cannot be expressed using vectors, can be represented by the following density matrix:

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

Such a density matrix represents a *mixed state* which corresponds to the sum (and then normalisation) of the density matrices for the two results of the observation.

4.2. *Superoperators*

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

```
type Super a b = (a,a) → Dens b

lin2super :: (Basis a, Basis b) => Lin a b → Super a b
lin2super f (a1,a2) = lin2vec (f a1 >*< f a2)
```

The function *lin*2*super* constructs a superoperator from a linear operator on vectors.

4.3. *Tracing and Measurement*

In contrast to the situation with the traditional model of quantum computing, it is possible to define a superoperator which "forgets," *projects*, or *traces out* part of a quantum state as well as a superoperator which *measures* part of a quantum state:

```
trL :: (Basis a, Basis b) => Super (a,b) b
trL ((a1,b1),(a2,b2)) =  if a1 ≡ a2 then return (b1,b2) else mzero

meas :: Basis a => Super a (a,a)
meas (a1,a2) = if a1 ≡ a2 then return ((a1,a1),(a1,a1)) else mzero
```

We are considering *projective* measurements which are described by a set of projections onto mutually orthogonal subspaces. This kind of measurement returns a classical value and a post-measurement state of the quantum system. The operation `meas` is defined in such a way that it can encompass both results. Using the fact that a classical value m can be represented by the density matrix $|m\rangle\langle m|$ the superoperator `meas` returns the output of the measurement attached to the post-measurement state.

For example, the sequence:

```
pureD qFT >>= meas >>= trL
```

first performs a measurement on the pure density matrix representing the vector qFT. This measurement produces a vector with two components: the first is the resulting collapsed quantum state and the second is the classical observed value. The last operation forgets about the collapsed quantum state and returns the result of the classical measurement. As explained earlier the resulting density matrix is:

$$\begin{pmatrix} 1/2 & 0 \\ 0 & 1/2 \end{pmatrix}$$

5. Superoperators as Arrows

By moving to density matrices and superoperators, it becomes possible to express both the original computations as well as measurements in the same formalism. One might hope that the original monadic structure of quantum computations is preserved, but it appears that this is not the case. The best we can do is to prove that the new model of computation fits within a generalisation of monads called *arrows*.

5.1. *Arrows*

The application of a superoperator to a density matrix can still be achieved with the monadic bind operation, instantiated to the following type:

```
(>>=) :: Dens a → ((a,a) → Dens b) → Dens b
```

This type does not however correspond to the required type as computations now *consume multiple input values*. This observation is reminiscent of Hughes's motivation for generalising monads to *arrows* (Hughes 2000). Indeed, in addition to defining a notion of procedure which may perform computational effects, arrows may have a static component independent of the input, or may accept more than one input.

In Haskell, the arrow interface is defined using the following class declaration:

```
class Arrow a where
  arr   ::  (b → c) → a b c
  (>>>) ::  a b c → a c d → a b d
  first ::  a b c → a (b,d) (c,d)
```

In other words, to be an arrow, a type a must support the three operations arr, $\ggg$, and $first$ with the given types. The operations must satisfy the following equations:

$$
\begin{array}{rcl}
arr\ id \ggg f & = & f \\
f \ggg arr\ id & = & f \\
(f \ggg g) \ggg h & = & f \ggg (g \ggg h) \\
arr\,(g\ .\ f) & = & arr\ f \ggg arr\ g \\
first\,(arr\ f) & = & arr\,(f \times id) \\
first\,(f \ggg g) & = & first\ f \ggg first\ g \\
first\ f \ggg arr\,(id \times g) & = & arr\,(id \times g) \ggg first\ f \\
first\ f \ggg arr\ fst & = & arr\ fst \ggg f \\
first\,(first\ f) \ggg arr\ assoc & = & arr\ assoc \ggg first\ f
\end{array}
$$

where the functions $\times$ and $assoc$ are defined as follows:

$$
\begin{array}{c}
(f \times g)\ (a,b) = (f\ a, g\ b) \\
assoc\ ((a,b),c) = (a,(b,c))
\end{array}
$$

The function arr allows us to introduce "pure" arrows which are simple functions from their inputs to their outputs. The function $\ggg$ is similar to $\gg\!=$: it composes two computations. The function *first* is the critical one for our purposes: it allows us to apply an arrow to a component of the global quantum state. The equations above ensure that these operations are always well-defined even with arbitrary permutations and change of associativity.

