Concurrent Quantum Programming in Haskell

Juliana Kaizer Vizzotto and Antônio Carlos da Rocha Costa

***Abstract*—This paper applies established techniques for concurrent programming in Haskell to the case of Concurrent Quantum Programming. The foundation of the approach is an extension to Concurrent Quantum Programming of the technique of "virtual values" proposed by Amr Sabry for quantum programming in Haskell. The basic idea is to encapsulate quantum values within MVars, the monadic variables that support thread synchronization and mutually exclusive accesses to shared references. In this way, quantum processes can be concurring to have access to quantum values and we will be applying the now established quantum programming paradigm of "control is classic, data is quantum" to the concurrent and distributed quantum programming domain: the case in focus is that the control of concurrency is classical control, while shared data between quantum processes are quantum data. We illustrate the use of the proposed approach by programming sample algorithms for quantum teleportation, quantum leader election, and quantum cryptographic key distribution.**

***Index Terms*—Quantum programming, Concurrent Haskell, Simulation of Quantum Algorithms.**

I. Introduction

There is much effort being put into the development of quantum programming languages, while quantum computers strive to open their way to become practical. Several quantum programming languages have been developed (including [17], [14], [16]), and the notion of computational flow (yet classical - first discussed in [10]) was firmly introduced in the description of quantum algorithms. That notion is associated with the idea of an abstract quantum computer operating with qubits, quantum registers, and a small set of suitable operations on those elements. Basically, the operations consist of state preparation, some unitary transformations and measurement.

On the other hand, several authors have noted the connections between quantum programming and functional programming. In [12], Bird and Mu present the applicability of functional languages for writing quantum codes using a monad of probabilistic computations to deal with the (nondeterministic) results of measurements. J. Karczmarczuk [9] takes advantage of the mathematical foundations of functional languages to model quantum mathematical entities (vector spaces, matrix algebra) in Haskell [7]. Also, Amr Sabry [15] develops an elegant approach to quantum programming in the purely-functional language Haskell. The latter is sufficiently powerful for the (inevitably, exponentially slowed down) simulation of quantum processing and the observation of its results. It uses global side-effects to shared references as a mechanism for observing components of entangled data structure such that the result of an observation affects all entangled values. That scenario is established in the context of a sequential programming environment.

In this paper, building on the work of Sabry, we propose an approach to Concurrent Quantum Programming in Concurrent Haskell [8]. Concurrent Haskell is an extension to Haskell that allows us to express explicitly concurrent computations. Basically, we represent a quantum cell as a global reference with a kind of semaphore to control the access to it, and construe a quantum process as a thread. In this way, the quantum processes will be concurring (non-deterministically) to have access to quantum values.

We believe that this simple and conventional approach to concurrent programming allows for a natural expression of some quantum algorithms executing in networks with quantum resources, which need some notion of multi-threaded programming, since they involve multiple (classical, non-quantum) agents and their communication strategies (sometimes via sharing of quantum resources). Concrete examples of such algorithms are quantum teleportation [1], quantum leader election and distributed consensus [4], and quantum cryptography [2], [3].

The paper is organized as follows. Section II presents Sabrys's approach to quantum programming in Haskell. Section III provides an overview of Concurrent Haskell. Our approach to Concurrent Quantum Programming is presented in Section IV. We show in Section V how the approach can be used to implement a quantum leader election. In Section VI, we implement a simplified version of a quantum key distribution algorithm. Finally, in Section VII we present some conclusions and plans for future works.

II. Sabry's Idea of Quantum Programming in Haskell

Amr Sabry presents in [15] an approach to (sequential) quantum programming using the functional language Haskell. He proposes to present quantum computing in a way closer to a programmer's usual vocabulary. In particular, he seeks to stimulate quantum programming with other kinds of quantum data types, besides quantum bits. So, in his approach quantum values are represented as a special data type $QV\ a$, such that all nullary constructors of for the type a are interpreted as unit vectors from a specific base. A specific base a can be obtained by an instantiation of a from the *Basis* class. We show this by defining the qubits in the *Binary* basis through the following declarations:

class $(Eq\ a, Ord\ a) \Rightarrow Basis\ a$ **where** $basis :: [a]$
data $Bin = Zero \mid One$
instance $Basis\ Bin$ **where** $basis = [Zero, One]$

