QJava: A Monadic Java Library for Quantum Programming

Bruno Crestani Calegaro [1]
Juliana Kaizer Vizzotto [2]

Data submissão: 22.10.2014
Data aceitação: 25.04.2015

Seção melhores artigos WEIT-2013, Rio Grande, Brasil, 2013.

Abstract:
To help the understanding and development of quantum algorithms there is an effort focused on the investigation of new semantic models and programming languages for quantum computing. Researchers in computer science have the challenge of developing programming languages to support the creation, analysis, modeling and simulation of high level quantum algorithms. Based on previous works that use monads inside the programming language Haskell to elegantly explain the odd characteristics of quantum computation (like superposition and entanglement), in this work we present a monadic Java library for quantum programming. We use the extension of the programming language Java called BGGA Closure, that allow the manipulation of anonymous functions (closures) inside Java. We exemplify the use of the library with an implementation of the Toffoli quantum circuit.

1 Introduction

Quantum computing is an emerging technology, based on the idea of a computer that has access to, and can manipulate, quantum information, i.e., information coded in a state of a quantum physical system. The idea of a quantum computer that takes advantage of the effects of quantum mechanics was theoretically introduced by Richard Feynman in 1982 [1]. After that, David Deutsch [2] defined a quantum Turing machine, suggesting that if quantum computers could solve quantum mechanical problems more quickly than classical computers, they might also could solve classical problems more quickly. In 1994, Peter Shor [3] invented a quantum algorithm to solve the problem of factoring large numbers in polynomial time on a quantum computer. Most famous cryptography techniques are based on the fact that classical algorithms to factor large numbers take exponential time [4]. A practical quantum

[1]Instituto Federal de Santa Catarina, IFSC
{bruno.calegaro@@ifsc.edu.br}
[2]Universidade Federal de Santa Maria, UFSM
{juvizzotto@@inf.ufsm.br}

computer capable of performing Shor's algorithm would be able to break current cryptography techniques. The invention of this algorithm motivated the topic of quantum computing and stimulated researchers around the world to investigate the creation of practical quantum computers and of new quantum algorithms applied to different areas.

Also, to help the understanding and development of quantum algorithms there is an effort focused on the investigation of new semantic models and programming languages for quantum computing. Researchers in computer science have the challenge of developing programming languages to support the creation, analysis, modeling and simulation of high level quantum algorithms.

In the work [5], the authors presented a semantic model for pure quantum computing (or more precisely, arbitrary linear functions over complex vector spaces) using the concept of monads [6]. Monads are an idea from category theory used to model side effects in classical functional programming languages. The appeal of using monads is that they can elegantly model side effects, such as global state and input/output, in the context of pure functional languages. The use of monads allows to elegantly explain the odd characteristics of quantum computing (like superposition) using tools applied in the development of traditional programming languages (not quantum).

Java is a general propose objected oriented programming language that enables secure and high performance software development on multiple platforms. It is one of the most widely used programming language in software development. Based on the previous monadic model for quantum programming mentioned above, we present in this paper a monadic Java library for quantum programming. We use the extension of Java called BGGA Closure, that allow the manipulation of anonymous functions (closures) inside Java. We exemplify the use of the library with an implementation of the Toffoli quantum circuit.

The practical contribution of our work is a monadic Java library, called QJava, that can be used to program pure quantum algorithms. Moreover the manuscript has a very detailed explanation of monads in the context of functional programming languages e its application to quantum computing.

The present work is organized as follows: Section 2 briefly reviews the basic notions involved in quantum programming. In Section 3 we explain the main concepts about monads applied to semantics of programming languages and the definition of a quantum monad. Section 4 presents the library QJava and examples. Finally, we conclude in Section 5.

2 Quantum Programming

Quantum computing is an information processing paradigm, concerning hardware and software, where information/data (and maybe control flow) can be encoded in the state of a *quantum* physical system. Quantum programming is about quantum software design, i.e., to design algorithms considering the quantum computing paradigm. In this section we briefly review the formalism needed to understand the basic principles of quantum programming. All the concepts presented in this section also apply to quantum computing in general, however we do not discuss quantum hardware here. For a complete reference on quantum computing see [7].

2.1 Quantum Bits

The basic unit of information in classical computing is the traditional bit, a binary classical physical system. The bit is usually represented as an scalar that belongs to $\{1,0\}$.

In quantum computing the basic unit information is represented by a *quantum bit*, or qubit, a binary *quantum* physical system. The qubit is a vector usually represented using the Dirac *braket* [3] notation:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle.$$

The Dirac notation has the advantage that it labels the basis vectors explicitly. The basic states $|0\rangle$ and $|1\rangle$ can be explained by analogy with the classical bit, i.e., form a two-level system and are an orthonormal basis for the vector space where the qubit lives (usually called the standard or computational basis). The coefficients or also called *probability amplitudes*, α and β, are complex numbers, such that $|\alpha|^2 + |\beta|^2 = 1$. In other words the qubit can be formalized as a vector in two-dimensional complex vector space (Hilbert space), with norm (size) equals 1.

The qubit can be seen as a column vector:

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

In this representation it is important to note that there is an ordering for the basis vectors which is relevant: the coefficient for $|0\rangle$, in this specific case equals α, always appear in the first row (the row for $|0\rangle$) and the coefficient for $|1\rangle$, in this specific case equals β, in the second row (the row for $|1\rangle$). This convention can be illustrated as follows:

$$\begin{array}{c} |0\rangle \\ |1\rangle \end{array} \begin{pmatrix} \alpha \\ \beta \end{pmatrix}$$

[3] The name *braket* comes from the convention that a column vector is called a “ket” and is denoted by $|\ \rangle$ and a row vector is called a “bra” and is denoted by $\langle\ |$.