5.2. *Superoperators are Arrows (with Eq constraint)*

Just as the probability effect associated with vectors is not strictly a monad because of the *Basis* constraint, the type *Super* is not strictly an arrow as the following types include the additional constraint requiring the elements to be comparable:

```
arr :: (Basis b, Basis c) => (b → c) → Super b c
arr f = fun2lin (λ (b1,b2) → (f b1, f b2))

(>>>) :: (Basis b, Basis c, Basis d) => Super b c → Super c d → Super b d
(>>>) = o

first :: (Basis b, Basis c, Basis d) => Super b c → Super (b,d) (c,d)
first f ((b1,d1),(b2,d2)) = permute ((f (b1,b2)) <*> (return (d1,d2)))
   where permute v ((b1,b2),(d1,d2)) = v ((b1,d1),(b2,d2))
```

The function *arr* constructs a superoperator from a pure function by applying the function to both the vector and its dual. The composition of arrows is simply the composition of linear operators. The function *first* applies the superoperator f to the first component (and its dual) and leaves the second component unchanged. The definition calculates each part separately and then permutes the results to match the required type.

Proposition 5.1. Superoperators satisfy the required equations for arrows.

The proposition implies that we can use the arrow combinators to structure our computations. For instance, the first few steps of the Toffoli circuit of Section 2.3 would now look like:

```
toffoli :: Super (Bool,Bool,Bool) (Bool,Bool,Bool)
toffoli =
  let hadS = lin2super hadamard
      cphaseS = lin2super (controlled phase)
      cnotS = lin2super (controlled qnot)
  in arr (λ (a0, b0, c0) → (c0, (a0, b0))) >>>
     (first hadS  >>> arr (λ (c1, (a0, b0)) → ((b0, c1), a0))) >>>
     (first cphaseS >>> arr (λ ((b1, c2), a0) → ((a0, b1), c2))) >>>
     (first cnotS >>> arr (λ ((a1, b2), c2) → ((b2, c2), a1))) >>> ...
```

Clearly this notation is awkward as it forces us to explicitly manipulate the entire state and to manually permute the values. However, all the tedious code can be generated automatically as we explain next.

5.3. *A Better Notation for Arrows*

Following the Haskell's monadic **do**-notation, Paterson (2001) presented an extension to Haskell with an improved syntax for writing computations using arrows. We concentrate only on the explanation of new forms which we use in our examples. Here is a simple example to illustrate the notation:

```
e1 :: Super (Bool,a) (Bool,a)
e1 = proc (a,b) → do
         r ← lin2super hadamard ≺ a
         returnA ≺ (r,b)
```

The **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* ≺ which feeds the value of an expression into an arrow.

Paterson (2001) shows that the above notation is general enough to express arrow computations and implemented a preprocessor which translates the new syntax to regular Haskell. In the case of `e1` above, the translation to Haskell produces the following code:

```
e2 :: Super (Bool,a) (Bool,a)
e2 = first (lin2super hadamard)
```

As the example shows, the output of the preprocessor is quite optimised.

5.4. *Superoperators are (probably) not monads*

Arrows are more general than monads. In particular, they include notions of computation that consume multiple inputs as well as computations with static components, independent of the input. Due to this general aspect of arrows, there are some subclasses of them which turns out to be equivalent to monads. More precisely, arrow types which support the following *app* function are just as expressive as monads.

```
class Arrow => ArrowApply a where
  app :: a (a b c, b) c
```

The function *app* is an arrow which also receives an arrow a of type b c, and element of type b, and returns an element of type c. In other words, for superoperators to be monads, we would have to define a superoperator of type *Super* (*Super* b c, b) c. which in our case would require *Super* b c to be an instance of *Basis*. Unfortunately there is no straightforward way to view the space of superoperators as a finite set of observables.

6. Examples Revisited: Toffoli and Teleportation

Using arrows and the notation introduced by Patterson, we can express both of our examples elegantly.

