Towards Quantum Haskell via Quantum Arrows

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

[1]ESIN/PPGINF – Universidade Católica de Pelotas (UCPel)
Felix da Cunha, 412 – 96010-000 – Pelotas – RS – Brazil

{jkv,rocha}@ucpel.tche.br

Abstract. *This work proposes a set of high level primitives to be used for quantum programming. We call Quantum Haskell (QHaskell) the language generated by the primitives as it is induced by a model for quantum programming structured as arrows and implemented in Haskell. The main aspects of the language are its special syntax for manipulation of quantum data inspired on the* **do** *notation for arrows in Haskell, and a quantum control primitive inspired by the language QML, following the paradigm "quantum data and control". We show how the constructions of the language fit as high level primitives to quantum programming through some examples. Finally, we briefly sketch a linear type system for the proposed language.*

1. Introduction

In this paper we propose a mixed programming language with classical and quantum data, and classical and quantum control. The syntax of the language is inspired by the model for mixed computations structured as indexed arrows which we presented in [Vizzotto et al. 2006b]. The paper is structured as follows. In next section we discuss indexed monads and indexed arrows. Section 3 reviews our previous work [Vizzotto et al. 2006b]. Section 4 presents the syntax, examples, and briefly discuss the type system of the proposed language. Section 5 concludes.

2. Indexed Monads and Indexed Arrows

The mathematical concept of monads was introduced to computer science by Moggi [Moggi 1989] in the late 1980's as a way of structuring denotational semantics of programming languages. Several different language features, including nontermination, state, exceptions, continuations, and interaction can be viewed as monads. More recently, this construction has been internalised in the programming language Haskell as a tool to elegantly express computational effects within the context of a pure functional language.

Since the work of Moggi, several natural notions of computational effects were discovered which could only be expressed as generalisations of monads. Of particular importance to us is the generalisation of monads known as arrows [Hughes 2000] which is also internalised in the programming language Haskell. In this section, we briefly discuss a small variation of these notions in the context of the programming language Haskell, which we call *indexed* monads and *indexed* arrows. Those are the right notions needed to structure quantum computations.

2.1. Indexed Monads

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

More precisely, to understand monadic computations one can consider a value category $\mathcal{C}$, as a model for functions, and build on top of that, notions of computation via an operator (*endofunctor*) T acting on objects of $\mathcal{C}$ - i.e., T maps an object B from $\mathcal{C}$, viewed as the *set of values of type* τ, to an object TB corresponding to *computations of type* τ. Then a program which takes an input of type A, and after performing a certain computation returns a value of type B, can be identified with a morphism from A to TB in $\mathcal{C}$ [Moggi 1991].

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

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

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

However, sometimes we want to *select* some objects (sets) from $\mathcal{C}$ to apply the constructor T. This notion is slightly more general than monads, and it is captured by the definition of *Kleisli structure* [Altenkirch and Reus 1999]. Basically, for *indexed monads* (as we prefer to call Kleisli structures), the function T does not need be an endofunctor on $\mathcal{C}$. We can select some objects from $\mathcal{C}$ to apply the constructor. This is exactly the notion we need to model quantum state vectors [1] as monads. The constructor for a quantum vector can only act over the types which constitute a basis.

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

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

That is, for all a for which $F\ a$ holds we can apply the constructor m, and for all a and b for which $F\ a$ and $F\ b$ hold we can apply $\gg\!=$. Moreover, to construe a proper monad or *indexed* monad, the $return$ and $\gg\!=$ functions must work together according to the three monad laws [Moggi 1989].

2.2. Indexed Arrows

To handle situations where monads are inapplicable, Hughes [Hughes 2000] introduced a new abstraction generalising monads, called *arrows*. Indeed, in addition to defining a notion of procedure which may perform computational effects, arrows may have a static component, or may accept more than one input.