Given unit vectors for type a, values of the type $QV\ a$

Instituto de Infomática/PPGC.
Universidade Federal do Rio Grande do Sul.
Porto Alegre/RS, Brazil. email: jkv@inf.ufrgs.br.
Supported by CNPq.
Escola de Informática
Universidade Católica de Pelotas.
Pelotas/RS, Brazil. email: rocha@atlas.ucpel.tche.br

are finite maps of the library *FiniteMap* [1], which associates each unit vector of a specific basis with a probability amplitude:

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

having the following constructor:

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

and selector:

$pr :: Basis\ a \Rightarrow QV\ a \rightarrow a \rightarrow PA$
$pr\ v\ k = lookupWithDefaultFM\ v\ 0\ k$

Then, some specific values of $QV\ Bin$ can be declared as:

$qZ, qO, qZO :: QV\ Bin$
$qZ = qv\ [(Zero, 1)]$
$qO = qv\ [(One, 1)]$
$qZO = qv\ [(Zero, 1\ /\ sqrt\{2\}),$
$\quad (One, 1\ /\ sqrt\{2\})]$

Moreover, one can construct pairs of values of type $QV\ (a, b)$, which builds on the basis of pairs of quantum values, and allows for the representation of entangled values:

$\mathbf{instance}\ (Basis\ a, Basis\ b) \Rightarrow Basis\ (a, b)\ \mathbf{where}$
$\quad basis = [(a, b) \mid a \leftarrow basis, b \leftarrow basis]$
$qZZ, qOO, qZZOO :: QV\ (Bin, Bin)$
$qZZ = qv\ [((Zero, Zero), 1)]$
$qOO = qv\ [((One, One), 1)]$
$qZZOO = qv\ [((Zero, Zero), 1\ /\ sqrt\ (2)),$
$\quad ((One, One), 1\ /\ sqrt\ (2))]$

The last vector describes an *entangled* quantum state which cannot be separated into the product of independent quantum states. The vector "*qZZOO*" is an EPR-pair, where "EPR" refers to the initials of Einstein, Podolsky, and Rosen who used such a vector in a thought experiment to demonstrate some strange consequences of quantum mechanics [5].

Unitary transformations are implemented as functions on quantum data types. For instance, the *hadamardf* function on quantum values in the binary basis is defined as follows:

$hadamardf :: QV\ Bin \rightarrow QV\ Bin$
$hadamardf\ v =$
$\quad \mathbf{let}\ a = pr\ v\ Zero$
$\quad \mathbf{let}\ b = pr\ v\ One$
$\quad \mathbf{in}\ qv\ [(Zero, (a + b)\ /\ sqrt\ (2)),$
$\quad\quad (One, (a - b)\ /\ sqrt\ (2))]$

The author also presents a matrix alternative for the representation of quantum operations, which specifies how each input amplitude contributes to each output amplitude. Such matrices are also implemented by finite maps, with the constructor below:

$\mathbf{data}\ Qop\ a\ b = Qop\ (FiniteMap\ (a, b)\ PA)$
$qop :: (Basis\ a, Basis\ b) \Rightarrow [((a, b), PA)] \rightarrow Qop\ a\ b$
$qop = Qop.listToFM$

Then, to apply an operation to a quantum value we multiply the matrix and the vector representing the value:

$qApp :: (Basis\ a, Basis\ b) \Rightarrow$
$\quad Qop\ a\ b \rightarrow QV\ a \rightarrow QV\ b$
$qApp\ (Qop\ m)\ v =$
$\quad \mathbf{let}\ pa\ b = sum\ [pr\ m\ (a, b) * pr\ v\ a \mid a \leftarrow basis]$
$\quad \mathbf{in}\ qv\ [(b, pa\ b) \mid b \leftarrow basis]$

For example, the *hadamard* operation can be defined using the following matrix:

