

The Arrow Calculus as a Quantum Programming Language

Juliana Kaizer Vizzotto¹, André Rauber Du Bois² and Amr Sabry³

¹ Mestrado em Nanociências, Centro Universitário Franciscano
Santa Maria, RS/ Brazil

² PPGI, Universidade Católica de Pelotas
Pelotas, RS/Brazil

³ Department of Computer Science, Indiana University
Bloomington, USA

Abstract. We express quantum computations (with measurements) using the arrow calculus extended with monadic constructions. This framework expresses quantum programming using well-understood and familiar classical patterns for programming in the presence of computational effects. In addition, the five laws of the arrow calculus provide a convenient framework for *equational reasoning* about quantum computations that include measurements.

1 Introduction

Quantum computation [1] can be understood as a *transformation* of information encoded in the state of a *quantum* physical system. Its basic idea is to encode data using quantum bits (qubits). Differently from the classical bit, a qubit can be in a *superposition* of basic states leading to “quantum parallelism.” This form of parallelism is due to the non-local wave character of quantum information and is qualitatively different from the classical notion of parallelism. This characteristic of quantum computation can greatly increase the processing speed of algorithms. However, quantum data types are computationally very powerful not only due to superposition. There are other odd properties like *measurement*, in which the observed part of the quantum state and every other part that is *entangled* with it immediately lose their wave character.

These interesting properties have led to the development of very efficient quantum algorithms, like Shor’s quantum algorithm for factorizing integers [2], and Grover’s quantum search on databases [3]. Another important theme is the development of quantum cryptographic techniques [4].

Since these discoveries, much research has been done on quantum computation. Summarizing the field of research we can classify it according three main areas: i) physical implementations of quantum computers, ii) development of new quantum algorithms; and iii) design of quantum programming languages.

This work is about the design of a quantum programming language, and consequently about a high-level, structured and well-defined way to develop new quantum algorithms and to *reason* about them.

We have been working on semantic models for quantum programming. In previous work [5] we established that general quantum computations (including measurements) are an instance of the category-theoretic concept of arrows [6], a generalization of *monads* [7] and *idioms* [8]. Translating this insight to a practical programming paradigm has been difficult however. On one hand, directly using arrows is highly non-intuitive, requiring programming in the so-called “point-free” style where intermediate computations are manipulated without giving them names. Furthermore reasoning about arrow programs uses nine, somewhat idiosyncratic laws.

In recent work, Lindley *et. al.* [9] present the *arrow calculus*, which is a more friendly version of the original presentation of arrows. The arrow calculus augments the simply typed lambda calculus with four constructs satisfying five laws. Two of these constructs resemble function abstraction and application, and satisfy familiar beta and eta laws. The remaining two constructs resemble the unit and bind of a monad, and satisfy left unit, right unit, and associativity laws. Basically, using the arrow calculus we can understand arrows through classic well-known patterns.

In this work we propose to express quantum computations using the arrow calculus extended with monadic constructions. We show that quantum programming can be expressed using well-understood and familiar classical patterns for programming in the presence of computational effects. Interestingly, the five laws of the arrow calculus provide a convenient framework for *equational reasoning* about quantum computations (including measurements).

This work is organized as follows. The next two sections review the background material on modeling quantum computation using classical arrows. Section 4 presents the *arrow calculus*. We show the quantum arrow calculus in Section 5. We express some traditional examples of quantum computations using the quantum calculus. Additionally, we illustrate how we can use the calculus to reason about quantum programs. Section 6 concludes with a discussion of some related works. Finally, Appendix A presents the constructs of simply-typed lambda calculus, Appendix B gives an extension of the simply-typed lambda calculus with monadic constructions, and Appendix C reviews general quantum computations.

2 Classic Arrows

The simply-typed lambda calculus is an appropriate model of pure functional programming (see Appendix A). The standard way to model programming in the presence of effects is to use *monads* [10] (see Appendix B). Arrows, like monads, are used to elegantly program notions of computations in a pure functional setting. But unlike the situation with monads, which wrap the *results of computations*, arrows wrap the *computations* themselves.

From a programming point of view, *classic arrows* extend the simply-typed lambda calculus with one type and three constants satisfying nine laws (see Figure 1). The type $A \rightsquigarrow B$ denotes a computation that accepts a value of type A and returns a value of type B , possibly performing some side effects. The

three constants are: *arr*, which promotes a function to a pure arrow with no side effects; \gg , which composes two arrows; and *first*, which extends an arrow to act on the first component of a pair leaving the second component unchanged.

To understand the nine equations, we use some auxiliary functions. The function *second*, is like *first*, but acts on the second component of a pair, and $f \&\& g$, applies arrow *f* and *g* to the same argument and then pairs the results.

