Modeling Quantum Computing in Haskell

Amr Sabry*
Department of Computer Science, Indiana University
sabry@cs.indiana.edu

ABSTRACT

The paper develops a model of quantum computing from the perspective of functional programming. The model explains the fundamental ideas of quantum computing at a level of abstraction that is familiar to functional programmers. The model also illustrates some of the inherent difficulties in interpreting quantum mechanics and highlights the differences between quantum computing and traditional (functional or otherwise) computing models.

Categories and Subject Descriptors

C.0 [**Computer Systems Organizations**]: General; D.1 [**Programming Techniques**]: Applicative (Functional) Programming); F.0 [**Theory of Computation**]: General

General Terms

Algorithms, Languages, Theory

Keywords

Haskell, Quantum Computing, Qubit, Entanglement, Virtual Value, Adaptor

1. INTRODUCTION

Quantum computing evokes strong connections to pure functional programming: it includes a built-in notion of parallelism (even though this notion is qualitatively different from the one found in functional programming) and is based on mathematical foundations (vector spaces, matrix algebra, etc) which can be modeled elegantly in a functional language [15].

It is therefore natural for one to model quantum computing within a functional language. As a first approximation, quantum computing can be seen as an extension of classical probabilistic computation: building on a monad of probabilistic computations, one can develop an elegant but rudimentary model of functional quantum computing [21]. This paper attempts to offer a more complete model of quantum computing in Haskell with two major goals in mind:

1. To explain quantum computing at a level of abstraction familiar to the programming language community instead of the model used by physicists.
2. To elicit the connections between quantum computing and functional programming and evaluate the appropriateness of functional abstractions to the domain of quantum computing.

The first goal is achieved to a reasonable degree. The main challenge here is that any operational model of quantum computing must somehow commit to an *interpretation of quantum mechanics* which has been, and still is, a subject of debate among physicists. In particular, our model must *implement* some mechanism for the collapse of the wave function inherent in measurement. We use global side-effects whether this has anything to do with "physical reality" or not.

As for the second goal, we argue that, contrary to preliminary investigations, functional programming abstractions (as realized in Haskell) are not that well-suited to quantum computing. The mismatches are however quite instructive. First, they explain some of the essence of quantum computing as it differs from functional programming. Most significantly, unlike the case for functional programs, reasoning about quantum systems is non-compositional which we argue requires new abstractions. Second, the mismatches also expose some of the limitations of Haskell when applied to a radically new domain.

The rest of the paper is organized as follows. Section 2 introduces the basic ingredients of quantum computing: quantum data structures. Section 3 augments the model with operations or functions over quantum data. Section 4 addresses the thorny issue of observation or measurement. Section 5 proposes a design pattern reminiscent of the Façade pattern [12] and closely related to Kagawa's *composable references* [14] to conveniently manipulate components of entangled data structures. Section 6 illustrates the resulting model by giving several examples. Finally Section 7 discusses related work and concludes.

We have attempted to make the paper as self-contained as possible but naturally we cannot include a complete introduction to quantum mechanics and its mathematical foundations, nor can we completely survey the field of quantum

*Supported by the National Science Foundation under Grants No. CCR-0196063 and CCR-0204389.

Haskell'03, August 25, 2003, Uppsala, Sweden.

computing. We invite the reader to consult classical introductory papers [29, 23] for additional background information on the concepts and operations introduced in this paper.

2. QUANTUM DATA

The building blocks of quantum computing are *qubits* or quantum bits. After explaining qubits as a datatype similar to the classical *Bool* type, we generalize the construction to other datatypes.

2.1 Enumerated Types

In Haskell the boolean datatype and constructors are defined as follows:

$$\textbf{data}\ Bool\ =\ False \mid True$$

A value of type *Bool* can be either *False* or *True* but never both at the same time. In contrast qubits or quantum booleans, whose type we denote $QV\ Bool$, are values of the following general form:

$$\alpha\quad |False\rangle\quad +\quad \beta\quad |True\rangle$$

where α and β are complex numbers representing *probability amplitudes*, each constructor c is interpreted as a *unit vector* $|c\rangle$, and + is vector addition. Such a quantum boolean value is somehow both *False* and *True* at the same time and this *superposition* can be exploited in quantum computations. For example, the superposition could be exploited to explore two alternative paths of a computation in parallel. The use of complex numbers to represent the probability amplitudes means that probability amplitudes have a phase and hence can reinforce each other or cancel each other during intermediate computations. Ultimately however the only observables are still just *False* and *True*. The probability of observing *False* or *True* is proportional to the square of the magnitude of α and β respectively.

Generalizing from booleans to arbitrary types a and their quantum versions $QV\ a$, we observe the following:

- All the constructors for the type a will be interpreted as unit vectors from which we can build quantum values. For convenience, the list of unit vectors is an overloaded operator for each type of interest:

  $$\textbf{class}\ (Eq\ a,\ Ord\ a)\ \Rightarrow\ Basis\ a\ \textbf{where}$$
  $$\quad basis\ ::\ [a]$$

  We must be able to distinguish the unit vectors from each other hence we require that the type a be a member of the *Eq* class. We also require that the unit vectors come associated with an arbitrary but fixed order (*i.e.*, that the type a be a member of the *Ord* class) in order to use the *FiniteMap* library below. Here are simple examples:

  $$\textbf{data}\ Move\ =\ Vertical \mid Horizontal$$
  $$\textbf{data}\ Rotation\ =\ CtrClockwise \mid Clockwise$$
  $$\textbf{data}\ Color\ =\ Red \mid Yellow \mid Blue$$

  $$\textbf{instance}\ Basis\ Bool\ \textbf{where}$$
  $$\quad basis\ =\ [False, True]$$
  $$\textbf{instance}\ Basis\ Move\ \textbf{where}$$
  $$\quad basis\ =\ [Vertical, Horizontal]$$
  $$\textbf{instance}\ Basis\ Rotation\ \textbf{where}$$
  $$\quad basis\ =\ [CtrClockwise, Clockwise]$$
  $$\textbf{instance}\ Basis\ Color\ \textbf{where}$$
  $$\quad basis\ =\ [Red, Yellow, Blue]$$