$hadamard = qop\ [((Zero, Zero), 1\ /\ sqrt\ (2)),$
$\quad ((Zero, One), 1\ /\ sqrt\ (2)),$
$\quad ((One, Zero), 1\ /\ sqrt\ (2)),$
$\quad ((One, One), -1\ /\ sqrt\ (2))]$

The way to show quantum states to the outside world is to measure them. The outcome of this operation is inherently random and has side effects on the previous (possibly entangled) quantum state. To model such side effects Sabry uses explicit references to shared states. In this way, quantum values can only be accessed via a reference cell and any observation of the value results in the update of the reference cell with the observed value. A quantum reference $QR\ a$, which holds a quantum value $QV\ a$, is defined on top of Haskell's *IORef*. An *IORef* is a mutable variable in the IO monad [6]:

$\mathbf{data}\ QR\ a = QR\ (IORef\ (QV\ a))$
$mkQR :: QV\ a \rightarrow IO\ (QR\ a)$
$mkQR\ v = \mathbf{do}\ r \leftarrow newIORef\ v$
$\quad return\ (QR\ r)$

The IO-action *mkQR* allocates a new quantum reference cell and stores a quantum value in it. Therefore, to observe a quantum value accessible via a reference $QR\ a$, we get the reference's content, observe that value, and update the reference with the result of the observation. This is done by the functions:

$observeR :: QR\ a \rightarrow IO\ a$
$observeR\ (QR\ r) =$
$\quad \mathbf{do}\ v \leftarrow readIORef\ r$
$\quad\quad obs \leftarrow observeV\ v$
$\quad\quad writeIORef\ r\ (qv\ [(obs, 1)])$
$\quad\quad return\ obs$

$observeV :: QV\ a \rightarrow IO\ a$
$observeV\ v =$
$\quad \mathbf{do}\ probs = map\ ((**2) \circ magnitude \circ (pr\ v))\ basis$
$\quad\quad res \leftarrow simulateCollapse\ probs\ basis$
$\quad\quad return\ res$

where *simulateCollapse* is a function that simulates (in an exponentially slowed down way) the reduction of the quantum value due to the observation.

An important feature of quantum programming is that we can operate on parts of a quantum data structure even when that structure is entangled. To allow for such operations on registers of quantum bits, and in general on any other kind of quantum data structure, Sabry proposed the concept of *virtual value*, that is, a part of a data-structure that is virtually separated from the rest of the structure [2].

A virtual value is specified by giving the entire data structure to which it belongs and an *adaptor* that specifies

[1] The library *FiniteMap* is in the Haskell core libraries.

[2] Virtual values seem to generalize the symbolic registers [13] and the use of rotation operations [16] of the QPL and QCL quantum programming languages, respectively.

the mapping from the entire data structure to the part in question, and back:

data *Virt a na u* = *Virt* (*QR u*) (*Adaptator* (*a*, *na*) *u*)
data *Adaptor p ds* =
  Adaptor{ *dec* :: *ds* → *p*, *cmp* :: *p* → *ds* }

In the type (*Virt a na u*), *u* is the type of the entire (possibly entangled) data structure, *a* is the type of the virtual value itself, and *na* is the type of the complementary part of *u* that doesn't belong to *a*. Finally to provide a uniform programming model, it is suggested that all operations in a quantum program be defined in terms of virtual values. There is a way of forming virtual values from references to quantum values:

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

and there is a function *virtFormV* that makes virtual values from other virtual values:

```
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 = λu → let (a, na) = gdec u in
               let (a1, a2) = ldec a in
                   (a1, (a2, na)),
    cmp = λ(a1, (a2, na)) →
               gcmp (lcmp (a1, a2), na) })
```

There is also a way to create virtual values directly from quantum values:

```
virtFromQ = virtFromR ∘ mkQR
```

The input and output of quantum operations should now be virtual values, i.e., an operation with type *Qop a b* should map virtual values of type *Virt a na ua* to virtual values of type *Virt b nb ub*. Thus, the application operator for matrices *app* is defined as:

```
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 m)
  (Virt (QR ra)
    (Adaptor{ dec = deca, cmp = cmpa }))
  (Virt (QR rb)
    (Adaptor{ dec = decb, cmp = cmpb })) =
let m′ = qop [((ua, ub), pr m (a, b)) |
           ua ← basis, ub ← basis,
           let (a, na) = deca ua,
               (b, nb) = decb ub,
           in na ≡ nb]
  in do va ← readIORef ra
        let vb = (qApp m′ va)
        writeIORef rb vb
```

Note that since virtual values live in memory cells, the application operator works quite as an assignment operator: *vb* ← *m* (*va*).