As an example, the classical bit 0 can be represented as the basis state $|0\rangle = 1|0\rangle + 0|1\rangle$ and the classical bit 1 can be represented as the basis state $|1\rangle = 0|0\rangle + 1|1\rangle$. These two states can also be represented as the following column vectors, respectively:

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

Any other state with different values for α and β is said to be in a *quantum superposition* of $|0\rangle$ and $|1\rangle$, for instance the state $1/\sqrt{2}|0\rangle + 1/\sqrt{2}|1\rangle$.

The interpretation of the probability amplitudes α and β can be given by the following: *when we interact or measure a quantum state like $\alpha|0\rangle + \beta|1\rangle$ we will see/get the state $|0\rangle$ with probability $|\alpha|^2$ and the state $|1\rangle$ with probability $|\beta|^2$.*

Now lets understand a composite quantum state with two or more qubits like above. Again, in analogy with the classical bit, consider a state with two bits, we would have four alternatives: $00, 01, 10, 11$. Then, the state of a pair of qubits is a linear combination of these four classical states:

$$\alpha|00\rangle + \gamma|01\rangle + \delta|10\rangle + \beta|11\rangle$$

This two qubit state can also be represented as a column vector:

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

Note again that there is an ordering for the basis vectors which is relevant: the coefficient for $|00\rangle$, in this specific case equals α, always appear in the first row (the row for $|00\rangle$), the the coefficient for $|01\rangle$, in this specific case equals γ, in the second row (the row for $|01\rangle$), the coefficient for $|10\rangle$, in this specific case equals δ, in the third row (the row for $|10\rangle$), and the coefficient for $|11\rangle$, in this specific case equals β, in the last row.

If $q = \alpha|0\rangle + \beta|1\rangle$ and $p = \gamma|0\rangle + \delta|1\rangle$ are two independent quantum bits, then we can form a combined state using the *tensor operation* on vector spaces, $\otimes$, defined as follows:

$$q \otimes p = \alpha\gamma|00\rangle + \alpha\delta|01\rangle + \beta\gamma|10\rangle + \beta\delta|11\rangle$$

However, there are some combined quantum bits which are not of the form $q \otimes p$. For instance, the state:

$$1/\sqrt{2}|00\rangle + 1/\sqrt{2}|11\rangle$$

is clearly not of the form $q \otimes p$, for any q and p. This kind of *combined* state which can not be described using the tensor product operation is called *entangled*.

2.2 Quantum Operations

There are two kinds of operations on quantum bits: *unitary transformations* [4] and *measurement*. The first kind corresponds to reversible operations that change the state of a qubit without loss of information and the second kind *observes* the state to find out its value. A measurement *collapses* the quantum state making it impossible to recover the original state after the observation.

2.2.1 Unitary Transformations As we explained above a qubit is a unit vector in a two dimensional complex vector space (in general spanned by the computational basis $|0\rangle$ and $|1\rangle$). A unitary transformation changes the state of the unit vector without loss of information. As a qubit is usually represented by a column vector a unitary transformation is in general represented by a unitary matrix. That is, for a state with *n* qubits we will have a $2^n \times 2^n$ matrix.

Suppose a state $|\phi\rangle = \alpha|0\rangle + \beta|1\rangle$, then the application of a unitary transformation T can be represented as usual matrix/vector multiplication:

$$T|\phi\rangle = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix}$$

As an example of a unitary transformation acting in one qubit, consider the quantum NOT transformation, which flips the amplitudes of $|0\rangle$ and $|1\rangle$. So, given a quantum state $|\phi\rangle = \alpha|0\rangle + \beta|1\rangle$, the application of the quantum NOT should return $|\phi'\rangle = \beta|0\rangle + \alpha|1\rangle$. The quantum NOT is represented by the following *unitary matrix*:

$$NOT = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

Intuitively, the first column of this 2×2 matrix says what happens to the input vector $|0\rangle$ and the second column says what happens to the input vector $|1\rangle$. Hence, the application to $\alpha|0\rangle + \beta|1\rangle$ gives:

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} \beta \\ \alpha \end{pmatrix}$$

i.e., $\beta|0\rangle + \alpha|1\rangle$.

[4] A unitary matrix is a square ($n \times n$) complex matrix U satisfying the condition $U^\dagger U = UU^\dagger = I_n$, such that $U^\dagger$ is the conjugate transpose (also called the Hermitian adjoint) of U.

Other example of an important unitary transformation acting in one qubit is the Hadamard quantum operation, which generates a quantum state in a *coherent superposition*:

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

Note the rule of the columns again: the first column says what happens to the input vector $|0\rangle$, that is, should return $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$, and the second column says what happens to the input vector $|1\rangle$, that is, should return $\frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle$. Hence, this operation generates a qubit in a *coherent superposition*. Here *coherent* means that the state has the same probability amplitude for $|0\rangle$ and for $|1\rangle$, i.e., $|\frac{1}{\sqrt{2}}|^2 = 1/2$.

Additionally, there are operations that change the *phase factor* of a qubit. In quantum mechanics a phase factor is a complex coefficient that multiplies a vector $|\psi\rangle$. In terms of classical observation of a qubit the phase factor has no physical meaning. However the phase factor is an important quantity when considering *interference*. An example is the phase-flip transformation:

$$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

An example of a quantum operation acting on two qubits is the controlled not, $CNOT$, operation. It is a quantum controlled operation for a state of two qubits (a 4×4 matrix). The first qubit is the controller and the $CNOT$ only applies the NOT on the second qubit if the first qubit is 1, otherwise it does nothing.

$$CNOT = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Again, it is interesting to note the rule of the columns, however for a two qubit state: the first column says what happens if the input vector is $|00\rangle$, the second column says what happens if the input vector is $|01\rangle$, the third says what happens if the input vector is $|10\rangle$ and the last column says what happens if the input vector is $|11\rangle$. Note that the input can be in a superposition, so the output is a combination of each case. Also note that there is an ordering for the coefficients in the rows (each row represents a basis vector), which is the same as explained in section above.