- Given the unit vectors for type a, values of type $QV\ a$ are maps which associate each unit vector with a probability amplitude. In many articles on quantum computing, the idendity of the unit vectors and their order are kept implicit and quantum values are represented using just a sequence of probability amplitudes. Although this appears convenient, it does not scale well. Our representation of quantum values will therefore consist of an explicit map from each unit vector to its probability amplitude. Abstractly speaking this mapping could be realized using a function but an early prototype of this idea introduced a massive performance penalty to the point where even some of the simplest examples could not be simulated. The problem is that, unless functions are memoized, each function call re-calculates the mapping. Even in a small quantum computing example, the functions would typically be nested to several levels and the exponential slowdown is unacceptable. Instead we realize the mapping using the *FiniteMap* library (which assumes that the type a is an instance of the *Ord* class as we require):

  $$\textbf{type}\ PA\ =\ Complex\ Double$$
  $$\textbf{type}\ QV\ a\ =\ FiniteMap\ a\ PA$$

  As a convention unit vectors associated with a zero probability amplitude will often be omitted from the finite map. The function *pr* returns the probability amplitude associated with a given unit vector:

  $$qv\ ::\ Basis\ a\ \Rightarrow\ [(a, PA)]\ \rightarrow\ QV\ a$$
  $$qv\ =\ listToFM$$

  $$pr\ ::\ Basis\ a\ \Rightarrow\ FiniteMap\ a\ PA\ \rightarrow\ a\ \rightarrow\ PA$$
  $$pr\ fm\ k\ =\ lookupWithDefaultFM\ fm\ 0\ k$$

  Here are some simple examples:

  $$qFalse,\ qTrue,\ qFT\ ::\ QV\ Bool$$
  $$qFalse\ =\ unitFM\ False\ 1$$
  $$qTrue\ =\ unitFM\ True\ 1$$
  $$qFT\ =\ qv\ [(False, 1/\sqrt{2}),\ (True, 1/\sqrt{2})]$$

  $$qUp\ ::\ QV\ Move$$
  $$qUp\ =\ unitFM\ Vertical\ 1$$

  The value *qFalse* is a quantum boolean value which is always *False*; similarly the value *qTrue* is a quantum value which is always *True*. The value *qFT* is "half-*False*" and "half-*True*." The value *qUp* is a quantum value which is always *Vertical*. The normalization factor of $1/\sqrt{2}$ in *qFT* is not computationally significant and will often be omitted.

2.2 Infinite Types

In principle it is possible to extend the ideas in the previous section to infinite datatypes such as the natural numbers:

$$\textbf{instance}\ Basis\ Integer\ \textbf{where}$$
$$\quad basis\ =\ [0..]$$

A "good" quantum value of type $QV\ Integer$ would associate a non-zero probability amplitude to just a few unit vectors, or if it associates non-zero probability amplitudes to all unit vectors, the amplitudes should "vanish quickly enough" like:

$$qi :: QV\ Integer$$
$$qi = qv\ [\,(i,\ 1\ /\ i)\ |\ i \leftarrow basis,\ i \neq 0]$$

But other than being able to express quantum values such as qi, we can do little with them, at least if we are to keep the presentation and code reasonably simple. As will be apparent in Sections 3 and 4, we will need to perform *strict* operations like addition on the probability amplitudes associated with all unit vectors. When the number of unit vectors is finite, this can be done with the Haskell primitive +; when the number of unit vectors is infinite like in the case of *Integer*, we would need to manipulate convergent series and integrals instead of additions and summations. We do not deal with infinite datatypes in the rest of this report.

2.3 Pairs

Given two quantum values of type $QV\ a$ and $QV\ b$, we can build two kinds of pairs: one of type $(QV\ a,\ QV\ b)$ and one of type $QV(a, b)$. The first kind of pair is nothing special: quantum values are like any other value in that they can be stored in data structures. The second kind of pair is an instance of a more general notion of *entangled* values which have several new properties with no counterpart in the classical case and hence requires a careful study.

First, given the basis for pairs:

$$\textbf{instance}\ (Basis\ a,\ Basis\ b) \Rightarrow Basis\ (a, b)$$
$$\quad \textbf{where}\ basis = [\,(a, b)\ |\ a \leftarrow basis,\ b \leftarrow basis\,]$$

we can write some examples:

$$p_1,\ p_2,\ p_3 :: QV\ (Bool, Bool)$$

$$p_1 = qv\ [((False, False), 1),\ ((False, True), 1)]$$

$$p_2 = qv\ [((False, False), 1),\ ((True, True), 1)]$$

$$p_3 = qv\ [((False, False), 1),\ ((False, True), 1),\ ((True, False), 1),\ ((True, True), 1)]$$

The first component of the quantum pair p_1 is always *False* and the second is either *False* or *True* with equal probability. This can be formalized by saying that the pair is equivalent to the *tensor product* of the values *qFalse* and qFT. The tensor product is generally defined as follows:

$$(\&*) :: (Basis\ a,\ Basis\ b) \Rightarrow QV\ a \rightarrow QV\ b \rightarrow QV\ (a, b)$$
$$qa\ \&*\ qb =$$
$$\quad qv\ [\,((a, b),\ pr\ qa\ a\ *\ pr\ qb\ b)\ |\ (a, b) \leftarrow basis\,]$$

The fact that the two components of the pair p_1 can be separated into two values is not a general property of quantum pairs. Indeed the components of the pair p_2 cannot be separated. The intuitive reason is simple: each component of the pair p_2 can be *False* or *True* with equal probability, which suggests that the pair might be equal to $qFT\ \&*\ qFT$. But of course this tensor product produces p_3 which is rather different from p_2. The components of a pair like p_2 that cannot be separated are *entangled*.

The situation for pairs generalizes to other structured datatypes which can also be entangled. Entangled values are a fundamental aspect of quantum computing and will be revisited in more detail in later sections. We immediately note however that entanglement implies that reasoning about quantum system is non-compositional:

> A surprising and unintuitive aspect of the state space of an n-particle quantum system is that the state of the system cannot always be described in terms of the state of its component pieces. [23, p.308]

3. FUNCTIONS/OPERATIONS

In the classical situation, the only non-trivial unary operation on booleans is the function *not* defined as follows:

$$not\ False = True$$
$$not\ True = False$$

The corresponding function on quantum boolean values maps the general value $(\alpha\ \ |False\rangle + \beta\ \ |True\rangle)$ to $(\beta\ \ |False\rangle + \alpha\ \ |True\rangle)$. It can be defined as follows:

$$qnot_f :: QV\ Bool \rightarrow QV\ Bool$$
$$qnot_f\ v = qv\ [(False,\ pr\ v\ True),\ (True,\ pr\ v\ False)]$$

It is easy to calculate that $qnot_f\ qFalse$ evaluates to $qTrue$, and vice-versa. Naturally $qnot_f$ can also be applied to mixed values like qFT.