Fig. 1. Classic Arrows

Types	
$arr :: (A \rightarrow B) \rightarrow (A \rightsquigarrow B)$	
$(\gg) :: (A \rightsquigarrow B) \rightarrow (B \rightsquigarrow C) \rightarrow (A \rightsquigarrow C)$	
$first :: (A \rightsquigarrow B) \rightarrow (A \times C \rightsquigarrow B \times C)$	
Definitions	
$second : (A \rightsquigarrow B) \rightarrow (C \times A \rightsquigarrow C \times B)$	
$second = \lambda f. arr\ swap \gg first\ f \gg arr\ swap$	
$(\&\&) : (C \rightsquigarrow A) \rightarrow (C \rightsquigarrow B) \rightarrow (C \rightsquigarrow A \times B)$	
$(\&\&) = \lambda f. \lambda g. arr\ sup \gg first\ f \gg second\ g$	
Equations	
$(\rightsquigarrow_1) arr\ id \gg f$	$= f$
$(\rightsquigarrow_2) f \gg arr\ id$	$= f$
$(\rightsquigarrow_3) (f \gg g) \gg h$	$= f \gg (g \gg h)$
$(\rightsquigarrow_4) arr(g.f)$	$= arr\ f \gg arr\ g$
$(\rightsquigarrow_5) first(arr\ f)$	$= arr(f \times id)$
$(\rightsquigarrow_6) first(f \gg g)$	$= first\ f \gg first\ g$
$(\rightsquigarrow_7) first\ f \gg arr(id \times g)$	$= arr(id \times g) \gg first\ f$
$(\rightsquigarrow_8) first\ f \gg arr\ fst$	$= arr\ fst \gg f$
$(\rightsquigarrow_9) first(first\ f) \gg arr$	$= arr\ assoc \gg first\ f$

3 Quantum Arrows

Quantum computation is generally expressed in the framework of a Hilbert space (see Appendix C for a short review of that model). As expressive and as convenient is this framework for mathematical reasoning, it is not easily amenable to familiar programming techniques and abstractions. In recent work [5] however, we established that this general model of quantum computations (including measurements) can be structured using the category-theoretic concept of arrows. Figure 2 explains the main ideas which we elaborate on in the remainder of this section.

In the figure, we have added type definitions (i.e, type synonyms) for convenience. Type **Vec** *A* means that a vector is a function mapping elements from a vector space orthonormal basis to complex numbers (i.e., to their probability amplitudes). Type **Lin** represents a linear operator (e.g, a unitary matrix) mapping a vector of type *A* to a vector of type *B*. Note that if we *uncurry* the arguments *A* and *B*, it turns exactly into a square matrix (i.e, **Vec** (*A*, *B*)).

Type **Dens** A stands for density matrices and it is straight to build from **Vec**. Type **Super** $A B$ means a superoperator mapping a density matrix of type A to a density matrix of type B . This type can be understood by interpreting it in the same style as **Lin**.

Fig. 2. Quantum Arrows

Type Definitions	
<i>type</i> Vec A	$= A \rightarrow \mathbb{C}$
<i>type</i> Lin $A B$	$= A \rightarrow \mathbf{Vec} B$
<i>type</i> Dens A	$= \mathbf{Vec} (A, A)$
<i>type</i> Super $A B$	$= (A, A) \rightarrow \mathbf{Dens} B$
Syntax	
Types $A, B, C ::= \dots \mathbf{Vec} A \mid \mathbf{Lin} A \mid \mathbf{Dens} A \mid \mathbf{Super} A B$	
Terms $L, M, N ::= \dots \mid \mathit{return} \mid \ggg \mid \mathit{arr} \mid \ggg \mid \mathit{first}$	
Monadic Definitions	
<i>return</i> : $A \rightarrow \mathbf{Vec} A$	
<i>return</i> $a b$	$= \text{if } a == b \text{ then } 1.0 \text{ else } 0.0$
(\ggg) : $\mathbf{Vec} A \rightarrow (A \rightarrow \mathbf{Vec} B) \rightarrow \mathbf{Vec} B$	
$va \ggg f = \lambda b. \sum a (va a)(f a b)$	
Auxiliary Definitions	
<i>fun2lin</i> : $(A \rightarrow B) \rightarrow \mathbf{Lin} A B$	
<i>fun2lin</i> $f = \lambda a. \mathit{return} (f a)$	
$(\langle * \rangle)$: $\mathbf{Vec} A \rightarrow \mathbf{Vec} B \rightarrow \mathbf{Vec} (A, B)$	
$v_1 \langle * \rangle v_2 = \lambda (a, b). v_1 a * v_2 b$	
Arrow Types and Definitions	
<i>arr</i> : $(A \rightarrow B) \rightarrow \mathbf{Super} A B$	
<i>arr</i> $f = \mathit{fun2lin} (\lambda (b_1, b_2) \rightarrow (f b_1, f b_2))$	
$(\ggg) :: (\mathbf{Super} A B) \rightarrow (\mathbf{Super} B C) \rightarrow (\mathbf{Super} A C)$	
$f \ggg g = \lambda b. (f b \ggg g)$	
<i>first</i> :: $(\mathbf{Super} A B) \rightarrow (\mathbf{Super} (A \times C) (B \times C))$	
<i>first</i> $f ((b_1, d_1), (b_2, d_2)) = \mathit{permute} ((f(b_1, b_2)) \langle * \rangle \mathit{return} (d_1, d_2))$	
<i>where permute</i> $v ((b_1, b_2), (d_1, d_2)) = v ((b_1, d_1), (b_2, d_2))$	

We have defined in our previous work [5] the arrow operations for quantum computations into two levels. First we have proved that *pure* quantum states (i.e, vector states) are an instance of the concept of monads [7]. The definitions of the monadic functions are shown in Figure 2. The function *return* specifies how to construct vectors and \ggg defines the behavior of an application of matrix to a vector. Moreover we have used the auxiliary functions *fun2lin*, which converts a classical (reversible) function to a linear operator, and $\langle * \rangle$ which is the usual tensor product in vector spaces.