It is intriguing to note the *orthogonality condition*: due to the condition that all these operations must be unitary/reversible each column of the matrix must represent an orthogonal basic vector with respect to the other columns.

Unitary transformations acting in quantum vector states (or also called pure states) are called *pure quantum computations* as no information is lost.

2.2.2 Measurement Intuitively, unitary transformations represent a kind of *reversible rotation* on the vector which represents the state of the quantum system. A *measurement* causes other kind of transformation on the vector, called a collapse, and allows to get some classical information about the quantum state.

A measurement in a state like $\alpha|0\rangle + \beta|1\rangle$ will return the state $|0\rangle$ (or simply 0) with probability $|\alpha|^2$ and the state $|1\rangle$ (or simply 1) with probability $|\beta|^2$. Moreover, the quantum state is modified by the measurement: in this case, after the measurement the state will be $|0\rangle$ if the observed value was 0 and $|1\rangle$ if the observed value was 1. Immediately after the measurement we cannot recover the state before the measurement. This implies that we cannot collect any additionally information about α and β by repeating the measurement.

In this example, we explained the measurement in the *computational basis*, $|0\rangle$ and $|1\rangle$. However we can measure a quantum state considering any orthonormal basis for the qubit. For instance, the states $|+\rangle = 1/\sqrt{2}(|0\rangle+|1\rangle)$ and $|-\rangle = 1/\sqrt{2}(|0\rangle-|1\rangle)$ also form an orthonormal basis for the qubit. Then, we can rewrite the state in that basis: $\alpha'|+\rangle+\beta'|-\rangle$. So a measurement of a qubit considering that basis will produce $|+\rangle$ with probability $|\alpha'|^2$, and $|-\rangle$ with probability $|\beta'|^2$. As an example, consider we measure the state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ in the $|+\rangle$, $|-\rangle$ basis. We first express $|\psi\rangle$ in the $|+\rangle$, $|-\rangle$ basis:

$$\begin{aligned} |\psi\rangle &= \alpha|0\rangle + \beta|1\rangle \\ &= \alpha\tfrac{1}{\sqrt{2}}(|+\rangle + |-\rangle) + \beta\tfrac{1}{\sqrt{2}}(|+\rangle - |-\rangle) \\ &= \tfrac{1}{\sqrt{2}}((\alpha+\beta)|+\rangle + (|\alpha-\beta)|-\rangle). \end{aligned}$$

Then the probability of measuring $|+\rangle$ is $|\frac{1}{\sqrt{2}}(\alpha+\beta)|^2 = |\alpha+\beta|^2/2$, and the probability of measuring $|-\rangle$ is $|\alpha+\beta|^2/2$.

Moreover, we can measure a composite quantum system. For instance, if two qubits are in a state $|\psi\rangle$ and we measure them, then the probability that the first qubit is in a state i, and the second qubit is in a state j is $P(i,j) = \langle ij|\psi\rangle$, where $\langle|\rangle$ is the traditional inner product operation in vector spaces. The inner product returns a scalar quantity for a pair of vectors. Formally, it is given by the multiplication of a row vector by a column vector. After the measurement the state of two qubits is $|\psi'\rangle = |ij\rangle$. However, we can still measure only the first qubit of a two state quantum system. In that case , the outcome is the same as if we had measured both qubits. That is, the probability that the first qubit is in state i is $P(i) = \sum_j^{0,1}\langle ij|\psi\rangle$. The new state after the measurement then consists of those terms in the superposition that are consistent with the outcome of the measurement, but normalized. This partial measurement represents a *projection* of the vector onto the subspace spanned by the states $|ij\rangle$, such that $j \in \{0, 1\}$.

Entanglement can also be explained using measurement. Consider the following entangled state: $|\psi\rangle = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$. If we take a measurement in the first qubit, then the state of the other qubit is determined by the outcome of the measurement. For instance, with probability $\frac{1}{2}$ we get $|00\rangle$ as the outcome of the measurement, and in this case, we know that the state of the system after the measurement must be $|00\rangle$.

3 Monads in Programming Languages

Monads were first applied to Computer Science by Moggi [6] when he presented a *category-theoretic semantics of computations*. As an example of this semantic approach for computations, Moggi showed a general way to structure various *notions of computation* in the *computational λ-calculus*. The computational λ-calculus is a modification of the typed λ-calculus, which is correct for proving equivalence of programs independent from any specific *notion of computation*. By *notion of computation* (or also called *computational effect*) he means a quantitative description of the denotation of *programs*. Examples of notions of computations are: computations with side effects, where a program denotes a map from a store to a pair, value and modified store; computations with exceptions, where a program denotes either a value or an exception; partial computations, where a program denotes either a value or diverges; nondeterministic computations, where a program denotes a set of possible values.

Typed λ-calculus is the core language for functional programming languages. However, functional programming languages extend typed λ-calculus with many *notions of computation* that are indispensable for programming. Besides simple and pure functions, all realistic programming languages include some kind of computational effects, and there is a long debate about how to structure the semantics of these computational effects in the context of functional programming languages (see papers about pure and impure functional programming languages [8, 9]). In this context, the monadic approach by Moggi was elegantly used to structure computational effects in pure functional programming languages [10], like Haskell [11].