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

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

$$\begin{aligned}
&\mathit{arr} \ \ \in \mathit{forall}\ b\ c.(b \to c) \to a\ b\ c\\
&(\ggg) \in \mathit{forall}\ b\ c\ d.a\ b\ c \to a\ c\ d \to a\ b\ d\\
&\mathit{first}\ \in \mathit{forall}\ b\ c\ d.a\ b\ c \to a\ (b,d)\ (c,d)
\end{aligned}$$

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

Observe the requirements of forall in the definitions. They mean that we can build computations on top of ***all*** value functions. However, as with monads, we want to ***select*** some specific value functions. This is the case for quantum functions: we want to lift simple functions acting on the ***basis*** elements to functions acting on vectors over those basis. Hence we define ***indexed arrows*** [2]:

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

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

3. Review: Quantum Arrows

3.1. Vectors as Indexed Monads

In this section we quickly review the work presented in [Vizzotto et al. 2006a]. Given a set a representing a ***basis*** set, a pure quantum state is a vector $a \to \mathbb{C}$ which associates each basis element with a complex probability amplitude. In Haskell, a finite set a can be represented as an instance of the class $Basis$, shown below, in which the constructor $basis \in [a]$ explicitly lists the basis elements. The basis elements must be distinguishable from each other, which explains the constraint $Eq\ a$ on the type of elements:

$$\begin{aligned}
&\mathbf{class}\ \mathit{Eq}\ a \Rightarrow \mathit{Basis}\ a\ \mathbf{where}\ \mathit{basis} \in [a]\\
&\mathbf{type}\ K = \mathbb{C}\ \mathit{Double}\\
&\mathbf{type}\ \mathit{Vec}\ a = a \to K
\end{aligned}$$

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

The monadic functions for vectors are defined as:

$$\begin{aligned}
&\mathit{return} \in \mathit{Basis}\ a \Rightarrow a \to \mathit{Vec}\ a\\
&\mathit{return}\ a\ b = \mathbf{if}\ a \equiv b\ \mathbf{then}\ 1.0\ \mathbf{else}\ 0.0\\
&(\gg\!=) \in (\mathit{Basis}\ a, \mathit{Basis}\ b) \Rightarrow \mathit{Vec}\ a \to (a \to \mathit{Vec}\ b) \to \mathit{Vec}\ b
\end{aligned}$$

[2] Categorically, the definition of arrows is captured by Freyd-categories [Paterson 2001]. ***Indexed arrows*** are ***indexed*** Freyd-categories.

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

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

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

Proof.

- First monad law: $(return\ x)\ >>=\ f\ =\ f\ x$

$$\begin{aligned}(return\ x)\ \gg\!=\ f &= \lambda b \rightarrow sum\ [(return\ x\ a) * (f\ a\ b) \mid a \leftarrow basis] \\ &= \lambda b \rightarrow sum\ [(\mathbf{if}\ x \equiv a\ \mathbf{then}\ 1.0\ \mathbf{else}\ 0.0) * (f\ a\ b) \mid \\ &\qquad a \leftarrow basis] \\ &= \lambda b \rightarrow f\ x\ b \\ &= f\ x\end{aligned}$$

- Second monad law: $m\ >>=\ return\ =\ m$

$$\begin{aligned}m\ \gg\!=\ return &= \lambda b \rightarrow s\ um\ [(m\ a) * (return\ a\ b) \mid a \leftarrow basis] \\ &= \lambda b \rightarrow sum\ [(m\ a) * (\mathbf{if}\ a \equiv b\ \mathbf{then}\ 1.0\ \mathbf{else}\ 0.0) \mid \\ &\qquad a \leftarrow basis] \\ &= \lambda b \rightarrow m\ b \\ &= m\end{aligned}$$

- Third monad law: $(m\ >>=\ f)\ >>=\ g\ =\ m\ >>=\ (\lambda\, x\, .\, f\ x\ >>=\ g)$

$$\begin{aligned}(m\ \gg\!=\ f)\ \gg\!=\ g &= (\lambda b \rightarrow sum\ [(m\ a) * (f\ a\ b) \mid a \leftarrow basis])\ \gg\!=\ g \\ &= \lambda c \rightarrow sum\ [(sum\ [(m\ a) * (f\ a\ b) \mid a \leftarrow basis]) \\ &\qquad * (g\ b\ c) \mid b \leftarrow basis] \\ &= \lambda c \rightarrow sum\ [(m\ a) * (f\ a\ b) * (g\ b\ c) \\ &\qquad a \leftarrow basis, b \leftarrow basis]\end{aligned}$$