The function *arr* constructs a quantum superoperator from a pure function by applying the function to both vector and its dual. The composition of arrows just composes two superoperators using the monadic *bind*. The function *first* applies the superoperator f to the first component (and its dual) and leaves the second component unchanged.

We have proved in our previous work that this superoperator instance of arrows satisfy the required nine equations [5].

4 The Arrow Calculus

In this section we present the arrow calculus [9] and show the translation of the calculus to classic arrows (described in Section 2) and vice versa. The translation is important because it essentially corresponds to the denotational semantic function for the quantum version of the arrow calculus. The material of this section closely follows the original presentation in [9].

4.1 The Calculus

The arrow calculus as shown in Figure 3 extends the core lambda calculus with four constructs satisfying five laws. Type $A \rightsquigarrow B$ denotes a computation that

Fig. 3. Arrow Calculus

Syntax	
Types	$A, B, C ::= \dots \mid A \rightsquigarrow B$
Terms	$L, M, N ::= \dots \mid \lambda^\bullet x.Q$
Commands	$P, Q, R ::= L \bullet P \mid [M] \mid \text{let } x = P \text{ in } Q$
Types	
$\frac{\Gamma; x : A \vdash Q!B}{\Gamma \vdash \lambda^\bullet x.Q : A \rightsquigarrow B}$	$\frac{\Gamma \vdash L : A \rightsquigarrow B \quad \Gamma; \Delta \vdash M : A}{\Gamma; \Delta \vdash L \bullet M!B}$
$\frac{\Gamma, \Delta \vdash M : A}{\Gamma; \Delta \vdash [M]!A}$	$\frac{\Gamma; \Delta \vdash P!A \quad \Gamma; \Delta, x : A \vdash Q!B}{\Gamma; \Delta \vdash \text{let } x = P \text{ in } Q!B}$
Laws	
$(\beta^{\rightsquigarrow})$	$(\lambda^\bullet x.Q) \bullet M = Q[x := M]$
$(\eta^{\rightsquigarrow})$	$\lambda^\bullet x.(L \bullet [x]) = L$
(left)	$\text{let } x = [M] \text{ in } Q = Q[x := M]$
(right)	$\text{let } x = P \text{ in } [x] = P$
$(\text{assoc}) \text{let } y = (\text{let } x = P \text{ in } Q) \text{ in } R = \text{let } x = P \text{ in } (\text{let } y = Q \text{ in } R)$	

accepts a value of type A and returns a value of type B , possibly performing some side effects.

There are two syntactic categories. Terms are ranged over by L, M, N , and commands are ranged over by P, Q, R . In addition to the terms of the core lambda calculus, there is one new term form: arrow abstraction $\lambda^\bullet x.Q$. There are three command forms: arrow application $L \bullet M$, arrow unit $[M]$ (which resembles unit in a monad), and arrow bind $\text{let } x = P \text{ in } Q$ (which resembles bind in a monad).

In addition to the term typing judgment $\Gamma \vdash M : A$ there is also a command typing judgment $\Gamma; \Delta \vdash P!A$. An important feature of the arrow calculus is that

the command type judgment has two environments, Γ and Δ , where variables in Γ come from ordinary lambda abstractions $\lambda x.N$, while variables in Δ come from arrow abstraction $\lambda^\bullet x.Q$.

Arrow abstraction converts a command into a term. Arrow abstraction closely resembles function abstraction, save that the body Q is a command (rather than a term) and the bound variable x goes into the second environment (separated from the first by a semicolon).

Conversely, arrow application, $L \bullet M!B$ embeds a term into a command. Arrow application closely resembles function application. The arrow to be applied is denoted by a term, not a command; this is because there is no way to apply an arrow that is itself yielded. This is why there are two different environments, Γ and Δ : variables in Γ may denote arrows that are applied to arguments, but variables in Δ may not.

Arrow unit, $[M]!A$, promotes a term to a command. Note that in the hypothesis there is a term judgment with one environment (i.e, there is a comma between Γ and Δ), while in the conclusion there is a command judgment with two environments (i.e, there is a semicolon between Γ and Δ).

Lastly, using `let`, the value returned by a command may be bound.

Arrow abstraction and application satisfy beta and eta laws, (β^\sim) and (η^\sim) , while arrow unit and bind satisfy left unit, right unit, and associativity laws, (left), (right), and (assoc). The beta law equates the application of an abstraction to a bind; substitution is not part of beta, but instead appears in the left unit law. The (assoc) law has the usual side condition, that x is not free in R .

4.2 Translation

The translation from the arrow calculus to classic arrows, shown below, gives a denotational semantics for the arrow calculus.

$$\begin{aligned} [\lambda^\bullet x.Q] &= [Q]_x \\ [L \bullet M]_\Delta &= \text{arr}(\lambda\Delta.[M]) \gg [L] \\ [[M]]_\Delta &= \text{arr}(\lambda\Delta.[M]) \\ [\text{let } x = P \text{ in } Q]_\Delta &= (\text{arr } id \ \&\& \ [P]_\Delta) \gg [Q]_{\Delta,x} \end{aligned}$$

An arrow calculus term judgment $\Gamma \vdash M : A$ maps into a classic arrow judgment $\Gamma \vdash [M] : A$, while an arrow calculus command judgment $\Gamma; \Delta \vdash P!A$ maps into a classic arrow judgment $\Gamma \vdash [P]_\Delta : \Delta \rightsquigarrow A$. Hence, the denotation of a command is an arrow, with arguments corresponding to the environment Δ and result of type A .