In this section we review the category-theoretic concept of a monad, explain why it is a useful tool to elegantly model notions of computations and review how the approach can be applied in the context of the functional programming language Haskell.

3.1 Monads in Category Theory

When considering the categorical semantics of computations one can consider different levels of abstraction. Here we consider the *denotational view*, where programs can be viewed as morphisms of a category, whose objects are types.

Considering a simplification of the denotational view, programs can be identified with total functions from values to values. However, that is a gross simplification as state in [12]. Indeed, a simple program may include notions of computational effects, like input/output, exceptions, side effects, etc. In this sense, a program can be understood as a *function form values to computations*. For instance, a program that receives a *value* and *prints* it in the standard output.

Hence, we take a category C as a model for functions and develop on top of that a general understanding of *type* for values and computations. As explained in [12], we use a unary operation T on the objects of C, which maps an object A, viewed as the set of values of type τ, to an object TA corresponding to the set of computations of type τ. Then a program from A to B, i.e., which takes as input a value of type A and after performing a certain computation will return a value of type B, can be identified with a morphism form A to TB in C. Finally, there is a minimum set of requirements on values and computations so that programs are the morphism of a suitable category.

These informal discussion on notions of computations and category theory leads to the use of Kleisli triples for modeling notions of computations and Kleisli categories for modeling categories of programs.

Definition 3.1 *A* ***Kleisli triple*** *over a category* C *is a triple* $(T, \eta, \_^*)$*, where* $T : Obj(\mathsf{C}) \to Obj(\mathsf{C})$ *(such that,* $Obj(\mathsf{C})$ *corresponds to the class of objects from the category* C*),* $\eta_A : A \to TA$*,* $f^* : TA \to TB$ *for* $f : A \to TB$*, and the following equations hold:*

- $\eta_A^* = id_{TA}$
- $\eta_A; f^* = f$
- $f^*; g^* = (f; g^*)^*$

Intuitively η_A is the *inclusion* of values into computations and f^* is the *extension* of a function f from values to computations to a function from computations to computations. The axioms amounts exactly to say that programs form a category, the Kleisli category C_T, where the set $\mathsf{C}_T(A, B)$ of morphisms from A to B is $\mathsf{C}(A, TB)$, the identity over A is η_A and the composition of $f : A \to TB$ followed by $g : B \to TC$ is $f; g^*$.

Monads are equivalent to Kleisli triples, which, as seen above, are easy to justify computationally. There is a one-to-one correspondence between Kleisli triples and monads. However, the formal definition of monads is given in terms of functors and natural transformations. We are not going to formally define monads here as the alternative definition of Kleisli triples is enough in the context of denotational semantics of computational effects in programming languages. Additional information on monads can be found in [13].

3.2 Adding Monadic Effects in Functional Programming

The pure λ-calculus is the core language for functional programming languages. It offers simple equational reasoning mechanism and referential transparency. Essentially, that means the equations and inference rules of pure λ-calculus say that “the order of evaluation of an expression is irrelevant”.

As types are very relevant for the discussion about adding monadic effects to the language we are going to present pure λ-calculus with simple types and with constants for numbers and addition. The following grammar generates the set of simple types over the type of natural numbers:

$$A \quad ::= \quad N \mid A \to A$$

i.e., we have the type of natural numbers and the type of functions. Consider the syntax of expressions:

$$e \quad ::= \quad x \mid \lambda x : A.e \mid e_1\ e_2 \mid n \mid e_1 + e_2$$

The syntax is given using traditional BNF (*Backus-Naur Form*) notation. The derivation rule says the expressions of the language are constituted by simple variables, λ-abstraction with an explicit type annotation on the bound variable telling us to assume hat the argument will be of an explicit type A, application, constants for numbers and addition, respectively.

The typing relation for pure λ-calculus, written $\Gamma \vdash e : A$ and read in the environment Γ, the expression e has type A, where Γ is a set of assumptions about the free variables in e, is defined by a set of inference rules assigning types to expressions, summarized in Figure 1.

Below we show the axioms for the equational semantics for the language.

$$\begin{array}{rcll}
(\lambda x : A.e)\ e' & = & e[e'/x] & (\beta) \\
\lambda x : A.e\ x & = & e & (\eta) \\
(e_1 + e_2) + e_3 & = & e_1 + (e_2 + e_3) & (S) \\
e_1 + e_2 & = & e_2 + e_1 & (C) \\
n_1 + n_2 & = & n \qquad \text{where } n = n_1 + n_2 & (A)
\end{array}$$

$$\frac{}{\Gamma \vdash n : N}\ \textit{Type Cons} \qquad \frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ \textit{Type Var} \qquad \frac{\Gamma, x : A_1 \vdash e : A_2}{\Gamma \vdash \lambda x : A_1.e : A_1 \to A_2}\ \textit{Type Abs}$$

$$\frac{\Gamma \vdash e_1 : A_{11} \to A_{12} \qquad \Gamma \vdash e_2 : A_{11}}{\Gamma \vdash e_1\ e_2 : A_{12}}\ \textit{Type App}$$

$$\frac{\Gamma \vdash e1 : N \qquad \Gamma \vdash e_2 : N}{\Gamma \vdash e_1 + e_2 : N}\ \textit{Type Sum}$$

Figure 1. Pure Simply Typed λ-Calculus

First two equations are usual β and η reduction. The notation $e[e'/x]$ represents substitution, saying that the final expression will be e with the free occurrences of x substituted by e'. The three following equations stand for associativity, commutativity and evaluation of addition, respectively.With the following inference rule:

$$\frac{e_1 = e_2}{C[e_1] = C[e_2]}$$

for any evaluation context C. The evaluation context, or simply context, was introduced to reduction semantics of programming languages by [14] and controls where evaluation occurs. A context is an expression or a sub-expression with a placeholder (represented by $\Box$) for where the next evaluation step can take place. By any context we mean that the order of evaluation of an (sub)expression is irrelevant.