$$\begin{aligned}m\ \gg\!=\ (\lambda x \rightarrow f\ x\ \gg\!=\ g) &= \lambda c \rightarrow sum\ [(m\ a) * ((f\ a\ \gg\!=\ g)\ c) \mid \\ &\qquad a \leftarrow basis] \\ &= \lambda c \rightarrow sum\ [(m\ a) * (sum\ [(f\ a\ b) * (g\ b\ c) \mid \\ &\qquad b \leftarrow basis]) \mid a \leftarrow basis] \\ &= \lambda c \rightarrow sum\ [(m\ a) * (f\ a\ b) * (g\ b\ c) \mid \\ &\qquad a \leftarrow basis, b \leftarrow basis]\end{aligned}$$

□

3.2. Superoperators as Indexed Arrows

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

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

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

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

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

Just as the probability effect associated with vectors is modelled by an *indexed monad* because of the $Basis$ constraint, the type $Super$ is modelled by an *indexed arrow* because the types include the additional constraint requiring the elements to form a set of basis values (the definition for arr, $\ggg$, and $first$ for $Super$ are in [Vizzotto et al. 2006a]).

Using this *general* model of quantum computations structured as arrows we can elegantly express quantum computations involving measurements. However, since that work is strictly based on the paradigm of "quantum data and quantum control", we can not express algorithms with combined interactions of quantum and classical operations directly. Yet as noted in [Gay and Nagarajan 2005, Unruh 2005] a *complete* model for expressing quantum algorithms should accommodate both measurements and combined interactions of quantum and classical data.

3.3. Mixed Programs as Indexed Arrows

The model presented in the previous section is purely quantum. However, various quantum algorithms are explained in terms of the *interchanging* of quantum and classical information. For instance, quantum teleportation is a traditional example of an algorithm which is based on two quantum process communicating via *classical data*. There is interest to consider a *mixed* model for quantum computations involving *measurements* and the *information flow* between quantum and classical processes (for instance, see [Raussendorf et al. 2003, Kashefi et al. 2004, Gay and Nagarajan 2005, Unruh 2005]).

Hence we have introduced in [Vizzotto et al. 2006b] a model for *mixed computations*, quantum and classical, acting on a *combined state*, with a classical and a quantum part, based on a measurement approach. The idea is to have a density operator representing the quantum part of the state, and a probability distribution of classical values representing the classical part of the state. A quantum program acting on this combined state is interpreted by a special *tracing superoperator*, which in the general case traces out part of the state, returning a classical output, and leaving the system in a new state (possibly leaving the quantum part of the state in a space with reduced dimension).

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

$$\begin{array}{l}\textbf{class}\ (Basis\ a, Basis\ b, Basis\ o) \Rightarrow Dec\ a\ b\ o\ \textbf{where}\\ \quad dec \in [a] \rightarrow [(b, o)]\end{array}$$

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

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

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

Any unitary operator can be lifted to a quantum program which traces out $()$ and does nothing to the classical part of the state. The idea is to apply the default construction to build a superoperator from a unitary transformation. The classical input is ignored and the classical output is empty: there is no interaction with the classical world when considering unitary transformations.

More interestingly we can define *measurements* and *projections*. Given, a quantum state over a basis set (a, b), a quantum program can forget the *right* component, returning a new state over b. The subspace is measured before being discharged outputting a classical probability distribution over the basis which forms that subspace. In this case, the input classical data is just ignored.

Finally, we have defined the function $arr, \ggg$ and $first$ for $QProgram\ \ i\ \ o$ and shown that this indexed arrow satisfies the required equations for arrows.

Using such programas structured as arrows we can use Paterson (2001)'s *arrow notation* for writing our combined programs. Here is a simple example to illustrate the notation:

$$
\begin{array}{l}
e_1 \in QProgram\ \ i\ \ o\ \ (a_1, a_2)\ \ (b_1, b_2) \\
e_1 = proc\ \ (a, b) \rightarrow \mathbf{do} \\
\qquad\qquad r \leftarrow f a_1 \prec a \\
\qquad\qquad returnA \prec (r, b)
\end{array}
$$

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* $\prec$ which feeds the value of an expression into an arrow.