We omitted the translation of the constructs of core lambda calculus as they are straightforward homomorphisms. The translation of the arrow abstraction $\lambda^\bullet x.Q$ just undoes the abstraction and call the interpretation of Q using x . Application $L \bullet P$ translates to \gg , $[M]$ translates to `arr` and `let $x = P$ in Q` translates to pairing `&&` (to extend the environment with P) and composition \gg (to then apply Q).

The inverse translation, from classic arrows to the arrow calculus is defined as:

$$\begin{aligned}
[arr]^{-1} &= \lambda f. \lambda^{\bullet} x. [f x] \\
[(\gg)]^{-1} &= \lambda f. \lambda g. \lambda^{\bullet} x. g \bullet (f \bullet x) \\
[first]^{-1} &= \lambda f. \lambda^{\bullet} z. \text{let } x = f \bullet \text{fst } z \text{ in } [(x, \text{snd } z)]
\end{aligned}$$

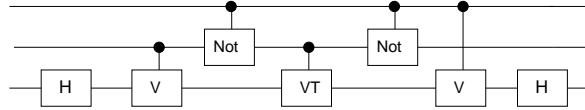
Again we omitted the translation of the constructs of core lambda calculus as they are straightforward homomorphisms. Each of the three constants from classic arrows translates to an appropriate term in the arrow calculus.

5 The Arrow Calculus as a Quantum Programming Language

In this section we discuss how the arrow calculus can be used as a quantum programming language.

We start by showing quantum programs using the standard quantum circuit notation. The lines carry quantum bits. The values flow from left to right in steps corresponding to the alignment of the boxes which represent quantum gates. Gates connected via bullets to another wire are called *controlled operations*, that is, the wire with the bullet conditionally controls the application of the gate. The circuit in Figure 4 represents a quantum program for the Toffoli gate. Using the

Fig. 4. Circuit for the Toffoli gate



classic arrows approach for quantum programming presented in Section 3 and using the type of booleans, **Bool**, as the orthonormal basis for the qubit, this program would be coded as follows:

```

toffoli :: Super (Bool, Bool, Bool) (Bool, Bool, Bool)
toffoli = arr (\(a0, b0, c0) -> (c0, (a0, b0))) >>>
  (first H >>> arr (\(c1, (a0, b0)) -> ((b0, c1), a0))) >>>
  (first cV >>> arr (\(b1, c2), a0 -> ((a0, b1), c2))) >>>
  (first cNot >>> arr (\(a1, b2), c2 -> ((b2, c2), a1))) >>> ...

```

As already noted by Paterson [11] this notation is cumbersome for programming. This is a “point-free” notation, rather different from the usual way of writing functional programs, with λ and **let**. Paterson introduced syntactic sugar for arrows, which we have used in our previous work [5]. However, the notation simply abbreviates terms built from the three constants, and there is no claim about reasoning with arrows. Using the *quantum arrow calculus* presented in Figure 5, this program would be like:

```

toffoli :: Super (Bool, Bool, Bool) (Bool, Bool, Bool)
toffoli =  $\lambda^\bullet.(x, y, z).\text{let } z' = H \bullet z \text{ in}$ 
          $\text{let } (y', z'') = cV \bullet (y, z') \text{ in}$ 
          $\text{let } (x', y'') = cNot \bullet (x, y') \text{ in } \dots$ 

```

This style is more convenient and elegant as it is very similar to the usual familiar classical functional programming and is amenable to formal reasoning in a convenient way. Consider, for instance, the program which applies the quantum not gate twice. That is obviously equivalent to identity. To do such a simple proof using the *classic arrows* we need to learn how to use the *nine* arrow laws and also to recover the definitions of the functions *arr*, \gg and *first* for quantum computations presented in Figure 2.

The action of the quantum not gate, **QNot**, is to swap the amplitude probabilities of the qubit. For instance, **QNot** applied to $|0\rangle$ returns $|1\rangle$, and vice versa. But **QNot** applied to $\alpha|0\rangle + \beta|1\rangle$ returns $\alpha|1\rangle + \beta|0\rangle$.

Given the classical definition of not as follows:

```
not =  $\lambda x.\text{if } x == \text{True} \text{ then False else True} : \mathbf{Bool} \rightarrow \mathbf{Bool}$ 
```

Using the arrow calculus, the **QNot** would be written as:

```
QNot =  $\lambda^\bullet y.[\text{not } y] : \mathbf{Super Bool Bool}$ .
```

Then, the program which applies the **QNot** twice, would be:

```
 $\Gamma \vdash \lambda^\bullet x.\text{let } w = (\lambda^\bullet z.[\text{not } z]) \bullet x \text{ in } (\lambda^\bullet y.[\text{not } y]) \bullet w$ 
```

Again the syntax, with arrow abstraction and application, resembles lambda calculus. Now we can use the intuitive arrow calculus laws (from Figure 3) to prove the obvious equivalence of this program with identity. The proof follows the same style of the proofs in classical functional programming.