A virtual value can be observed by the function *observeVV* that first uses the adaptor to select the virtual value from the whole data structure, and then uses the function *observeV*, defined above, to observe the value:

```
observeVV :: Virt a na u → IO a
observeVV (Virt (QV r)
(Adaptor{ dec = dec, cmp = cmp })) =
  do let pa a = sqrt (sum [((∗∗2) ∘ magnitude ∘ pr v)
       (cmp (a, na)) | na ← basis])
     let virtV = qv [(a, pa a) | a ← basis]
     obs ← observeV virtV
     let nv = qv [(u, pr v (cmp (obs, na))) |
       u ← basis,
       let (a, na) = dec u,
       a ≡ aobs]
     writeIORef r nv
     return obs
```

III. Concurrent Haskell

Concurrent Haskell [8] is a concurrent extension to the lazy functional language Haskell that introduces two main new ingredients:

- threads, and a mechanism for thread initiation; and
- atomically-mutable state, to support inter-thread communication and cooperation.

Firstly, the language provides a new primitive called *forkIO*, which starts a thread. The type of *forkIO* is:

```
forkIO :: IO a → IO ThreadId
```

It takes an I/O action and arranges to run it concurrently with the "parent" thread.

For communication between different threads, Concurrent Haskell offers a variety of concepts, all based on mutable variables (*MVar*). Mutable variables are embedded in the IO monad [6], which guarantees that threads access *MVar*s only in a mutually exclusive way. This is necessary because of the nondeterminism of the underlying interleaving semantics. Different schedules may lead to different interactions taking place and therefore to different results. In this context, threads can create *MVars*, read values from *MVars* and write values to *MVars*. If a thread tries to read from an empty *MVar* or write to a full *MVar*, then it is suspended until the *MVar* is filled or emptied (respectively) by another thread. Using *MVars*, a type of buffered channels was defined [6]. A channel can be read or written to by multiple threads, it in a safe way.

A. Communication and MVars

The basic set of operations on *MVars* is listed below.

```
data MVar a   -- Abstract
newEmptyMVar :: IO (MVar a)
newMVar :: a → IO (MVar a)
takeMVar :: MVar a → IO a
putMVar  :: MVar a → a → IO ()
readMVar :: MVar a → IO a
```

An *MVar* is (a reference to) a mutable location that either can contain a value of type *a*, or can be *empty*. The operation *newEmptyMVar* creates an empty *MVar*.

The function *putMVar* fills an empty *MVar* with a value, and *takeMVar* takes the contents of an *MVar* out, leaving it empty. If it was empty in the first place, the call to *takeMVar* blocks until another thread fills it by calling *putMVar*. A call to *putMVar* on an *MVar* that is already full blocks the thread until the *MVar* becomes empty. Unlike *takeMVar*, *readMVar* reads the value of an *MVar*, but leaves it full.

B. Channels

A channel with unbounded buffering is defined using the *MVars* [6]. The *Channel* type has the following interface:

```
type Channel a = (MVar (Stream a),     -- read end
                  MVar (Stream a))     -- write end
type Stream a = MVar (Item a)
data Item a = Item a (Stream a)
newChan :: IO (Channel a)
putChan :: Channel a → a → IO ()
getChan :: Channel a → IO a
```

A channel permits multiple processes to write to it (*putChan*), and read from it (*getChan*), safely. Concretely, the channel is represented by a pair of *MVars*, that hold the read end and write end of the buffer. The *MVars* in a *Channel* are required so that channel put and get operations can automatically modify the write and read end of the channels, respectively. The data in the buffer are held in a *Stream*, that is, an *MVar* which is either empty (in which case there is no data in the *Stream*), or holds an *Item*. An *Item* is just a pair of the first element of the *Stream* together with a *Stream* holding the rest of the data.