4. QHaskell Syntax and Type System

4.1. Syntax and Examples

The language QHaskell consists of a simple *mixed* functional language which features:

- classical and quantum values;
- a classical control operation;
- a quantum control operation in the same sense as presented in [Altenkirch and Grattage 2005];
- a measurement operation, which allows the reading of classical information from quantum values;
- a special kind of procedure, $proc$, to *sequentialise* operations. From the model in section above, we noted that we could add the syntax sugar for arrows computations to our language. The sequence of actions on quantum data must be specified into a $proc$. The idea of $proc$ is to explicitly allow the programmer, in a high level way, to manipulate quantum parts of the (possible entangled) global quantum part of the state; and

- *commands* which can be executed following the sequence imposed by $proc$.

$$
\begin{array}{lll}
(\mathit{Variables}) & x, y, ... \in & \mathit{Vars} \\
(\mathit{Prob.ampl.})\ \kappa, \iota & \in & \mathbb{C} \\
(\textit{data}) \qquad d & ::= & \mathit{false} \mid \mathit{true} \mid \kappa * d \mid q + r \\
(\mathit{programs})\ p & ::= & e \mid x \mid \mathbf{meas}\ x \\
 & \mid & \mathbf{if}\ x\ \mathbf{then}\ p_1\ \mathbf{else}\ p_2 \\
 & \mid & \mathit{proc}\ xl \rightarrow \mathbf{do}\ cl \\
 & \mid & pr \prec xl \\
 & \mid & \mathbf{let}\ x = p_1\ \mathbf{in}\ p_2 \\
 & \mid & x \leftarrow e \\
 & \mid & p_1; p_2 \\
 & \mid & \mathbf{qif}\ x\ \mathbf{then}\ p_1\ \mathbf{else}\ p_2 \\
 & \mid & \mathit{returnA} \prec xl
\end{array}
$$

The classical part of the language consists of variables, **let**-expressions, booleans, and conditionals. Quantum values can be build by the constructors $*$ and $+$. The term $\kappa * t$v associatew a complex number κ, a *probability amplitude*, with the term t. Moreover, $t_1 + t_2$ constructs a *superposition* of t_1 and t_2. The commands $x \leftarrow e$ and $pr \prec xl$, assigns the value of e to x and feeds the value of an expression into a program, respectively.

To give intuition about the language we show some examples. To simplify the presentation of the examples we use "global" procedure definitions.

We can define simple rotations on qubits as follows:

$$
\begin{array}{l}
\mathit{qnot} = \mathit{proc}\ x \rightarrow \mathbf{do} \\
\qquad\qquad \mathbf{qif}\ x\ \mathbf{then}\ x' \leftarrow \mathit{false} \\
\qquad\qquad\qquad\ \mathbf{else}\ \ x' \leftarrow \mathit{qtrue} \\
\qquad\qquad \mathit{returnA} \prec x' \\
\mathit{hadamard} = \mathit{proc}\ x \rightarrow \mathbf{do} \\
\qquad\qquad\qquad \mathbf{qif}\ x\ \mathbf{then}\ x' \leftarrow (\mathit{false} + (-1) * \mathit{true}) \\
\qquad\qquad\qquad\qquad \mathbf{else}\ x' \leftarrow \mathit{false} + \mathit{true} \\
\qquad\qquad\qquad \mathit{returnA} \prec x'
\end{array}
$$

The idea of the quatum control **qif** is the same as originally proposed for the language QML [Altenkirch and Grattage 2005], that is, it analyses the input qubit x without measuring it. Intuitively, it is a constructor for unitary operations. Hence, the evaluation of $\mathit{qnot}\ (\kappa * \mathit{qfalse} + \iota * \mathit{qtrue})$ swaps the probability amplitudes associated with qfalse and qtrue. The same applies to $\mathit{hadamard}$.