Because of the richer structure of quantum booleans one can define many more functions other than the simple $qnot_f$. The $hadamard_f$ function below maps a general value of the form $(\alpha\ |False\rangle + \beta\ |True\rangle)$ to $((\alpha + \beta)\ |False\rangle + (\alpha - \beta)\ |True\rangle)$. This operation can be defined as follows:

$$hadamard_f :: QV\ Bool \rightarrow QV\ Bool$$
$$hadamard_f\ v =$$
$$\quad \textbf{let}\ \alpha = pr\ v\ False$$
$$\quad\quad \beta = pr\ v\ True$$
$$\quad \textbf{in}\ qv\ [(False,\ \alpha + \beta),\ (True,\ \alpha - \beta)]$$

A simple calculation shows that $hadamard_f\ qFalse$ evaluates to qFT.

It should be clear at this point that there is a general pattern for all operations on quantum data. The output value has a probability amplitude associated with each one of its unit vectors; and each of these amplitudes may depend on the probability amplitudes for all the unit vectors of the input value. In other words an operation on quantum data is completely specified by a matrix which specifies how each input probability amplitude contributes to each output probability amplitude. We represent this matrix by another finite map:

$$\textbf{data}\ Qop\ a\ b = Qop\ (FiniteMap\ (a, b)\ PA)$$

$$qop :: (Basis\ a,\ Basis\ b) \Rightarrow [((a, b), PA)] \rightarrow Qop\ a\ b$$
$$qop = Qop\ .\ listToFM$$

To apply an operation to a quantum value we multiply the matrix by the vector representing the value:

$$qApp :: (Basis\ a,\ Basis\ b) \Rightarrow Qop\ a\ b \rightarrow QV\ a \rightarrow QV\ b$$

```
qApp (Qop m) v =
  let bF b = sum [ pr m (a, b) * pr v a | a ← basis]
  in qv [(b, bF b) | b ← basis ]
```

For example the $qnot_f$ and $hadamard_f$ operations mentioned above can be defined using the following matrices:

$$
\begin{aligned}
qnot_{op} &= qop\ [((False, True), 1),\\
&\qquad ((True, False), 1)]\\
hadamard_{op} &= qop\ [((False, False), 1),\\
&\qquad ((False, True), 1),\\
&\qquad ((True, False), 1),\\
&\qquad ((True, True), -1)]
\end{aligned}
$$

As another example, the following operator translates from the vertical/horizontal polarization states of a photon to the clockwise/counter-clockwise polarization states [19]:

$$
\begin{aligned}
&m2r :: Qop\ Move\ Rotation\\
&m2r = qop\ [((Vertical, CtrClockwise), 1),\\
&\qquad ((Vertical, Clockwise), 1),\\
&\qquad ((Horizontal, CtrClockwise), 0 :+ -1),\\
&\qquad ((Horizontal, Clockwise), 0 :+ 1)]
\end{aligned}
$$

The notation $a :+ b$ is Haskell's syntax for the complex number $a + ib$.

3.1 Lifting

To further understand the nature of quantum operations, we briefly discuss a way to lift classical functions to operations on quantum values. The basic idea is simple: an input unit vector contributes to an output unit vector if and only if the classical function relates the corresponding constructors:

$$
\begin{aligned}
&opLift :: (Basis\ a, Basis\ b) \Rightarrow (a \to b) \to Qop\ a\ b\\
&opLift\ f = qop\ [((a, f\ a), 1) \mid a \leftarrow basis]
\end{aligned}
$$

However, this is not always sensible as all quantum operations must be *unitary* (*i.e.*, the operation is invertible and when viewed as a matrix the inverse of the operation is just the conjugate transpose of the matrix). In the special case where f is a *reversible* function, the above construction works and produces what is called a *pseudo-classical operator*. For example, the function *not* is reversible and the above construction indeed produces $qnot_{op}$.

Other classical functions like *and* are not reversible: from an output *False*, one cannot calculate the two inputs of the *and* function. However a non-reversible function like *and* can be trivially made reversible by adding additional outputs which transfer the inputs:

$$reversible_{and}\ a\ b = (a, b, a\ \&\&\ b)$$

In general any classical computation, no matter how complex, can be made reversible by remembering enough of its intermediate results. Bennett shows how a universal Turing Machine can be simulated by a reversible Turing Machine [4]. The idea can also be adapted to abstract machines like the SECD machine [17] and optimized beyond the naïve requirement of storing every intermediate value [5]. Reversible computation is also an interesting topic with its own merits [11, 1].

3.2 Producing Entangled Pairs

Controlled operations are the most common way of introducing entanglement in quantum systems. The simplest such operation is the controlled-not (*cnot*) operation; *cnot* takes two quantum boolean values and:

- does nothing if the first value (called the control value) is *False*, and
- negates the second value (called the target value) otherwise.

This seems simple enough until we remember that the control qubit can be simultaneously *False* and *True*. In this case the operation performs both actions simultaneously and the resulting pair of qubits is entangled. For example consider the situation when the control qubit is *qFT* and the target qubit is *qFalse*. Since the control qubit is *False* with a non-zero probability, a possible output of the operation is the state (*False*, *False*). Also since the control qubit is *True* with a non-zero probability, a possible output of the operation is the state (*True*, *True*). No other outputs are possible. In other words applying the *cnot* operation to *qFT* and *qFalse* produces the entangled pair p_2 of Section 2.3.

More generally, the operation performed on the second value need not be negation, and the control value need not be a boolean. We abstract from these two situations to define a generic controlled operation which takes two arguments: a quantum operator u which might be applied to the target qubit, and a classical function *enable* which decides for each control value a whether it should enable the application of the operation u to the target:

```
cop :: (Basis a, Basis b) ⇒
       (a → Bool) → Qop b b → Qop (a, b) (a, b)
cop enable (Qop u) =
  qop ([ (((a, b), (a, b)), 1) |
         (a, b) ← basis, not (enable a)] ++
       [ (((a, b1), (a, b2)), pr u (b1, b2)) |
         a ← basis, enable a,
         b1 ← basis, b2 ← basis])
```

If the input control value is not enabled then the output pair is identical to the input pair (first group); otherwise if the input control value is enabled and identical to the output control value, then the contribution is the one given by the operator u. In all others cases, the output probability is zero and hence omitted following our convention.