$$\begin{aligned}
\lambda^\bullet x.\text{let } w &= (\lambda^\bullet z.[\text{not } z]) \bullet x \text{ in } (\lambda^\bullet y.[\text{not } y]) \bullet w && \stackrel{(\beta^\sim)}{=} \\
\lambda^\bullet x.\text{let } w &= [\text{not } x] \text{ in } (\lambda^\bullet y.[\text{not } y]) \bullet w && \stackrel{(\text{left})}{=} \\
\lambda^\bullet x.(\lambda^\bullet y.[\text{not } y]) &\bullet (\text{not } x) && \stackrel{(\beta^\sim)}{=} \\
\lambda^\bullet x.[\text{not}(\text{not } x)] &&& \stackrel{=def.not}{=} \\
\lambda^\bullet x.[x] &&&
\end{aligned}$$

It is interesting to note that we have two ways for defining superoperators. The first way is going directly from classical functions to superoperators as we did above for *not*, using the default definition of *arr*. The other way is going from the monadic pure quantum functions to superoperators. As monads are a special case of arrows [6] there is *always* a translation from monadic functions to arrows. Hence, any **Lin** $A B$ is a special case of **Super** $A B$.

Hence, we construct the quantum arrow calculus in Figure 5 in three levels. First we inherit all the constructions from simply-typed lambda calculus with the type of booleans and with classical let and if (see Appendix A). Then we

Fig. 5. Quantum Arrow Calculus

Syntax

Types $A, B, C ::= \dots \mid \mathbf{Bool} \mid \mathbf{Dens} A \mid \mathbf{Vec} A \mid \mathbf{Super} A B$

Terms $L, M, N ::= [T] \mid \mathbf{let} x = M \mathbf{in} N \mid \lambda^\bullet x. Q \mid + \mid -$

Commands $P, Q, R ::= L \bullet P \mid [M] \mid \mathbf{let} x = P \mathbf{in} Q \mid \mathbf{meas} \mid \mathbf{trL}$

Monad Types

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash [M] : \mathbf{Vec} A} \quad \frac{\Gamma \vdash M : \mathbf{Vec} A \quad \Gamma, x : A \vdash N : \mathbf{Vec} B}{\Gamma \vdash \mathbf{let} x = M \mathbf{in} N : \mathbf{Vec} B}$$

$$\frac{\Gamma \vdash M, N : \mathbf{Vec} A}{\Gamma \vdash M + N : \mathbf{Vec} A} \quad \frac{\Gamma \vdash M, N : \mathbf{Vec} A}{\Gamma \vdash M - N : \mathbf{Vec} A}$$

Arrow Types

$$\frac{\Gamma; x : A \vdash Q! \mathbf{Dens} B}{\Gamma \vdash \lambda^\bullet x. Q : \mathbf{Super} A B} \quad \frac{\Gamma \vdash L : \mathbf{Super} A B \quad \Gamma; \Delta \vdash M : A}{\Gamma; \Delta \vdash L \bullet M! \mathbf{Dens} B}$$

$$\frac{\Gamma; \Delta \vdash M : A}{\Gamma; \Delta \vdash [M]! \mathbf{Dens} A} \quad \frac{\Gamma; \Delta \vdash P! \mathbf{Dens} A \quad \Gamma; \Delta, x : A \vdash Q! \mathbf{Dens} B}{\Gamma; \Delta \vdash \mathbf{let} x = P \mathbf{in} Q! \mathbf{Dens} B}$$

$$\frac{}{\Gamma; x : A \vdash \mathbf{meas} ! \mathbf{Dens} (A, A)} \quad \frac{}{\Gamma; x : (A, B) \vdash \mathbf{trL} ! \mathbf{Dens} B}$$

add the monadic *unit*, $[]$, to build pure vectors (over booleans), **let** to sequence computations with vectors, and plus and minus to add and subtract vectors (the monadic calculus [7] with its laws is presented in Appendix B). Finally, we add the constructions of the arrow calculus. The appeal of using the arrows approach is because we can express measurement operations (i.e, extract classical information from the quantum system) inside the formalism. Therefore, we have two computations for measurements on mixed states, **meas** and **trL**. The computation **meas** returns a classical value and a post-measurement state of the quantum system. The computation **trL** *traces out* or *projects* part of the quantum state (the denotation of these operations is provided in Appendix D).

To exemplify the use of the monadic constructions, consider, for example, the hadamard quantum gate, which is the source of superpositions. For instance, hadamard applied to $|0\rangle$ returns $|0\rangle + |1\rangle$, and applied to $|1\rangle$ returns $|0\rangle - |1\rangle$. But, hadamard applied to $|0\rangle + |1\rangle$ returns $|0\rangle$, as it is a reversible gate. To define this program in the quantum arrow calculus, we just need to define its work for the basic values, $|0\rangle$ and $|1\rangle$, as follows:

$$\mathbf{hadamard} = \lambda x. \mathbf{if} x == \mathbf{True} \mathbf{then} [False] - [True] \mathbf{else} [False] + [True] : \mathbf{Lin Bool Bool}$$

Then, the superoperator would be:

$$\text{Had} = \lambda^\bullet y. [\text{hadamard } y] : \mathbf{Super\ Bool\ Bool}$$

Another interesting class of operations are the so-called *quantum controlled operations*. For instance, the controlled not, **Cnot**, receives two qubits and applies a not operation on the second qubit depending on the value of the first qubit. Again, we just need to define it for the basic quantum values:

$$\begin{aligned} \text{cnot} = \lambda(x, y). \text{if } x \text{ then } [(x, \text{not } y)] \\ \text{else } [(x, y)] : \mathbf{Lin\ (Bool, Bool)\ (Bool, Bool)} \end{aligned}$$

Again, the superoperator of type **Super (Bool, Bool) (Bool, Bool)** would be $\text{Cnot} = \lambda^\bullet(x, y). [\text{cnot } (x, y)]$.