IV. Concurrent Quantum Programming with Concurrent Haskell

The central idea of our proposal is to encapsulate quantum values within concurrent Haskell's *MVar*, that is, to extend Sabry's quantum registers with semaphores to control concurrent access. In this way, a scenario for multithreaded quantum programming arises where threads are guaranteed to have mutually exclusive accesses to quantum values.

A. Defining Quantum Semaphores and Related Structures

A quantum semaphore *QMVar a*, that holds a quantum value *QV a*, is defined as:

```
data QMVar a = QMVar (MVar (QV a))
```

Operations to allocate a new *QMVar*, and to read and write its quantum value can be given as:

```
mkQMVar :: QV a → IO (QMVar a)
mkQMVar v = do p ← newMVar v
               return (QMVar p)
putQMVar :: QMVar a → QV a → IO ()
putQMVar (QMVar p) v = putMVar p v
takeQMVar :: QMVar a → IO (QV a)
takeQMVar (QMVar p) = do v ← takeMVar p
                         return v
```

Because of the mechanism of *MVArs*, the operation *PutQMVar* on a full *QMVAr* blocks until other thread fills that *QMVAr* with a quantum value. In the same way *takeQMVar* blocks if the *QMVAr* is empty.

Note that *QMVars* provide the necessary mechanism for mutual exclusion during the observation of quantum values, for when a value inside an *QMVar* is being observed by a thread, all other threads should be blocked until the former updates the value with the observed value:

```
observeQMVar :: Basis a ⇒ QMVar a → IO a
observeQMVar (QMVar r) =
  do v ← takeMVar r
     res ← observeV v
     putMVar r (qv [(res, 1)])
     return res
```

We saw in the section II that Sabry's proposal is that all computation with quantum values be performed with virtual values built upon reference cells. Therefore, we upgrade the reference cell with *MVars* to allow mutual exclusion.

```
data Virt a na u =
        Virt (QMVar u) (Adaptor (a, na) u)
virtFromQMVar :: QMVar a → Virt a () a
virtFromQMVar r =
  Virt r (Adaptor{ dec = λa → (a, ()),
                   cmp = λ(a, ()) → a})
```

Analogously, we redefine *observeVV* to work with quantum semaphores.

V. Programming Quantum Leader Election

In the course of a distributed computation, it is often useful to be able to designate one and only one process as the coordinator of some activity. This selection of a coordinator is known as the "leader election problem". In anonymous networks, where there is no unique naming scheme for processes, purely deterministic *classical* leader election is impossible. If each process has a coin then they can elect a leader by tossing the coin. If they get a head they are the leader. This is not guaranteed to work: there may be more than one leader or no leaders. In this section we implement the leader election (fair and terminating) quantum algorithm for anonymous network proposed in [4]. The protocol is very simple. Essentially, in such an algorithm the processors share a special quantum entangled state called W-state [3]:

$$W_n = \sum_{j=1}^{n} |2^j\rangle.$$

For instance

```
w4 =
  normalize (qv [((Zero, Zero, Zero, One), 1),
                 ((Zero, Zero, One, Zero), 1),
                 ((Zero, One, Zero, Zero), 1),
                 ((One, Zero, Zero, Zero), 1)])
```

[3] Here using the "bracket" Dirac notation.

The idea is that each process p_i initially owns the i qubit from W. Then each process carries out the following protocol:

```
p_i qmv = do putStrLn ("Pi")
             result ← newEmptyMVar
             let qi = virtFromQMVar qmv
             let vqi = virtFromV qi ad_quadi
             meas ← observeVV vqi
             if meas ≡ One
               then do putMVar result "leader"
               else do putMVar result "follower"
             res ← takeMVar result
             print (res)
```

if it observers *One* then it is the leader otherwise is in the follower state.