For instance, consider the evaluation of the expression $(\lambda x.x + x)\ (4 + 2)$. Depending on the evaluation context we consider we can have different orders of evaluation. Considering $C = \Box$, the empty context, and $e_1 = (\lambda x.x + x)\ (4 + 2)$ we can have the evaluation order on left below, and considering $C = (\lambda x.x + x)\ \Box$ and $e_1 = (4 + 2)$ we can have the evaluation order on right below.

$$\begin{array}{ll} & (\lambda x.x + x)\ (4 + 2) \\ =^{(by\ \beta)} & (4 + 2) + (4 + 2) \\ = & 6 + (4 + 2) \\ = & 6 + 6 \\ = & 12 \end{array} \qquad \begin{array}{ll} & (\lambda x.x + x)\ (4 + 2) \\ =^{(by\ C)} & (\lambda x.x + x)\ (2 + 4) \\ = & (\lambda x.x + x)\ 6 \\ = & 6 + 6 \\ = & 12 \end{array}$$

The example also illustrate the referential transparency property, i.e, the expressions yield the same value each time they are invoked independent of order of evaluation.

Now consider we want to add to the syntax of the language an implicit global location initialized to 0, and the expressions get and inc:

$$\begin{array}{lcl} e & ::= & x \mid \lambda x : A.e \mid e_1\ e_2 \mid n \mid e_1 + e_2 \\ & & inc \mid get \end{array}$$

where get returns the current contents of the global location and inc returns the current contents of the global location and increments it as a side effects.

Then, lets evaluate the expression $(\lambda x.x + x)\ inc$:

$$\begin{array}{llcll} & (\lambda x.x + x)\ inc & \qquad & & (\lambda x.x + x)\ inc \\ =^{(by\ \beta)} & inc + inc & & =^{(by\ inc)} & (\lambda x.x + x)\ 0 \\ = & 0 + inc & & = & 0 + 0 \\ = & 0 + 1 & & = & 0 \\ = & 1 & & & \end{array}$$

Again we can consider the two different orders of evaluation above. However in this example, as we are considering access to a *global variable*, the **order of evaluation is very relevant**. Depending on the order we can get **different** results. This is because we are adding some notion of **computational effect** to the language. Indeed, inc is not a simple function which returns a value, the invocation of inc may return different results each time depending on the global state.

The state of the art in pure functional languages uses the concept of monads explained in section above to sequencing effects. Monads separate the pure language from the effects through the type system, and impose minimal constraints on the language used to sequence effects: must have a composition operator that is associative and a unit for the composition. Moreover, they provide a generic treatment of a large class of computational effects.

Considering the definition of a monad as a Kleisli triple in section above, we need to add the tripe $(T, \eta, \_^*)$ for a global location to the language. In this case, the functor $T\ A = (A \times S)^S$, where S is a set of states. That is, the functor maps a type A of values to computations that are functions from a state/store to a pair of a value together with the modified state/store: $S \rightarrow (A \times S)$. The natural transformation η is the map $a \mapsto (\lambda s : S.\langle a, s\rangle)$ and if $f : A \rightarrow TB$ and $c \in TA$, then $f^*(c) = \lambda s : S.(let\langle a, s'\rangle = c(s)\ in\ f(a)(s'))$. Note that f^* is a passing state function.

In the programming language community, the η is usually called a function $return ::$ $A \rightarrow TA$ and the composition, $\_^*$, is usually pronounced "bind", $\gg\!= :: TA \rightarrow (A \rightarrow TB) \rightarrow TB$, a function that receives two inputs, a computational effect, TA, and a function from $A \to TB$, and returns a computational effect, TB [5]. So we add these two constructions to the language:

$$\begin{array}{rcl} e & ::= & x \mid \lambda x : A.e \mid e_1\ e_2 \mid n \mid e_1 + e_2 \\ & & return\ e \mid e_1 \ggg\!= e_2 \mid inc \mid get \end{array}$$

With the following typing rules:

$$\frac{}{\Gamma \vdash inc : TA}\ \textit{Type Inc} \qquad \frac{}{\Gamma \vdash get : TA}\ \textit{Type Get}$$

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash return\ e : TA}\ \textit{Type Return} \qquad \frac{\Gamma \vdash e_1 : TA \quad \Gamma \vdash e_2 : a \to TB}{\Gamma \vdash e_1 \ggg\!= e_2 : TB}\ \textit{Type Bind}$$

Figure 2. Types for the Monadic Functions

Hence, from now is illegal to write $(\lambda x.x + x)\ inc$ because monads *separate* the pure language from the effects through the type system. One must use the monadic functions for sequencing the effects and write one of the following two programs depending on the intended meaning:

$$\begin{array}{ll} inc \ggg\!= \lambda x_1. & \qquad inc \ggg\!= \lambda x_1. \\ inc \ggg\!= \lambda x_2. & \qquad return(x_1 + x_1) \\ return(x_1 + x_2) & \end{array}$$

In this way, the value of the global location is propagated and maintained by the monad. Using the axioms below one can see that in the left expression, $x_1 + x_2$ reduces to 1, and that in the right expression, $x_1 + x_1$ reduces to 0.

Regarding the axioms and inference rules, the monadic approach states that we should maintain all the axioms for the pure language, add the generic axioms for monads (the monads laws reflecting the monadic equations on Definition 3.1), and also add the specific axioms to the particular effect in question, in this case get and inc.