The motivation of using superoperators is that we can express *measurement* operations inside of the formalism. One classical example of quantum algorithm which requires a measurement operation is the quantum teleportation [4]. It allows the transmission of a qubit to a partner with whom is shared an entangled pair. Below we define the two partners of a teleportation algorithm.

$$\begin{aligned} \text{Alice} : \mathbf{Super\ (Bool, Bool)\ (Bool, Bool)} \\ \text{Alice} = \lambda^\bullet(x, y). \text{let } (x', y') = \text{Cnot} \bullet (x, y) \text{ in} \\ \quad \text{let } q = (\text{Had} \bullet x', y') \text{ in} \\ \quad \quad \text{let } (q', v) = \text{meas} \bullet q \text{ in trL} \bullet (q, v) \end{aligned}$$

$$\begin{aligned} \text{Bob} : \mathbf{Super\ (Bool, Bool, Bool)\ Bool} \\ \text{Bob} = \lambda^\bullet(x, y, z). \text{let } (z', x') = \text{Cnot} \bullet (z, x) \text{ in} \\ \quad \text{let } (y', x'') = (\text{Cz} \bullet (y, x')) \text{ in trL} \bullet ((y', z'), x'') \end{aligned}$$

6 Conclusion

We have presented a lambda calculus for general quantum programming that builds on well-understood and familiar programming patterns and reasoning techniques. Besides supporting an elegant functional programming style for quantum computations, the quantum arrow calculus allows reasoning about *general* or *mixed* quantum computations. This is the first work proposing reasoning about *mixed* quantum computations. The equations of the arrow calculus plus the equations of the monadic calculus provide indeed a powerful mechanism to make proofs about quantum programs. In [12] we have proposed very similar reasoning techniques, however for *pure* quantum programs. Also, in [13] the author presents a quantum lambda calculus based on linear logic, but just for pure quantum computations.

Acknowledgements

We thank Jeremy Yallop for very helpful comments.

References

1. Nielsen, M.A., Chuang, I.L.: Quantum Computation and Quantum Information. Cambridge University Press (2000)
2. Shor, P.W.: Algorithms for quantum computation: Discrete logarithms and factoring. In: Proc. IEEE Symposium on Foundations of Computer Science. (1994) 124–134
3. Grover, L.K.: A fast quantum mechanical algorithm for database search. In: Proc. 28., Annual ACM Symposium on Theory of Computing. (1996) 212–219
4. Bennett, C.H., Brassard, G., Crepeau, C., Jozsa, R., Peres, A., Wootters, W.: Teleporting an unknown quantum state via dual classical and EPR channels. Phys Rev Lett (1993) 1895–1899
5. Vizzotto, J.K., Altenkirch, T., Sabry, A.: Structuring quantum effects: Superoperators as arrows. Journal of Mathematical Structures in Computer Science: special issue in quantum programming languages **16** (2006) 453–468
6. Hughes, J.: Generalising monads to arrows. Science of Computer Programming **37** (May 2000) 67–111
7. Moggi, E.: Computational lambda-calculus and monads. In: Proceedings of the Fourth Annual Symposium on Logic in computer science, IEEE Press (1989) 14–23
8. McBride, C., Paterson, R.: Applicative programming with effects. J. Funct. Program. **18**(1) (2008) 1–13
9. Lindley, S., Wadler, P., Yallop, J.: The arrow calculus (functional pearl). In: International Conference on Functional Programming. (2008)
10. Moggi, E.: Notions of computation and monads. Information and Computation **93**(1) (1991) 55–92
11. Paterson, R.: A new notation for arrows. In: Proc. International Conference on Functional Programming. (September 2001) 229–240
12. Altenkirch, T., Grattage, J., Vizzotto, J.K., Sabry, A.: An algebra of pure quantum programming. Electron. Notes Theor. Comput. Sci. **170** (2007) 23–47
13. Tonder, A.v.: A lambda calculus for quantum computation. SIAM J. Comput. **33**(5) (2004) 1109–1135
14. : MonadPlus. <http://www.haskell.org/hawiki/MonadPlus> (2005)
15. Hinze, R.: Deriving backtracking monad transformers. In: ICFP '00: Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming, ACM Press (2000) 186–197
16. Aharonov, D., Kitaev, A., Nisan, N.: Quantum circuits with mixed states. In: Proceedings of the thirtieth annual ACM symposium on Theory of computing, New York: ACM Press (1998) 20–30
17. Selinger, P.: Towards a quantum programming language. Journal of Mathematical Structures in Computer Science: special issue in quantum programming languages **16** (2006) 527–586

A Simply-Typed Lambda Calculus

The simply-typed lambda calculus with the type of booleans, and with `let` and `if` is shown in Figure 6. Let A, B, C range over types, L, M, N range over terms, and Γ, Δ range over environments. A type judgment $\Gamma \vdash M : A$ indicates that in environment Γ term M has type A . As presented in the arrow calculus [9], we are using a Curry formulation, eliding types from terms.