The *cnot* operation is easily obtainable from the generic operation.

```
cnot :: Qop (Bool, Bool) (Bool, Bool)
cnot = cop id qnot_op
```

Another common controlled operation, is the controlled-controlled-not operation, also called the *toffoli* operation [23]. The *toffoli* operation is essentially identical to the *cnot* operation but is controlled by a pair of boolean values. Its definition is almost identical to *cnot*:

```
toffoli :: Qop ((Bool, Bool), Bool) ((Bool, Bool), Bool)
toffoli = cop (uncurry (&&)) qnot_op
```

The *toffoli* operation is significant because it can implement all classical boolean operations. When both control values are *True* the operation negates the target value. If the target value is *False* the operation performs a logical *and* of the control values. Since it can implement *and* and *not*, the operation can implement any boolean function.

4. MEASUREMENT

Values whether resulting from a classical computation or from a quantum computation must ultimately be observed to communicate results to the outside world. In a classical programming model, the process of observing a value simply returns the value. In the quantum model, the process of observation is more complicated.

4.1 Normalization

So far, we have not imposed any restrictions on the probability amplitudes associated with the unit vectors of a quantum value. This is valid since the magnitude of the vectors is of no computational relevance. It is however useful to have a normalized representation before discussing the details of measurement. Normalization simply consists of dividing each probability amplitude by the *norm* of the value. (In the code below we use $|\ |^2$ to refer to the function (*square . magnitude*).)

$$
\begin{array}{l}
\mathit{normalize} :: \mathit{Basis}\ a \Rightarrow \mathit{QV}\ a \rightarrow \mathit{QV}\ a \\
\mathit{normalize}\ v = (1 \,/\, \mathit{norm}\ v :+ 0) *> v \\
\\
\mathit{norm} :: \mathit{Basis}\ a \Rightarrow \mathit{QV}\ a \rightarrow \mathit{Double} \\
\mathit{norm}\ v = \textbf{let}\ \mathit{probs} = \mathit{map}\ |\ |^2\ (\mathit{eltsFM}\ v) \\
\qquad\qquad \textbf{in}\ \sqrt{\mathit{sum}\ \mathit{probs}} \\
(*>) :: \mathit{Basis}\ a \Rightarrow \mathit{PA} \rightarrow \mathit{QV}\ a \rightarrow \mathit{QV}\ a \\
c *> v = \mathit{mapFM}\ (\backslash \_\ a \rightarrow c * a)\ v
\end{array}
$$

For example, normalizing p_1, p_2, and p_3 produces:

$$
\begin{array}{rcl}
np_1 & = & qv\ [((\mathit{False}, \mathit{False}), 1/\sqrt{2}), \\
& & \quad ((\mathit{False}, \mathit{True}), 1/\sqrt{2})] \\
np_2 & = & qv\ [((\mathit{False}, \mathit{False}), 1/\sqrt{2}), \\
& & \quad ((\mathit{True}, \mathit{True}), 1/\sqrt{2})] \\
np_3 & = & qv\ [((\mathit{False}, \mathit{False}), 1/2), \\
& & \quad ((\mathit{False}, \mathit{True}), 1/2), \\
& & \quad ((\mathit{True}, \mathit{False}), 1/2), \\
& & \quad ((\mathit{True}, \mathit{True}), 1/2)]
\end{array}
$$

4.2 Observing Simple Values

Let q be a normalized quantum boolean value $(\alpha\ |\mathit{False}\rangle + \beta\ |\mathit{True}\rangle)$ where $|\alpha|^2 + |\beta|^2 = 1$. A measurement of q:

- returns a result *res* which is either *False* with probability $|\alpha|^2$ or *True* with probability $|\beta|^2$;
- as a *side-effect* updates q so that all future observations return *res*.

Thus as soon as the value q is observed, any superposition of *False* and *True* that might have been present vanishes, and the value becomes either a pure *False* or a pure *True*.

4.3 Observation and Entanglement

Given a pair of type $QV\ (a, b)$, quantum mechanics permits three measurements: a measurement of the state of the pair itself (both components are measured at once); or a measurement in which either the left component or the right component (but not both) are measured.

In some sense, it is rather strange that one can operate on one of the components of an entangled pair individually *even* if this component cannot be separated from the other one. In fact, the process of observation provides another way to understand entanglement. Two values are entangled if observing one affects the measurement of the other. Looking back at our examples from Section 2.3, we had decided that the components of np_3 are not entangled and that the components of np_2 are entangled. Indeed:

- Each component of np_3 is *False* and *True* with equal probability hence observing the first component can return *False* or *True*. If it returns *False*, the pair is "updated" (the wave function collapses) to be consistent with this observation and becomes:

  $qv\ [((\mathit{False}, \mathit{False}), 1/\sqrt{2}),\ ((\mathit{False}, \mathit{True}), 1/\sqrt{2})]$

  A future observation of the second component is still equally likely to be *False* or *True*.

- In contrast, even though each component of np_2 is *False* and *True* with equal probability, the values are correlated. Observing the first component can return *False* or *True*. If it returns *False*, the pair is updated to be:

  $qv\ [((\mathit{False}, \mathit{False}), 1)]$

  and any future observation of the second component *must* now return *False*.

4.4 The EPR Paradox

Quantum mechanics describes the phenomena of entanglement and observation without interpretation:

> It is important to notice that there is no mechanism postulated in this theory for how a wave function is sent into an eigenstate by an observable. Just as mathematical logic need not demand causality behind an implication between propositions, the logic of quantum mechanics does not demand a specified cause behind an observation... Nevertheless, the debate over the interpretation of quantum theory has often led its participants into asserting that causality has been demolished in physics. [16, p.6]

If we are to provide an operational model of quantum computing, we would need some interpretation of quantum mechanics to explain *how* the second component of a pair is affected when the first component is observed. To understand the difficulties, it is useful to review the famous Einstein, Podolsky, and Rosen [9] paradox and some of the attempts at resolving it.

Einstein, Podolsky, and Rosen [9] proposed a gedanken experiment that uses entangled values in a manner that seemed to violate fundamental principles of relativity. The question is the following: when one component of a pair of entangled values is observed, *how* does the information about the observed value flow to the other component, if indeed there is any information flow in the first place! There are two standard attempts at resolving the paradox:

1. The first attempted explanation is that each component of the pair has a *local state* which determines its observed value. Before observation, the local state is hidden and can only be described probabilistically. As soon as the component is observed the hidden state is exposed. In the case of the pair np_2 above the

local hidden state of each component might be *False*; the components can then be observed in any order, and without any communication or interaction, both observations will be equal as expected. If valid, this idea would yield a simple and completely local computational model for quantum computing. Unfortunately Bell formulates this idea mathematically and shows it to be incompatible with the statistical predictions of quantum mechanics [2]. Bell concludes that any theory based on hidden variables and which is consistent with the statistical predictions of quantum mechanics must also include a mechanism whereby the setting of one measuring device can influence the reading of another instrument, however remote, and that the signal between them must propagate instantaneously. This violates special relativity.