6.1. *Toffoli*

The code mirrors the structure of the circuit and the structure of the monadic computation expressed earlier:

```
toffoli :: Super (Bool,Bool,Bool) (Bool,Bool,Bool)
toffoli = let hadS = lin2super hadamard
              cnotS = lin2super (controlled qnot)
              cphaseS = lin2super (controlled phase)
              caphaseS = lin2super (controlled (adjoint phase))
          in proc (a0,b0,c0) → do
             c1 ← hadS ≺ c0
             (b1,c2) ← cphaseS ≺ (b0,c1)
             (a1,b2) ← cnotS ≺ (a0,b1)
             (b3,c3) ← caphaseS ≺ (b2,c2)
             (a2,b4) ← cnotS ≺ (a1,b3)
             (a3,c4) ← cphaseS ≺ (a2,c3)
             c5 ← hadS ≺ c4
             returnA ≺ (a3,b4,c5)
```

6.2. *Teleportation*

We use the machinery we have developed to faithfully express the circuit presented in Section 3.1. We break the algorithm in two individual procedures, *alice* and *bob*. Besides the use of the arrows notation to express the action of superoperators on specific qubits, we incorporate the measurement in Alice's procedure, and trace out the irrelevant qubits from the answer returned by Bob.

```
alice :: Super (Bool,Bool) (Bool,Bool)
alice = proc (eprL,q) → do
             (q1,e1) ← (lin2super (controlled qnot)) ≺ (q,eprL)
             q2 ← (lin2super hadamard) ≺ q1
             ((q3,e2),(m1,m2)) ← meas ≺ (q2,e1)
             (m1',m2') ← trL ((q3,e2),(m1,m2))
             returnA ≺ (m1',m2')

bob :: Super (Bool,Bool,Bool) Bool
bob = proc (eprR,m1,m2) → do
           (m2',e1) ← (lin2super (controlled qnot)) ≺ (m2,eprR)
           (m1',e2) ← (lin2super (controlled z)) ≺ (m1,e1)
           q' ← trL ≺ ((m1',m2'),e2)
           returnA ≺ q'

teleport :: Super (Bool,Bool,Bool) Bool
teleport = proc (eprL,eprR,q) → do
                (m1,m2) ← alice ≺ (eprL,q)
```

```
q' ← bob ≺ (eprR,m1,m2)
returnA ≺ q'
```

7. Linear Typing: QML

The category of superoperators is considered to be an adequate model of non-reversible quantum computation (Selinger 2004). Our construction presented so far seems to suggest that this category corresponds to a functional language with arrows, and so that we can accurately express quantum computation in such a framework. But as we explain below, this is not quite the whole story.

First consider the well-known "non-cloning" property of quantum states (Nielsen and Chuang 2000). The arrow notation allows us to reuse variables more than once, and we are free to define the following operator:

```
copy :: Super Bool (Bool, Bool)
copy = arr (λ x → (x,x))
```

But can this superoperator be used to clone a qubit? The answer, as explained in Section 1.3.5 of the classic book on quantum computing (Nielsen and Chuang 2000), is no. The superoperator *copy* can be used to copy classical information encoded in quantum data, but when applied to an arbitrary quantum state, for example like qFT, the superoperator does not make two copies of the state qFT but rather it produces the *epr* state which is the correct and desired behaviour. Thus, in this aspect the semantics of arrows is coherent with quantum computation, *i.e.*, the use of variables more than once models sharing, not cloning.

In contrast, in our model there is nothing to prevent the definition of:

```
weaken :: Super (Bool,Bool) Bool
weaken = arr (λ (x,y) → y)
```

This operator is however not physically realizable. Applying *weaken* to *epr* gives qFT. Physically forgetting about x corresponds to a measurement: if we measure the left qubit of *epr* we should get *qFalse* or *qTrue* or the mixed state of both measurements, but never qFT.

Therefore, our use of Haskell as a vehicle for expressing the ideas finally hits a major obstacle: arrow computations must be required to use every value that is introduced. Instead of attempting to continue working within Haskell, a better approach might be to now consider a functional quantum language like QML whose type system is designed to explicitly control weakening and decoherence, and to express the separation of values and arrow computations in that framework.