Fig. 6. Simply-typed Lambda Calculus

Syntax			
Types	$A, B, C ::= \mathbf{Bool} \mid A \times B \mid A \rightarrow B$		
Terms	$L, M, N ::= x \mid \mathbf{True} \mid \mathbf{False} \mid (M, N) \mid \mathbf{fst} L \mid \mathbf{snd} L \mid \lambda x. N \mid L M$ $\quad \text{let } x = M \text{ in } N \mid \text{if } L \text{ then } M \text{ else } N$		
Environments Γ, Δ	$::= x_1 : A_1, \dots, x_n : A_n$		
Types			
	$\frac{}{\emptyset \vdash \mathbf{False} : \mathbf{Bool}}$	$\frac{}{\emptyset \vdash \mathbf{True} : \mathbf{Bool}}$	$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A}$
$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash (M, N) : A \times B}$	$\frac{\Gamma \vdash L : A \times B}{\Gamma \vdash \mathbf{fst} L : A}$	$\frac{\Gamma \vdash L : A \times B}{\Gamma \vdash \mathbf{snd} L : B}$	
$\frac{\Gamma, x : A \vdash N : B}{\Gamma \vdash \lambda x. N : A \rightarrow B}$		$\frac{\Gamma \vdash L : A \rightarrow B \quad \Gamma \vdash M : A}{\Gamma \vdash L M : B}$	
$\frac{\Gamma \vdash M : A \quad \Gamma, x : A \vdash N : B}{\Gamma \vdash \text{let } x = M \text{ in } N : B}$		$\frac{\Gamma \vdash L : \mathbf{Bool} \quad \Gamma \vdash M, N : B}{\Gamma \vdash \text{if } L \text{ then } M \text{ else } N : B}$	
Laws			
(β_1^x)	$\mathbf{fst} (M, N)$	$=$	M
(β_2^x)	$\mathbf{snd} (M, N)$	$=$	N
(η^x)	$(\mathbf{fst} L, \mathbf{snd} L)$	$=$	L
(β^{\rightarrow})	$(\lambda x. N) M$	$=$	$N[x := M]$
(η^{\rightarrow})	$\lambda x. (L x)$	$=$	L
(let)	$\text{let } x = M \text{ in } N$	$=$	$N[x := M]$
(β_1^{if})	$\text{if } \mathbf{True} \text{ then } M \text{ else } N$	$=$	M
(β_2^{if})	$\text{if } \mathbf{False} \text{ then } M \text{ else } N$	$=$	N

B Monadic Calculus

The simply-typed lambda calculus presented in Appendix A is the foundation of purely functional programming languages. In this section we show the *monadic calculus* [7], which also models monadic effects. A monad is represented using a type constructor for computations m and two functions: $\text{return} :: a \rightarrow m a$ and $\gg :: m a \rightarrow (a \rightarrow m b) \rightarrow m b$. The operation \gg (pronounced “bind”) specifies how to sequence computations and return specifies how to lift values to computations. From a programming perspective, a *monad* is a construct to structure *computations*, in a functional environment, in terms of values and sequence of computations using those values.

The monadic calculus extends the simply-typed lambda calculus with the constructs in Figure 7. Unit and bind satisfy left unit, right unit, and associativity laws, (left), (right), and (assoc).

Fig. 7. Monadic Calculus

Syntax
Types $A, B, C ::= \dots \mid \mathbf{M} A$
Terms $L, M, N ::= \dots \mid [M] \mid \text{let } x = M \text{ in } N \mid \text{mzero} \mid + \mid -$

Monadic Types

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash [M] : \mathbf{M} A} \quad \frac{\Gamma \vdash M : \mathbf{M} A \quad \Gamma, x : A \vdash N : \mathbf{M} B}{\Gamma \vdash \text{let } x = M \text{ in } N : \mathbf{M} B}$$

MonadPlus Types

$$\frac{}{\Gamma \vdash \text{mzero} : \mathbf{M} A} \quad \frac{\Gamma \vdash M, N : \mathbf{M} A}{\Gamma \vdash M + N : \mathbf{M} A}$$

Laws
(left) $\text{let } x = [L] \text{ in } N = N[x := L]$
(right) $\text{let } x = L \text{ in } [x] = L$
(assoc) $\text{let } y = (\text{let } x = L \text{ in } N) \text{ in } T = \text{let } x = L \text{ in } (\text{let } y = N \text{ in } T)$

MonadPlus Laws
 $\text{mzero} + a = a$
 $a + \text{mzero} = a$
 $a + (b + c) = (a + b) + c$
 $\text{let } x = \text{mzero} \text{ in } T = \text{mzero}$
 $\text{let } x = (M + N) \text{ in } T = (\text{let } x = M \text{ in } T) + (\text{let } x = N \text{ in } T)$

Beyond the three monad laws discussed above, some monads obey the `-MonadPlus` laws. The `MonadPlus` interface provides two primitives, `mzero` and `+` (called `mplus`), for expressing choices. The command `+` introduces a choice junction, and `mzero` denotes failure.