2. The other attempted explanation is closely related to the above: the value of each component is a function of the measured value of the other component. Whichever component is measured first communicates its value to the other component which updates its value. But as Einstein, Podolsky, and Rosen noticed, this explanation also violates the principles of special relativity. The notion of one component being measured "first" is not a well-defined notion as it depends on the speed of the agent observing the measurement. In other words, it is possible that one observer sees that the left component has been measured first, while another observer sees that it is the right component that has been measured first. In summary, the idea of communicating a value from the first component to be measured to the second component cannot be compatible with both observers, yet experiments are invariant under change of observer.

Unfortunately even though these two explanations are known to be wrong there aren't really any other widely-accepted explanations. There are however several interesting interpretations which should be investigated in more depth as they would provide interesting operational models of quantum computing: in particular two appealing interpretations are the many-worlds interpretation in which all possible observations are realized in parallel universes [10], and the transactional interpretation in which computation is described as the fixed point of a process happening in both forward-time and reverse-time [6].

For our purposes, we adopt the simplest operational mechanism for observing components of entangled data structures such that the result of the observation affects all other entangled values: we use a global side-effect to a shared reference. The communication among the entangled values happens implicitly and instantaneously via the assignment to the shared reference. Even though this may not be sensible from a physical perspective, it appears reasonable in a single-threaded programming environment. In the presence of multiple threads (which we do not consider), a problem reminiscent of the one noted by Einstein, Podolsky, and Rosen can occur in the form of race conditions if two threads attempt to measure different components of the pair simultaneously. It remains to be seen whether the use of global side-effects in our model can cause quantum computing simulations to deliver results and effects that do not correspond to physical counterparts.

4.5 References to Quantum Values

To model the side-effects implicit in the observation process, we will use explicit references: quantum values can only be accessed via a reference cell; the observation updates the reference cell with the observed value:

```
data QR a = QR (IORef (QV a))

mkQR :: QV a → IO (QR a)
mkQR v = do r ← newIORef v
            return (QR r)
```

The function *mkQR* is an IO-action which when executed allocates a new reference cell and stores the given quantum value in it.

To observe a quantum value accessible via a reference $QR\ a$, we read the contents reference, observe the value, and update the reference with the result of the observation. Observing a value requires the following steps. First we normalize the value. Then we calculate the probability associated with each unit vector in the basis. For each unit vector, we also compute a cumulative probability which is the sum of its probability and all the probabilities of the unit vectors before it in the (arbitrary since it is irrelevant) order given by the basis. Since the probabilities add to 1, we choose a random number between 0 and 1 and choose the first constructor with a cumulative probability that exceeds this random number:

```
observeR :: Basis a ⇒ QR a → IO a
observeR (QR ptr) =
  do v ← readIORef ptr
     res ← observeV v
     writeIORef ptr (unitFM res 1)
     return res

observeV :: Basis a ⇒ QV a → IO a
observeV v =
  do let nv    = normalize v
         probs = map (| |^2 . pr nv) basis
     r ← getStdRandom (randomR (0.0, 1.0))
     let cPsCs = zip (scanl1 (+) probs) basis
         Just(_, res) = find (\(p, _) → r < p) cPsCs
     return res
```

For example, each evaluation of *test* below prints either three occurrences of *False* or three occurrences of *True*: the first observation is equally likely to be *False* or *True* but once it is performed it fixes the results of the next two observations:

```
test = do x ← mkQR qFT
          o1 ← observeR x
          o2 ← observeR x
          o3 ← observeR x
          print (o1, o2, o3)
```

The observation of one of the components of a pair is slightly more complicated. We only show the case for observing the left component of the pair; the other case is symmetric:

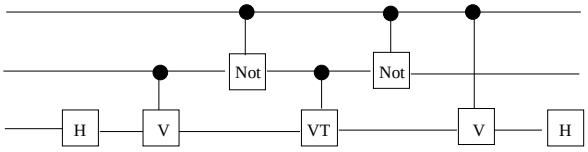
```
observeLeft :: (Basis a, Basis b) ⇒
               QR (a, b) → IO a
observeLeft (QR ptr) =
```

```
do v ← readIORef ptr
   let leftF a = √(sum [ |pr v (a, b)|² | b ← basis])
       leftV = qv [(a, leftF a) | a ← basis]
   aobs ← observeV leftV
   let nv = qv [((aobs, b), pr v (aobs, b)) |
                b ← basis]
   writeIORef ptr (normalize nv)
   return aobs
```

We first build a *virtual* quantum value *leftV* which gives the probability associated with each unit vector of the left component. This probability is calculated by summing over all occurrences of this unit vector in the pair. The virtual value is observed and this selects one of the unit vectors. The pair is reconstituted with only the components that are consistent with the observation, and the result is stored in the reference cell.

5. WAVE/PARTICLE DUALITY

We have in principle covered the basics of quantum computing and can move on to some examples. An elementary example that we consider is to model this alternative implementation of the *toffoli* operation:



The circuit diagram uses the de-facto standard notation for specifying quantum computations. The convention is that the values flow from left to right in steps corresponding to the alignment of the gates. For the remainder of this discussion we refer to the three relevant qubits as *top*, *middle*, and *bottom*:

1. In the first step, the *hadamard* operation is applied to the *bottom* qubit.

2. In the second step, a controlled-v_{op} (which is defined below) is applied to the pair consisting of the *middle* and *bottom* qubits:

   ```
   v_op :: Qop Bool Bool
   v_op = qop [((False, False), 1),
               ((True, True), 0 :+ 1)]
   ```

3. In the third step, the controlled operation *cnot* is applied to the pair consisting of the *top* and *middle* qubits.