In more detail, QML (Altenkirch and Grattage 2005) is a functional quantum programming language which addresses this problem by using a type system based on strict linear logic: contraction is permitted and modelled by *copy* while weakening has to be explicit and is translated by a partial trace. QML also features two conditional operators: a classical conditional operator **if** which measures a qubit and returns the appropriate branch and a quantum conditional operator **if**$^\circ$ which avoids measurement but requires that the branches return results in orthogonal subspaces.

QML programs can be compiled to quantum circuits, using the category of finite quantum computation FQC — Grattage's QML compiler (Grattage 2006) is based on this semantics. An irreversible computation can be modelled by a reversible circuit, allowing additional heap qubits, which are initialised to a predefined values at the beginning of the computation and disposing, *i.e.* measuring, qubits at the end of the computation. To any FQC morphism we can assign a superoperator and indeed every superoperator can be represented this way.

Alternatively, we can interpret QML programs directly as superoperators, giving rise to a constructive denotational semantics exploiting the library of arrow combinators developed here. We hope to exploit this semantics to further analyse QML and to develop high level reasoning principles for QML programs.

8. Conclusion

We have argued that a realistic model for quantum computations should accommodate both unitary operations and measurements, and we have shown that such general quantum computations can be modelled using arrows. This is an extension of the previous-known observation that one can model pure quantum probabilities using monads. Establishing such connections between quantum computations and monads and arrows enables elegant embeddings in current classical languages, and exposes connections to well-understood concepts from the semantics of (classical) programming languages. We have demonstrated the use of arrows to model elegantly two examples in Haskell, including the teleportation experiment which interleaves measurements with unitary operations.

Acknowledgements

We would like to thank Antônio Carlos da Rocha Costa and Jonathan Grattage for extensive discussions and feedback. We also would like to thank the anonymous reviewers for very useful comments.

References

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

Altenkirch, T. and Grattage, J. A functional quantum programming language. In *Proceedings of the Twentieth Annual IEEE Symp. on Logic in Computer Science, LICS 2005*, pages 249–258. IEEE Computer Society Press, June 2005.

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

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

Einstein, A., Podolsky, B. and Rosen, N. Can quantum-mechanical description of physical reality be considered complete? *Phys. Rev.*, **47**, 777–780, 1935.

Grattage, J. A functional quantum programming language, PhD, The University of Nottingham, forthcoming, 2006.

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

Karczmarczuk, J. Structure and interpretation of quantum mechanics: a functional framework. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, Uppsala, Sweden, pages 50–61, 2003. ACM Press.

Knill, E. Conventions for quantum pseudocode. Technical Report LAUR-96-2724, Los Alamos National Laboratory, 1996.

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

Mu, S. and Bird, R. Functional Quantum Programming. *Second Asian Workshop on Programming Languages and Systems*, KAIST, Korea, 2001.

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

Ömer, B. A procedural formalism for quantum computing. Master's thesis, Department of Theoretical Physics, Technical University of Vienna, 1998.

Paterson, R. A New Notation for Arrows. In *Proceedings of International Conference on Functional Programming*, Firenze, Italy, pages 229–240, 2001. ACM Press.

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

Selinger, P. Towards a quantum programming language. *Mathematical Structures in Computer Science*, **14** (4),527–586, 2004.

Selinger, P. and Valiron, B. A lambda calculus for quantum computation with classical control. In *Proceedings of the Seventh International Conference on Typed Lambda Calculi and Applications (TLCA), Springer-Verlag, LNCS 3461*, pages 354–368, 2005.

Skibinski, J. Haskell simulator of quantum computer. Available at `http://web.archive.org/web/20010630025035/www.numeric-quest.com/haskell%/QuantumComputer.html`, Initialized: 2001-05-02, last modified: 2001-05-05.

Valiron, B. Quantum typing. In Peter Selinger, *Proceedings of the 2nd International Workshop on Quantum Programming Languages*, TUCS General Publication No **33**, Turku Centre for Computer Science, 2004.

van Tonder, A. Quantum computation, categorical semantics and linear logic. Available as quant-ph/0312174, 2003.

van Tonder, A. A lambda calculus for quantum computation. *SIAM Journal on Computing*, **33** (5),1109–1135, 2004.

Vizzotto, J. K., Altenkirch, T. and Sabry, A. Haskell code for superoperators as arrows. `http://www.inf.ufrgs.br/~jkv/super.tgz`, 2005.