We simulate a leader election in a network with four process using a parent thread which sparks the four process defined as above.

```
leader_election =
  do qmv ← mkQMVar w_4
     o1 ← myForkIO (p_1 qmv)
     o2 ← myForkIO (p_2 qmv)
     o_3 ← myForkIO (p_3 qmv)
     o_4 ← myForkIO (p_4 qmv)
     mapM_ (λmvar → readMVar mvar) [o1, o2, o_3, o_4]
     print "The end!"
```

An example of output would be:

```
* ConcQComp > leader_election
P1
"follower"
P2
"follower"
P3
"leader"
P4
"follower"
"The end!"
* ConcQComp >
```

VI. Programming a Quantum Key Distribution Algorithm

In 1984 Bennet and Brassard described the first quantum key distribution algorithm [2], [3]. Quantum key distribution (QKD) is a protocol by which private key bits can be created between two parties over a *public* channel. The basic idea behind QKD is the following fundamental observation [11]: an eavesdropper cannot gain any information from observing a quantum channel, where quantum values are transmitted from the sender to the receiver, without disturbing the states of such values because of the effects that observations have on quantum states.

A. Defining Quantum Channels

A quantum channel is a Haskell channel that holds quantum values, together with operations to write to it, and read from it.

```
data QChan a = QChan (Chan (QV a))
mkQChan :: IO (QChan a)
mkQChan = do r ← newChan
             return (QChan r)
writeQChan :: QChan a → QV a → IO ()
writeQChan (QChan chan) qv = writeChan chan qv
readQChan :: QChan a → IO (QV a)
readQChan (QChan chan) = do v ← readChan chan
                            return v
```

B. Implementing the BB84 QKD Protocol

The algorithm we implement in this section is the BB84 protocol. The protocol is as follows: Alice begins with a (the key) and b (codifying basis), two strings each of $4\,n$ random classical bits. She encodes each data bit of a as $\{|\,0\,\rangle, |\,1\,\rangle\}$ (called X base) if the corresponding bit of b is 0 or $\{|+\rangle, |-\rangle\}$ (called Z base) if b is 1. Alice sends the resulting quantum states to Bob and tells when she finishes. Bob receives the $4n$ quantum values, announces this fact, and measures each of them in the X or Z bases at random. Alice announces b. Alice and Bob discard any bits where Bob measured a different basis than Alice prepared. With high probability, there are at least $2n$ bits left (if not, abort the protocol). Alice selects a subset of n bits that will serve as check bits on Eve's interference, and tells Bob which bits she selected. Alice and Bob announce and compare the values of the n check bits. If some bit disagree they abort the protocol [4].

In this context, there is a classical channel *chan* which is used for classical communication between Alice and Bob. This channel may hold single strings for the acknowledgments, and lists of classical bits for the announcement of the basis:

```
data Protocol = Single String | Multiple [Bit]
```

The parent thread creates a quantum channel *QChan*, and a classical channel *Chan*, and forks the two child threads *alice* and *bob*. We also use here the function *outForkIO* that generates an output, allowing the parent thread to force the program to wait for child threads to finish:

```
qkeyd = do putStrLn ("Beginning BB84")
           qchan ← mkQChan
           chan ← newChan
           o1 ← outForkIO (alice qchan chan)
           o2 ← outForkIO (bob qchan chan)
           map (λmvar → readMVar mvar) [o1, o2]
                                 -- wait for the children
           putStrLN ("The End")
```

Alice generates the two random bit lists (*bits* and *bases* - a and b above, respectively) using the function *randomBitList*[5]. The argument of this function is the number of key bits. Then, the function *qvList* builds the list of (key) quantum values according to the basis list (*basis*). Next, Alice puts the list of quantum values in the *QChan* and informs this fact to Bob with an "ASend_Ok".

[4] The phases of information reconciliation and privacy amplification on the remaining bits are left away from this paper.

[5] Because of lack of space we don't show here the coding of some auxiliary functions.