4. In the fourth step, a controlled operation-vt_{op} (the adjoint or conjugate transpose of v_{op} defined below) is applied to the pair consisting of the *middle* and *bottom* qubits:

   ```
   vt_op :: Qop Bool Bool
   vt_op = qop [((False, False), 1),
                ((True, True), 0 :+ -1)]
   ```

5. The fifth step is identical to the third step.

6. In the sixth step, a controlled-v_{op} is applied to the pair consisting of the *top* and *bottom* qubits. qubits.

7. Finally in the last step, the *hadamard* operation is applied to the *bottom* qubit.

Implementing this rather elementary circuit is complicated by the fact that the three qubits *top*, *middle*, and *bottom* are generally entangled. It is not possible to directly manipulate just the *bottom* qubit as required by the first step for example. Even worse, the circuit requires us to apply operations to three distinct pairs of qubits: (*middle*, *bottom*), (*top*, *middle*), and (*top*, *bottom*) which again, by definition of entanglement, cannot be isolated to suit each operation. This situation is the programming counterpart of the wave/particle duality: on one hand the three entangled values form a connected "wave"; on the other hand each of them is an independent "particle" which can be operated upon individually with the understanding that result of such an operation affects the entire wave.

The naïve way of modeling computations such as the one above is to define specialized functions that operate on components of data-structures similar to our *observeLeft* function of Section 4.5. This gets quickly out of hand and several quantum computing models try to provide a general mechanism to deal with this problem. For example, Selinger includes operations which perform arbitrary *permutations* of the variables [24], and QCL [22] includes the notion of a *symbolic register* which can refer to any collection of qubits even if they are part of entangled structures. In our case we propose a related idea of *virtual values*.

5.1 Virtual Values and Adaptors

A virtual value is a value which although possibly embedded deep inside a structure and entangled with others can be operated on individually. A virtual value is specified by giving the entire data structure to which it belongs and an *adaptor* which specifies the mapping from the entire data structure to the value in question and back. More specifically, we have:

```
data Adaptor l g =
  Adaptor { dec :: g → l, cmp :: l → g }

data Virt a na u = Virt (QR u) (Adaptor (a, na) u)
```

The type (*Virt a na u*) defines a virtual value of type a which is entangled with values of type na. The type u is the type of the entire data structure which contains both a and na. The adaptor maps back and forth between the type u and its decomposition. Virtual values are related to composable references [14] which provide access to a field or a substructure relative to a larger tuple or record used as a state.

For example, in a data structure of type

$$QV\ (((a, b, c), (d, e)), (f, g))$$

there are several ways to to isolate a quantum value of type $QV(d, g)$ depending on how one decides to group the other values with each d and g are entangled. Two possible ways are:

```
mkVirt₁ :: QR (((a, b, c), (d, e)), (f, g)) →
           Virt (d, g)
                (a, b, c, e, f)
                (((a, b, c), (d, e)), (f, g))
mkVirt₁ r = Virt r a₁
  where a₁ =
```

```
  Adaptor { dec =\ (((a, b, c), (d, e)), (f, g)) →
                   ((d, g), (a, b, c, e, f)),
            cmp =\ ((d, g), (a, b, c, e, f)) →
                   (((a, b, c), (d, e)), (f, g)) }

mkVirt2 :: QR (((a, b, c), (d, e)), (f, g)) →
           Virt (d, g)
                ((a, b, c), e, f)
                (((a, b, c), (d, e)), (f, g))
mkVirt2 r = Virt r a2
  where a2 =
    Adaptor { dec =\ (((a, b, c), (d, e)), (f, g)) →
                     ((d, g), ((a, b, c), e, f)),
              cmp =\ ((d, g), ((a, b, c), e, f)) →
                     (((a, b, c), (d, e)), (f, g)) }
```

The mechanism of virtual values allows us to pretend there is a pair of type (d, g) in the structure even though the type (d, g) does not occur directly in the type of the structure and the components of type d and g are deeply nested. This is reminiscent of the Façade pattern [12] in which a deeply nested structure is given a flat interface which gives access to its internal references.

5.2 Generating Adaptors

The definition of adaptors (at least for data structures like tuples) is so regular that we should be able to automate their generation from just the type information. We assume in the remainder of this article that the following adaptors have been generated. We only give the definitions for the first two:

```
ad_pair1 :: Adaptor (a1, a2) (a1, a2)
ad_pair1 = Adaptor { dec = \ (a1, a2) → (a1, a2),
                     cmp = \ (a1, a2) → (a1, a2) }
ad_pair2 :: Adaptor (a2, a1) (a1, a2)
ad_pair2 = Adaptor { dec = \ (a1, a2) → (a2, a1),
                     cmp = \ (a2, a1) → (a1, a2) }
ad_triple23 ...
ad_triple12 ...
ad_triple13 ...
```

5.3 Everything is a Virtual Value

To provide a uniform model, we rephrase all our operations in terms of virtual values. First we provide a way of converting individuals references to quantum values to trivial virtual values, and a way of creating virtual values from other virtual values by composing a new adaptor:

```
virtFromR :: QR a → Virt a () a
virtFromR r =
  Virt r (Adaptor { dec = \ a → (a, ()),
                    cmp = \ (a, ()) → a })

virtFromV :: Virt a na u → Adaptor (a1, a2) a →
             Virt a1 (a2, na) u
virtFromV
  (Virt r (Adaptor { dec = gdec, cmp = gcmp }))
  (Adaptor {dec = ldec, cmp = lcmp }) =
  Virt r
    (Adaptor { dec = \ u → let (a, na) = gdec u
                               (a1, a2) = ldec a
                           in (a1, (a2, na)),
               cmp = \ (a1, (a2, na)) →
                     gcmp (lcmp(a1, a2), na)})
```

An operation on quantum values was previously given the type *Qop ab* denoting the fact that it maps quantum values of type *QV a* to quantum values of type *QV b*. Instead of simple quantum values as before, the input and output values are now virtual, *i.e.*, they are of type *Virt a na ua* and *Virt b nb ub*. The operation of type *Qop a b* should still make sense as the input and output values are of the right type except that they are entangled in larger structures. The application does not however affect these surrounding entangled values which should therefore have the same type. Hence the general application is defined as follows:

```
app :: (Basis a, Basis b, Basis nab,
        Basis ua, Basis ub) ⇒
       Qop a b → Virt a nab ua → Virt b nab ub →
       IO ()
app (Qop f)
    (Virt (QR ra)
          (Adaptor { dec = deca, cmp = cmpa }))
    (Virt (QR rb)
          (Adaptor { dec = decb, cmp = cmpb })) =
  let gf = qop
           [((ua, ub), pr f (a, b)) |
            ua ← basis, ub ← basis,
            let (a, na) = deca ua,
            let (b, nb) = decb ub,
            na == nb]
  in do fa ← readIORef ra
        let fb = normalize $ qApp gf fa
        writeIORef rb fb
```

The first argument is the operation to apply. The next two arguments are the input and output virtual values which share the same entangled neighbors. The operation is promoted from something acting on a and b to something acting on the entire entangled structure in the expected way.