The precise set of laws that a `MonadPlus` implementation should satisfy is not agreed upon [14], but in [15] is presented a reasonable agreement on the laws. We use in Figure 7 the laws introduced by [15].

The intuition behind these laws is that `MonadPlus` is a disjunction of goals and \gg is a conjunction of goals. The conjunction evaluates the goals from left-to-right and is not symmetric.

C General Quantum Computations

Quantum computation, as its classical counterpart, can be seen as processing of information using quantum systems. Its basic idea is to encode data using quantum bits (qubits). In quantum theory, considering a *closed* quantum system, the qubit is a *unit* vector living in a complex inner product vector space know as *Hilbert space* [1]. We call such a vector a *ket* (from *Dirac's notation*) and denote it by $|v\rangle$ (where v stands for elements of an orthonormal basis), a column vector. Differently from the classical bit, the qubit can be in a *superposition* of the two basic states written as $\alpha|0\rangle + \beta|1\rangle$, or

$$\begin{pmatrix} \alpha \\ \beta \end{pmatrix}$$

with $|\alpha|^2 + |\beta|^2 = 1$. Intuitively, one can think that a qubit can exist as a 0, a 1, or simultaneously as both 0 and 1, with numerical coefficient (i.e., the probability amplitudes α and β) which determines the probability of each state. The quantum superposition phenomena is responsible for the so called “quantum parallelism.”

Operations acting on those *isolated* or *pure* quantum states are linear operations, more specifically *unitary matrices* S . A matrix A is called *unitary* if $S^*S = I$, where S^* is the adjoint of S , and I is the identity. Essentially, those unitary transformations act on the quantum states by changing their probability amplitudes, without loss of information (i.e., they are reversible). The application of a unitary transformation to a state vector is given by usual matrix multiplication.

Unfortunately in this model of quantum computing, it is difficult or impossible to deal formally with another class of quantum effects, including measurements, decoherence, or noise.

Measurements are critical to some quantum algorithms, as they are the only way to extract *classical* information from quantum states.

A *measurement* operation projects a quantum state like $\alpha|0\rangle + \beta|1\rangle$ onto the basis $|0\rangle, |1\rangle$. The outcome of the measurement is not deterministic and it is given by the probability amplitude, i.e., the probability that the state after the measurement is $|0\rangle$ is $|\alpha|^2$ and the probability that the state is $|1\rangle$ is $|\beta|^2$. If the value of the qubit is initially unknown, then there is no way to determine α and β with that single measurement, as the measurement may *disturb* the state. But, *after* the measurement, the qubit is in a *known* state; either $|0\rangle$ or $|1\rangle$. In fact, the situation is even more complicated: measuring part of a quantum state collapses not only the measured part but any other part of the global state with which it is *entangled*. In an entangled state, two or more qubits have to be described with reference to each other, even though the individuals may be spatially separated ⁴.

There are several ways to deal with measurements in quantum computing, as summarized in our previous work [5]. To deal formally and elegantly with measurements, the state of the computation is represented using a *density matrix* and the operations are represented using *superoperators* [16]. Using these notions, the *projections* necessary to express measurements become expressible within the model.

Intuitively, density matrices can be understood as a statistical perspective of the state vector. In the density matrix formalism, a quantum state that used to be modeled by a vector $|v\rangle$ is now modeled by its outer product $|v\rangle\langle v|$, where $\langle v|$ is the row vector representing the adjoint (or dual) of $|v\rangle$. For instance, the state of a quantum bit $|v\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$ is represented by the density matrix:

⁴ For more detailed explanation about entangled, see [1].

$$\begin{pmatrix} \frac{1}{2} & -\frac{1}{2} \\ -\frac{1}{2} & \frac{1}{2} \end{pmatrix}$$

Note that the main diagonal shows the classical probability distribution of basic quantum states, that is, these state has $\frac{1}{2}$ of probability to be $|0\rangle$ and $\frac{1}{2}$ of probability to be $|1\rangle$.

However, the appeal of density matrices is that they can represent states other than the pure ones above. In particular if we perform a measurement on the state represented above, we should get $|0\rangle$ with probability $1/2$ or $|1\rangle$ with probability $1/2$. This information, which cannot be expressed using vectors, can be represented by the following density matrix:

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

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

The two kinds of quantum operations, namely unitary transformation and measurement, can both be expressed with respect to density matrices [17]. Those operations now mapping density matrices to density matrices are called *superoperators*. A unitary transformation S maps a pure quantum state $|u\rangle$ to $S|u\rangle$. Thus, it maps a pure density matrix $|u\rangle\langle u|$ to $S|u\rangle\langle u|S^*$. Moreover, a unitary transformation extends linearly to mixed states, and thus, it takes any mixed density matrix A to SAS^* .

As one can observe in the resulting matrix above, to execute a measurement corresponds to setting a certain region of the input density matrix to zero.

D Definition of Measurement Operations

In this section we present the denotations of the programs for measurements, *trl* and *meas*, added to the quantum arrow calculus.

```
trl :: Super (A, B) B
trl((a1, b1), (a2, b2)) = if a1 == a2 then return(b1, b2) else mzero
```

```
meas :: Super A (A, A)
meas(a1, a2) = if a1 == a2 then return((a1, a1), (a1, a1)) else mzero
```

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