More interestingly, quantum controlled operations can be easily implemented using the $proc$ abstraction:

$$
\begin{array}{l}
\mathit{cqnot} = \mathbf{let}\ \mathit{qnot} = \mathit{proc}\ x \rightarrow \mathbf{do} \\
\qquad\qquad\qquad\qquad\qquad \mathbf{qif}\ x\ \mathbf{then}\ x' \leftarrow \mathit{false} \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \mathbf{else}\ \ x' \leftarrow \mathit{true} \\
\qquad\qquad\qquad\qquad\qquad \mathit{returnA} \prec x' \\
\qquad\qquad \mathbf{in}\ \mathit{proc}\ (c, y) \rightarrow \mathbf{do} \\
\qquad\qquad\qquad\qquad \mathbf{qif}\ c\ \mathbf{then}\ y' \leftarrow \mathit{qnot} \prec x \\
\qquad\qquad\qquad\qquad\qquad\ \mathbf{else}\ y' \leftarrow y
\end{array}
$$

$$returnA \prec (c, y')$$

Other interesting example, would be teleportation as it involves a measurement in the middle of the computation and classical controlled operations. The main procedure receives no classical data and three entangled qubits; then passes a qubit of the epr pair and the qubit to be teleported to Alice, which realizes some quantum operations and measures its two qubits, returning only classical values to the main procedure, which will be communicated to Bob.

$$\begin{aligned} teleportation = proc\ (eprL, eprR, q) \rightarrow \mathbf{do} \\ (m_1, m_2) \leftarrow alice \prec (eprL, q) \\ q' \leftarrow bob \prec (eprR, (m_1, m_2)) \\ returnA \prec q' \end{aligned}$$

$$\begin{aligned} alice = \mathbf{let}\ qnot = ... \\ hadamard = ... \\ \mathbf{in}\ proc\ (eprL, q) \rightarrow \mathbf{do} \\ (q_1, e_1) \leftarrow qcnot \prec (q, eprL) \\ q_2 \leftarrow hadamard \prec q_1 \\ (m_1, m_2) \leftarrow \mathbf{meas} \prec (e_1, q_2) \\ returnA \prec (m_1, m_2) \end{aligned}$$

Bob is a procedure which receives two classical values and a qubit. The procedure analyses the classical data and depending on its value applies or not a certain quantum operation to the input qubit.

$$\begin{aligned} bob = proc\ (eprR, (m_1, m_2)) \rightarrow \mathbf{do} \\ \mathbf{if}\ m_1\ \mathbf{then}\ r' \leftarrow qnot \prec eprR \\ \mathbf{else}\ r' \leftarrow eprR \\ \mathbf{if}\ m_2\ \mathbf{then}\ r'' \leftarrow z \prec r' \\ \mathbf{else}\ r'' \leftarrow r' \\ returnA\ r'' \end{aligned}$$

4.2. Type System

As usual for quantum programming languages, the main rôle of the type system is to control the use of variables. The typing rules of QHaskell, as for QML [Altenkirch and Grattage 2005], are based on strict linear logic, where contractions are implicit and weakenings are not allowed when they correspond to information loss.

We use σ, τ, ρ to vary over classical types which are given by the following grammar:

$$\sigma = 1 \mid Bool \mid \sigma \otimes \tau$$

We use θ, υ to vary over quantum types which are given by the following grammar:

$$\theta = \mathcal{Q}_1 \mid \mathcal{Q}_{Bool} \mid \theta \otimes \upsilon$$

Types are first-order and finite in the current version of the language: there are no higher-order types and no recursive types. The only types we can represent are the types