In general the input and output virtual values can be different. For example, given a virtual value

```
ip :: Virt Move Bool (Move, Bool)
```

and a virtual value

```
op :: Virt Rotation Bool (Bool, Rotation)
```

we can use *app m2r ip op* to translate from one polarization state of a photon to another even if when the photon is entangled with some qubit.

It is more common in simple examples to have just one global reference to a quantum value of type *QV u* which is repeatedly updated in place by successive operations. Each one of the successive operations is of type *Qop a a* for some type a which can be extracted from u via an adaptor. For these applications, we can use the following simpler version of *app*:

```
app1 :: (Basis a, Basis na, Basis ua) ⇒
        Qop a a → Virt a na ua → IO ()
app1 f v = app f v v
```

A virtual value can be observed using an idea that generalizes *observeLeft* from Section 4.5 using the adaptor to

decompose and compose the value instead of the built-in knowledge that we are manipulating the left component of a pair:

```
observeVV :: (Basis a, Basis na, Basis u) ⇒
             Virt a na u → IO a
observeVV (Virt (QR r)
          (Adaptor { dec = dec, cmp = cmp })) =
do v ← readIORef r
   let virtF a = √(sum [ | pr v (cmp(a, na))|² |
                         na ← basis ])
   let virtV = qv [(a, virtF a) | a ← basis]
   aobs ← observeV virtV
   let nv = qv [(u, pr v (cmp(aobs, na))) |
                u ← basis,
                let (a, na) = dec u,
                a == aobs]
   writeIORef r (normalize nv)
   return aobs
```

6. EXAMPLES

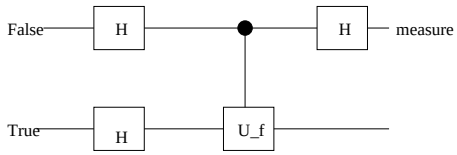
The machinery we have developed may appear quite heavy but it is quite powerful and makes programming circuit diagrams like the *toffoli* example in Section 5 quite simple. The complete code (excluding the adaptors) is:

```
toffoli' :: (Basis na, Basis u) ⇒
            Virt (Bool, Bool, Bool) na u → IO ()
toffoli' vtriple =
  let b = virtFromV vtriple ad_triple3
      mb = virtFromV vtriple ad_triple23
      tm = virtFromV vtriple ad_triple12
      tb = virtFromV vtriple ad_triple13
      cv = cop id v_op
      cvt = cop id vt_op
  in do app1 hadamard_op b
        app1 cv mb
        app1 cnot tm
        app1 cvt mb
        app1 cnot tm
        app1 cv tb
        app1 hadamard_op b
```

Given any three quantum boolean values (entangled or not; part of a larger data structure or not), we begin by isolating the relevant parts and then simply apply the operations in the obvious way: one line for each step in the circuit. We use the mnemonics *mb* to refer to the pair of the middle and bottom values, *tm* to refer to the pair of the top and middle values, etc.

6.1 The Deutsch Oracle

Another interesting example is the Deutsch oracle [7] which given a function on booleans decides with only *one* invocation of the function whether the function is balanced (*id* or *not*) or constant (*const True* or *const False*). Of course the Haskell simulation applies the function to both *True* and *False* but a real quantum implementation would apply the function once to the quantum superposition. The example does not really require the machinery of virtual values but it does use the power of the generic controlled operation of Section 3.2:




```
deutsch :: (Bool → Bool) → IO ()
deutsch f =
do inpr ← mkQR (qFalse &* qTrue)
   let both = virtFromR inpr
       top = virtFromV both ad_pair1
       bot = virtFromV both ad_pair2
       uf = cop f qnot_op
   app1 hadamard_op top
   app1 hadamard_op bot
   app1 uf both
   app1 hadamard_op top
   topV ← observeVV top
   putStr (if topV then "Balanced" else "Constant")
```

The oracle works as follows. The top value is transformed by the *hadamard* operation to $|False\rangle + |True\rangle$ and the bottom value is transformed to $|False\rangle - |True\rangle$. There are several cases depending on f:

- If f is *const False*: the control line is always disabled and both top and bottom values are unchanged. The last *hadamard* transforms the top value $|False\rangle + |True\rangle$ to $(|False\rangle + |True\rangle) + (|False\rangle - |True\rangle)$ which simplifies to $|False\rangle$ if we ignore the normalizing factor as usual.

- f is *id*: the control line is a superposition $|False\rangle + |True\rangle$ and the bottom value is both left unchanged and negated in a way that is entangled with the top value. More precisely, the resulting pair of top and bottom values is:

  $$|(False, False)\rangle - |(False, True)\rangle + |(True, True)\rangle - |(True, False)\rangle$$

  which can be explained as follows. The first two components correspond to the cases in which the top value is *False*; since f *False* is also *False*, the control line is disabled, and the bottom value is $|False\rangle - |True\rangle$. The last two cases correspond to the cases in which the top value is *True*; since f *True* is also *True*, the control line is enabled, and the bottom value becomes $|True\rangle - |False\rangle$. Finally the top value is operated on by *hadamard* while leaving the bottom value intact. This produces:

  $$\begin{aligned}&|(False, False)\rangle + |(True, False)\rangle\\ &- |(False, True)\rangle - |(True, True)\rangle\\ &+ |(False, True)\rangle - |(True, True)\rangle\\ &- |(False, False)\rangle + |(True, False)\rangle\end{aligned}$$

  which simplifies to: $|(True, False)\rangle - |(True, True)\rangle$ Thus observing the top (left) value always returns *True*.

- The situations in which f is *const True* or *not* are like above. In the case f is *const True* the control line is always enabled and the bottom value is always negated and hence is not entangled with the top value. In the case f is *not* the values are entangled and a similar

analysis shows that the top (left) value evaluates to *True*. Hence if all cases, if the top value is observed to be *False* the function is constant, and if the top value is observed to be *True* the function is balanced.