The function *wishGet* has a channel and a value as arguments. It observes the channel until getting the desired value. After observing "Ack_Bob" in the classical channel, Alice writes her basis in this channel. Finally, Alice receives Bob's basis and checks with her basis to confirm the generation of the secret key [6].

```
alice qchan chan =
  do putStrLn ("Alice started")
     basis ← randomBitList 36
     bits ← randomBitList 36
     x ← qvList bits basis
     putQVChan qchan x
     writeChan chan (Single "ASend_Ok")
     wishGet chan (Single "Ack_Bob")
     writeChan chan (Single "ASend_Ok")
     writeChan chan (Multiple basis)
     wishGet chan (Single "Ack_Bob")
     Multiple bbasis ← readChan chan
     result ← compBasis basis bbasis bits
     putStrLn ("Alice's key:")
     print (result)
     putStrLn ("Alice Finished")
bob qchan chan =
  do putStrLn ("Bob started")
     wishGet chan (Single "ASend_Ok")
     qvl ← getQVChan qchan
     writeChan chan (Single "Ack_Bob")
     basis ← randomBitList 36
     obs ← bitList qvl basis
     wishGet chan (Single "ASend_Ok")
     Multiple abasis ← readChan chan
     writeChan chan (Single "Ack_Bob")
     writeChan chan (Multiple basis)
     result ← compBasis abasis basis obs
     putStrLn ("Bob's Key:")
     print (result)
     putStrLn ("Bob finished")
```

After reading an "ASend_Ok" from the classical channel, Bob gets all quantum values from the quantum channel by the operation *getQVChan*, that gets quantum values from the channel until it is empty. Then he announces this fact to Alice by putting the message "Ack_Bob" in the *Chan*. Next, Bob creates his random base list and observes the quantum values according to the basis. Then, after reading the message "ASend_Ok" and Alice's basis, he writes his basis in the *Chan*. Finally, Bob also defines the secret key comparing his basis with Alice's basis.

A running without an eavesdropper would always give Bob and Alice finishing with the same key.

VII. Conclusions and Future Work

We presented an approach to Concurrent Quantum Programming in Concurrent Haskell building on Amr Sabry's proposal of storing quantum values as global references for modelling side effects of measurements, and casting quantum data structures as virtual values for supporting the separate handling of their parts. The basic idea is to embed quantum values in MVars, to guarantee mutually exclusive accesses to them by concurrently running quantum threads. The approach was demonstrated by the implementation of three sample quantum algorithms, namely, quantum teleportation, quantum leader election and quantum key distribution. Basing the work on the slogan "control is classic, data is quantum" we were able to use simple and conventional concurrent programming constructs to support Concurrent Quantum Programming. The full range of applicability of the approach still remains to be determined. In particular, the problem of distribution and parallelization of conventionally sequential quantum algorithms, and the determination of the advantages of doing that, seems to be interesting motivation for further work.

[6] Actually, at this point, this is not the real secret key because Alice and Bob should also perform some tests to determine how much noise or eavesdropping happened during their communication.

References

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

[2] C.H. Bennett and G. Brassard. Quantum cryptography: Public key distribution and coin tossing. In *Proceedings of IEEE International Conference on Computers Systems and Signal Processing*, pages 175–179, December 1984.

[3] Charles H. Bennett and Gilles Brassard. Quantum public key distribution reinvented. *SIGACT News*, 18(4):51–53, 1987.

[4] Ellie D'Hondt and Prakash Panangaden. Leader election and distributed consensus with quantum resources. 2005.

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

[6] Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell.

[7] Simon Peyton Jones. *Haskell 98 Language and Libraries.* Cambridge University Press, 2003.

[8] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Conference Record of POPL '96: The* 23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308, St. Petersburg Beach, Florida, 21–24 1996.

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

[10] E. Knill. Conventions for quantum pseudocode, 1996.

[11] Isaac L. Chuang Michael A. Nielsen. *Quantum Computation and Quantum Information.* Cambridge University Press, 2000.

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

[13] B. Omer. *A procedural formalism for quantum computing.* PhD thesis, Dept. Theor. Physics, Technical University of Vienna, 1998.

[14] B. Omer. Procedural quantum programming. In *Computing Anticipatory Systems: CASYS 2001 - 5th International Conference, AIP Conference Proceedings 627*, pages 276–285, 2001.

[15] Amr Sabry. Modeling quantum computing in haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 39–49. ACM Press, 2003.

[16] Peter Selinger. Towards a quantum programming language. *Mathematical Structures in Computer Science*, 14(4):527–586, 2004.

[17] Paolo Zuliani. *Quantum Programming.* PhD thesis, University of Oxford, 2001.