The generic axioms in the definition of the Kleisli triple can be rewritten in terms of the monadic functions, $return$ and $\ggg\!=$, as:

1. Left identity: $return\ a \ggg\!= f = f\ a$
2. Right identity: $m \ggg\!= return = m$

[5]Note that the of the arguments of $\ggg\!=$ and $\_^*$ are the same with a small changing in the order.

3. Associativity: $(m \gg= f) \gg= g = m \gg= (\lambda x \rightarrow f\ x \gg= g)$

The axioms for get and inc are given below:

$$\begin{array}{lcll} \langle inc \gg= \lambda x.e, n\rangle & = & \langle e[n/x], n+1\rangle & (inc) \\ \langle get \gg= \lambda x.e, n\rangle & = & \langle e[n/x], n\rangle & (get) \end{array}$$

Note that we are explicitly showing what happens to the implicit global location (gl) by means of the notation $\langle e, gl\rangle$. To illustrate the reduction steps we can evaluate the expression with the explicit global location initialized to 0:

$$\begin{array}{ll} & \langle inc\ \gg= \lambda x_1. \\ & return(x_1 + x_1), 0\rangle \\ =^{(by\ inc)} & \langle return(0+0), 1\rangle \\ =^{(by\ A)} & \langle return(0), 1\rangle \end{array}$$

3.3 Monads in Haskell

Haskell [11] is a pure functional programming language. Independently of being pure [6], Haskell has pure λ-calculus as its core language, which offers simple equational reasoning mechanism and referential transparency. As a general purpose language, Haskell offers computational effects via built-in monads for effects like input/output (the IO monad), lazy lists (the $List$ monad), and failure (the $Maybe$ monad). The approach used in the language design is the same as explained in section above.

Moreover, in Haskell, the user can also declare a new type to be a monad. That is, any computational effect of interest can be added to the language and treated as a monad. To do so, the user needs to define:

1. A Type constructor M: $data\ M$
2. Definition of a function return: $return :: a \rightarrow M\ a$
3. Definition of a function bind: $\gg= :: M\ a \rightarrow (a \rightarrow M\ b) \rightarrow M\ b$

More precisely, one can consider the type constructor as a model for functions with effects. The trivial mapping of values to computations is given in the definition of the function $return$. The function $\gg=$ must reflect how to sequence/transform effects. Note that a

[6] We are not going to discuss about pure and impure functional languages in this work, for a nice discussion see [8].

transformation of a computation $M\ a$ in a computation $M\ b$ must be defined in terms of a function of type $a \to M\ b$.

Of course, to construct a correct *monad*, the $return$ and $\gg\!=$ functions must work together according to the three *monadic axioms* presented in section above.

Additionally, Haskell has a type class mechanism, which is a type system construct that supports ad-hoc polymorphism. There is a standard Monad class that defines the names and signatures of the two monad functions $return$ and $\gg\!=$. It is not strictly necessary to make any user defined monad an instance of the Monad class, but it is a good idea if one wants to overload the name of the functions and also use Haskell' syntactic sugar for sequence monadic functions called *do*-notation.

3.4 Quantum Monad

Quantum bits can be thought as a kind of computation effect. More specifically, quantum bits can be modeled as type of a monad for non-determinism as first suggested by Mu and Bird [15].

Non-deterministic programs can not be implemented by pure functions as they may not produce the same value for the same input. However, non-deterministic programs can be simulated by functions returning the *set* of all possible solutions. A Kleisli triple $(T, \eta, \_^*)$ that gives a category of non-deterministic programs is defined as follows: $TA = \mathsf{P}(A)$, where $\mathsf{P}(A)$ is the power set over A. The function η_A is the singleton map $a \mapsto \{a\}$, and if $f : A \to TB$ and $c \in TA$, then $f^*(c) = \cup_{x \in c} f(x)$.

Following this idea of using a monad for non-determinism, the second author of the present paper modeled, in [5], pure quantum vector states and unitary transformations using a Kleisli triple $(T, \eta, \_^*)$ that gives a category of *pure quantum programs* defined as follows: $TA = \mathsf{P}(A \times \mathbb{C})$, where $\mathsf{P}(A \times \mathbb{C})$ is the power set over $A \times \mathbb{C}$. The function η_A is the singleton map $a \mapsto \{(a, 1)\}$, and if $f : A \to TB$ and $q \in TA$, then $f^*(q) = \cup_{(a,c) \in q}\ let\ (a_2, c_2) = f(a)\ in\ (a_2, c * c_2)$.

The idea behind this monad is that this kind of non-deterministic monad builds the quantum state space spanned by the basis set A, and $\_^*$ has the meaning of linearity. As we saw in Section 2.1 a quantum bit is a unit vector in a complex bi-dimensional vector space. Considering the computational basis $A = \{1, 0\}$, we can write a qubit as:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle.$$

where α and β are complex numbers given the respective probability amplitudes for the basic states, $|0\rangle$ and $|1\rangle$. So, the functor T builds the quantum state space for the qubit. For instance, for $A = \{1, 0\}$ then $\mathsf{P}(A \times \mathbb{C}) = \{\{(0, \alpha)\}, \{(1, \beta)\}, \{(0, \alpha), (1, \beta)\}\}$ such that $\alpha, \beta \in \mathbb{C}$.

In this case, η builds the basis vectors $|0\rangle$ by $0 \mapsto \{(0,1)\}$ and $|1\rangle$ by $0 \mapsto \{(1,1)\}$. The extension $\_^*$ denotes the application of a linear function to an input vector.