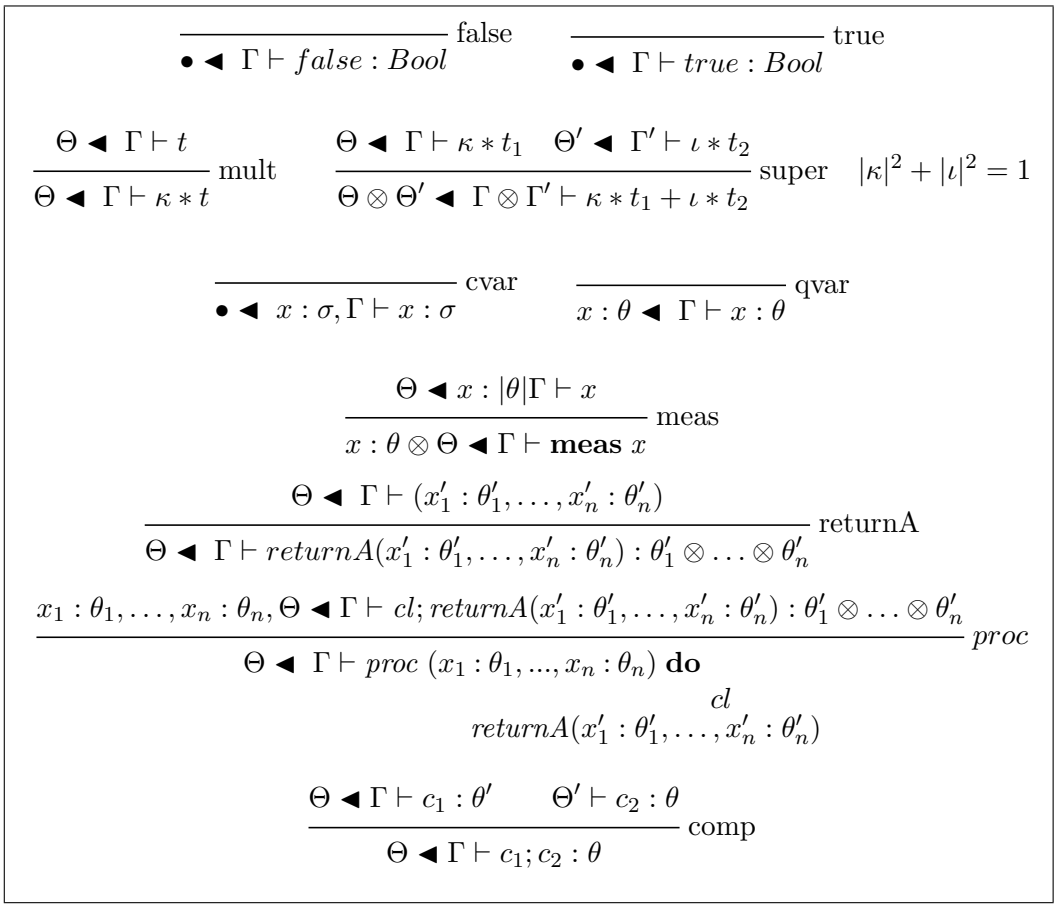

Figure 1. Typing terms

of collections of qubits.

Classical typing contexts (Γ, Δ) are given by:

$$\Gamma = \bullet \mid \Gamma, x : \sigma$$

where $\bullet$ stands for the empty context, but is omitted if the context is non-empty.

Quantum typing contexts (Θ, Υ) are given by:

$$\Theta = \bullet \mid \Theta, x : \theta$$

where $\bullet$ stands for the empty context, but is omitted if the context is non-empty.

The main improvement of our type system is to always maintain two environments: a classical and a quantum one. In this way we can control which operations are allowed to apply to quantum and classical values. Interestingly one can note that variables can pass from the classical environment to the quantum one and vice-versa.

In Figure 1 we sketch the type system for QHaskell. Judgements have two environments, that one which precedes $\blacktriangleleft$ is the quantum one and the following is the classical one.

5. Conclusion

In this paper we have presented a simple *mixed* programming language with quantum control. Mainly our language proposes a kind of sequential procedure $proc$ for quantum programming. The sequence of actions on quantum data must be specified into a $proc$. The idea of $proc$ is to explicitly allow the programmer, in a high level way, to manipulate quantum parts of the (possible entangled) global quantum part of the state. Besides of a measurement operation, which allows the reading of classical information from quantum values, we have also, as motivated by QML [Altenkirch and Grattage 2005], added a primitive for quantum control to the language. That allows the definitions of unitary operations by the programmer. No initial set of universal quantum operations must be assumed.

We are currently developing the language QHaskell. After implementing the language in Haskell itself, an obvious next step would be starting the extraction of a *rewriting semantics* for a mixed programming language.

References

Altenkirch, T. and Grattage, J. (2005). A functional quantum programming language. In *20th Annual IEEE Symposium on Logic in Computer Science*. to appear.

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

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

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

Kashefi, E., Panangaden, P., and Danos, V. (2004). The measurement calculus.

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

Moggi, E. (1991). Notions of computation and monads. *Information and Computation*, 93(1):55–92.

Paterson, R. (2001). A new notation for arrows. In *International Conference on Functional Programming*, pages 229–240. ACM Press.

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

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

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

Vizzotto, J. K., da Rocha Costa, A. C., and Sabry, A. (2006b). Quantum arrows in haskell. In *4rd International Workshop on Quantum Programming Languages*. to appear in ENTCS.