6.2 Quantum Adder

A 1-bit quantum adder can be defined using Toffoli and controlled-not gates [23]. The main highlights of the code are:

```
adder :: QV Bool → QV Bool → QV Bool → IO ()
adder inc x y =
  let sum = qFalse
      outc = qFalse
      adder_inputs = inc &* x &* y &* sum &* outc
  in do r ← mkQR vals
        let v = virtFromV (virtFromR r) ...
        ...
        app1 toffoli v_xyo
        app1 toffoli v_ixo
        app1 toffoli v_iyo
        app1 cnot v_is
        app1 cnot v_xs
        app1 cnot v_ys
        (sum, out_carry) ← observeR v_so
        print (sum, out_carry)
```

In the code, we have omitted the adaptors. The virtual values named v with subscripts use the following conventions: i refers to the carry-in qubit, x and y refer to the two qubits to add, s refers to the sum qubit, and o refers to the output-carry qubit. Thus v_{ixo} is the virtual value referring to the three qubits: input-carry, first input, and output-carry. The adder can be called with *qFalse qTrue qTrue* in which case it acts like a classical adder and produces (*False*, *True*), but is can also be called with *qFT qFT qFT*.

7. CONCLUSIONS

We have presented a model of quantum computing embedded in Haskell. We hope the model gives good intuitions about quantum computing for programmers. We have used the model to write several other algorithms including Shor's factoring algorithm [26]. For more involved examples than the ones presented here, it is useful to have a type of integers modulo n to allow convenient parameterization of algorithms over the size of input. This can be encoded using type classes [13, 20] but in an awkward way and could perhaps benefit from meta-programming extensions for Haskell [25]. Also, even though we believe that the idea of *virtual values* is the right one, perhaps its current execution leaves much room for improvement.

Our model elicits a fundamental difference between classical programming languages and quantum programming languages. In classical programming language theory, the expressions of the language can be grouped into *introduction constructs* and *elimination constructs* for the type connectives of the language. A quantum programming language can only have *virtual* elimination constructs because, by definition, the elements of an entangled data structure cannot be separated. This restriction leads to an unusual programming style which, we argue, requires new programming primitives.

Our model is distinguished from other work on quantum computing and functional programming as follows. Both Skibinski [27] and Karczmarczuk [15] used Haskell extensively to model the mathematical structures underlying quantum mechanics which is a different and complimentary focus to ours. Skibinski [28] also implemented a Haskell simulator for quantum computing that operates at the "physicist" level of abstraction of qubits and gates which we tried to abstract from by using abstract data types and functions instead.

There have also been several proposals for "quantum programming languages" that do not relate to functional programming [22, 8, 24]. Both pGCL, an imperative language extending Dijkstra guarded-command language [8], and QPL, a functional typed language with quantum data [24], are well-developed and semantically well-founded and could provide interesting links to functional programming.

Acknowledgments

We would like to thank the anonymous reviewers for extensive and very useful comments.

8. REFERENCES

[1] H. G. Baker. NREVERSAL of fortune - the thermodynamics of garbage collection. In Y. Bekkers and J. Cohen, editors, *Memory Management; International Workshop IWMM 92; Proceedings*, pages 507–524, Berlin, Germany, 1992. Springer-Verlag.

[2] J. S. Bell. On the Einstein-Podolsky-Rosen paradox. In *[3]*, pages 14–21. Cambridge University Press, 1987.

[3] J. S. Bell. *Speakable and Unspeakable in Quantum Mechanics*. Cambridge University Press, 1987.

[4] C. H. Bennett. Logical reversibility of computation. *IBM Journal of Research and Development*, 17(6):525–532, Nov. 1973.

[5] H. Buhrman, J. Tromp, and P. Vitányi. Time and space bounds for reversible simulation. *Lecture Notes in Computer Science*, 2076:1017–1027, 2001.

[6] J. G. Cramer. The transactional interpretation of quantum mechanics. *Modern Physics*, 58:647–688, 1986.

[7] D. Deutsch. Quantum theory, the Church-Turing principle and the universal quantum computer. *Proc. Roy. Soc. London, Ser. A*, 400:97–117, 1985.

[8] E. W. Dijkstra. *A Discipline of Programming*, chapter 14. Prentice-Hall, Englewood Cliffs, N. J., 1976.

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

[10] H. Everett, III. "Relative state" formulation of quantum mechanics. *Reviews of Modern Physics*, 29:454, 1957.

[11] M. P. Frank. *Reversibility for Efficient Computing*. PhD thesis, MIT, 1999.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1995.

[13] R. Hinze. Haskell does it with class. Slides of a talk given at the Generic Haskell meeting, May 2001.

[14] K. Kagawa. Mutable data structures and composable references in a pure functional language. In *State in*

Programming Languages (SIPL'95), pages 79–94, Jan. 1995.

[15] J. Karczmarczuk. Structure and interpretation of quantum mechanics — a functional framework. In *ACM SIGPLAN Haskell Workshop*, 2003.

[16] L. H. Kauffman. *Quantum Topology and Quantum Computing*, chapter IV of [18]. American Mathematical Society, 2002.

[17] W. Kluge. A reversible SE(M)CD machine. In *11th International Workshop on the Implementation of Functional Languages, Lochem, The Netherlands, September 7-10, 1999*, number 1868 in Lecture Notes in Computer Science, pages 95–113. Springer-Verlag, Sept. 2000.

[18] S. J. Lomonaco, Jr., editor. *Quantum Computation: A Grand Mathematical Challenge for the Twenty-First Century and the Millennium*, volume 58 of *Proceedings of Symposia in Applied Mathematics*. American Mathematical Society, Mar. 2002.

[19] S. J. Lomonaco, Jr. *A Rosetta Stone for Quantum Mechanics with an Introduction to Quantum Computation*, chapter I of [18]. American Mathematical Society, 2002.

[20] C. McBride. Faking it—simulating dependent types in Haskell. *Journal of Functional Programming*, 12(4&5):375–392, July 2002.

[21] S.-C. Mu and R. Bird. Functional quantum programming. In *Second Asian Workshop on Programming Languages and Systems, KAIST, Korea*, Dec. 2001.

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

[23] E. Rieffel and W. Polak. An introduction to quantum computing for non-physicists. *ACM Computing Surveys*, 32(3):300–335, Sept. 2000.

[24] P. Selinger. Towards a quantum programming language. Unpublished, 2002.

[25] T. Sheard and S. Peyton-Jones. Template meta-programming for Haskell. In *Proc. of the workshop on Haskell*, pages 1–16. ACM, 2002.

[26] P. W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, 1997.

[27] J. Skibiński. Collection of Haskell modules. Available at `http://web.archive.org/web/20010415043244/www.numeric-quest.com/haskell/index.html`, Initialized: 1998-09-18, last modified: 2001-04-02.

[28] J. Skibiński. 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.

[29] A. Steane. Quantum computing. *Reports on Progress in Physics*, 61:117–173, 1998.