Below we summarize the work presented in [5], as a quantum monad implementation using the Haskell functional programming language. The Java quantum library discussed in next section is based in this presentation.

$$\begin{array}{l} \textit{type}\ Vec\ a = a \rightarrow \mathbf{C} \\ return\ a_1\ a_2 = \textit{if}\ (a_1 \equiv a_2)\ \textit{then}\ 1\ \textit{else}\ 0 \\ v_a \gg\!= f = \lambda b \rightarrow sum\ [v_a\ a * f\ a\ b] \end{array}$$

The type constructor of the quantum monad named Vec, defines that a underlying type a is mapped to a *complex number*. The *return* function builds trivial quantum vectors, i.e, just the basis states and the *bind* specifies how to sequence computations.

Furthermore, vector spaces have additional properties abstracted in the monad with the plus operation. This supports two additional operations called $mzero$ and $mplus$ which provide a “zero” computation and an operation to “add” computations:

$$\begin{array}{l} mzero :: Vec\ a \\ mzero = \textit{const}\ 0 \\ mplus :: Vec\ a \rightarrow Vec\ a \rightarrow Vec\ a \\ mplus\ v_1\ v_2\ a = v_1\ a + v_2\ a \end{array}$$

For convenience, it is also possible to define various kinds of products over vectors: the *scalar* product is an example:

$$\begin{array}{l} (\$*) :: K \rightarrow Vec\ a \rightarrow Vec\ a \\ pa\ \$*\ \ v = \lambda\ a \rightarrow pa * v\ a \end{array}$$

For instance, consider the basis vectors $|0\rangle$ and $|1\rangle$ can be programmed as over the basis of Booleans as:

$$\begin{array}{l} qFalse, qTrue :: Vec\ Bool \\ qFalse = return\ False \\ qTrue = return\ True \end{array}$$

Superpositions can be generated using the $mplus$ and scalar product, for instance:

$$\begin{array}{l} qFT, qFmT :: VecBool \\ qFT = (1/\sqrt{2}\ \$*\ (qFalse\ `mplus`\ qTrue) \\ qFmT = (1/\sqrt{2}\ \$*\ qFalse)\ `mplus`\ (-1/\sqrt{2}\ \$*\ \ qTrue) \end{array}$$

Given two base sets A and B a linear operator $f \in A \multimap B$ is a function $f : A \to B \to \mathbb{C}$. Such operators can be represented as functions mapping values to vectors: $Lin\ a\ b = a \to Vec\ b$ For example, the quantum version of the boolean negation is:

$$\begin{array}{l} qnot :: Lin\ Bool\ Bool \\ qnot\ a = return(/a) \end{array}$$

and the hadamard operation can be programmed as:

$$\begin{array}{l} hadamard :: Lin\ Bool\ Bool \\ hadamard\ False = qFT \\ hadamard\ True = qFmT \end{array}$$

4 Quantum Monad using Java Closures

Java is a general propose objected oriented programming language that enables secure and high performance software development on multiple platforms. It is one of the most widely used programming language in software development. Hence, based on the monadic model for quantum programming discussed in section above we present in this section a monadic Java library for quantum programming. We use the extension of Java called BGGA Closure, that allow the manipulation of anonymous functions (closures) inside Java. We exemplify the use of the library with an implementation of the Toffoli quantum circuit.

4.1 Anonymous Functions and Closures in Java

Anonymous functions are strongly related to λ-calculus and are (like λ-functions) functions without a *name*. A *closure* is a kind of anonymous functions, it is a function with an *environment* containing the bindings of its free variables. The word *closure* comes from the term *closed expression* and refers to a *lambda* expression with its free variables closed by a lexical environment [16]. Indeed, when we have a function that is nested inside another, if it accesses a variable from it's parent's scope, we create a closure.

These concepts of anonymous functions and their closures are very common in the functional programming community, where functions are first class citizens, allowing higher order programming. The use of anonymous functions in an imperative set provides a way to express abstractions that are currently quite awkward in non-functional languages. One can use functions as arguments to other functions (or methods), or define functions dynamically.

Therefore, with the introduction of these concepts in the programming language Java now we can create a whole new kind of data structures. There are two basic syntax for anonymous function in BGGA:

```
(parameters) -> expression}
(parameters) -> { declarations;}
```

The declaration of anonymous function also allows omission of modifiers, return type, and, in some cases, the types of parameters. For instance:

```
1.  (int x, int y) -> x + y
2.  (x, y) -> x - y
3.  () -> 42
4.  (String s) -> System.out.println(s)
5.  x -> 2 * x
6.  c -> { int s = c.size(); c.clear(); return s; }
```

The first function takes two integers and returns their sum. The second one receives two values and returns their difference. The third receives no value and returns the number 42. Fourth receives a string, prints its value in the console, and returns no value. The fifth receives a number and returns its double. The sixth receives a collection, cleans it and returns its previous size.

Note that type parameters may be stated explicitly (ex: 1/4) or implicitly inferred (ex 2/5/6) but not mixed in a single lambda expression. The function body can be a block (surrounded by braces, ex 6) or an expression (ex 1-5). A function can have a return value or nothing (void). If the body is just an expression, it can return a value (ex 1/2/3/5) or nothing (ex 4) without explicitly declare the return statement (return). Parentheses may be omitted for a single inferred type parameter (ex 5/6).

In Java, to assign a dynamic function to a variable and then use it to call the method created or just to pass as an argument to another function, the use of *functional interfaces*[7] is required. Such functional interfaces can be seen as any interface that has exactly one abstract method declared.

Some functional interfaces are already distributed inside the Java platform. Among many, here are some examples:

```
public interface Runnable {
  void run(); }

public interface Callable<V> {
  V call() throws Exception; }

public interface ActionListener {
  void actionPerformed(ActionEvent e); }
```

As an example, consider a definition of an anonymous function representing a dynamic function that prints any sentence on the console screen. Then, lets assign it to a local

[7]This is also a new concept included in Java 8

variable. In this specific case we can use the already defined functional interface *Runnable*. Doing so, means that the invocation of the dynamic function must be done trhough the calling of the method *run*, the only abstract method of the functional interface. The code below, prints the sentence ”Hello World” in the console.

```
Runnable c = () -> { System.out.println(”Hello World”); };
c.run();
```

4.2 The Library QJava

In this section we present our library called QJava: Quantum monad in Java. We use Java Development Kit (JDK version 8), which has support to anonymous functions.

Considering the original definition of the quantum monad presented in Section 3.4, we can build the following relation between the monad definition and *object-oriented* programming: the type constructor corresponding to the declaration of the monadic type is a class declaration; the *return* function takes the role of the *constructor* method; and the *binding* operation contains the logic necessary to execute its registered *callbacks*.

Then, in order to implement a monad as a closure in Java we need to define a functional interface:

```
public interface V<A> {
  Complex invoke(A bool);
}
```

In this way, we can use a closure to represent an quantum state as a function that maps a basis type *A* to a complex number, i.e., to its probability amplitude. The generic type `A` provides a generic implementation, making it possible to create a quantum state of `N` qubits. For example, a quantum state with two qubits would have the type tuple<boolean,boolean>, a quantum state with three qubits would have the type tuple<boolean,boolean,boolean>, and so on.

Hence, quantum vectors are represented by the following class `Vec`:

```
public class Vec <A> {
  V<A> vec;
  A type;

  Vec (A type, V<A> vec){
    this.type = type;
    this.vec = vec;
  }

  Complex invoke(A bool) {
    return vec.invoke(bool);
    }
  }
```

Additionally, the class `Vec` has `mzero`, `mplus` and scalar product:

```
public static <A> Vec<A> mzero(A tipo) {
  return new Vec<A>(tipo, (A bool) -> Complex.ZERO);
}

public static <A> Vec<A> mplus(Vec<A> va, Vec<A> vb) {
  return new Vec<A>(va, (A bool) ->
        va.invoke(bool).plus(vb.invoke(bool)));
}

public static <A> Vec<A> scalar(Complex c, Vec<A> va) {
  return new Vec<A>(va, (A bool) ->
       va.invoke(bool).times(c));
}
```

The monadic functions, *return* and *bind*, are implemented in a class called `QMonad`. The *return* function is a method, `qreturn`, that receives a value of type `A` and construct a basic vector state over the basis `A`, as in the return function explained in section 3.4.

```
public static <A> Vec<A> qreturn(A base) {
  return new Vec<A>(base, (A bool) -> {
    return bool.equals(base)?Complex.ONE:Complex.ZERO;
  });
}
```

The returned type is a functional object representing a quantum state vector. More specifically, the method receives a type `A` and returns a closure, i.e., an instance of the class `Vec<A>`. The closure maps all values of the type `A` to zero except the value passed as its argument. Thus, we can create the basis quantum states $|0\rangle$ e $|1\rangle$, as:

```
Vec<Boolean> vzero = QMonad.qreturn(false);
Vec<Boolean> vone  = QMonad.qreturn(true);
```

Linear operators mapping quantum state vectors to new quantum state vectors, are implemented in Java exactly as described in Section 3.4, i.e, as a function mapping values to vectors. Then, we define a class `Lin<A,B>` with the following functional interface:

```
public interface L<A,B> {
  Vec<B> invoke(A bool);
}
```

This closure takes a value of type `A` and returns a `Vec<B>`.

Now we are ready to explain the implementation of the monadic *bind* function, which models the application of a linear operation to a quantum state vector, i.e, how to *sequence* quantum operations. Then, it is implemented as a method that receives a vector `va`, of type `Vec<A>`, and a linear operation `f`, of type `Lin<A,B>`. The method returns a new vector of type `Vec<B>` representing the application of the transformation `f` in `va`:

```
public static <A,B> Vec<B> bind(Vec<A> va, Lin<A,B> f) {
  return new Vec<B>( f.invoke(va.getBase()), (B b) -> {
    Complex soma = Complex.ZERO;
    for (A a : va.getBases())
      soma = soma.plus(va.invoke(a).times(f.invoke(a).invoke(b)));
    return soma;
  });
}
```

Basically, the behaviour of the method `bind` must capture a matrix vector multiplication. To do so, we need to know all possible values of the basis type `A`, which is done by the method `getBases()`. The methods `plus` and `times` are the sum and multiplication of complex numbers. Therefore, the `bind` method receives a vector state of type `Vec<A>`, a linear operation of type `Lin<A,B>` and binds the computation to a new vector `Vec<B>`.

To help the user of the library we define a universal set of quantum operations provided within the class `Lin<A,B>`. For instance, there are included the static methods `qnot()`, `qphase()` and `qhadamard()`. Furthermore, there is a general function `controlled()`, which returns the controlled operation of a given linear transformation. We can create the operation `cnot` for example, with the function call `controlled(qnot())`.

As a simple example consider the application of the transformation `qnot` in the quantum state $|0\rangle$:

```
Vec<Boolean> vzero  = QMoand.qreturn(false);
Vec<Boolean> result = QMonad.bind(vzero, qnot());
```

In this case, the quantum state *vresult* will be the vector that represent the application of the operation `qnot` in *vzero*. Since *vzero* represent the vector $|0\rangle$, the result of this operation will be quantum state $|1\rangle$.

4.2.1 A Toffoli Gate with QJava As an example of how to use the library QJava we present in this section an implementation of the quantum algorithm for the *Toffoli* circuit.

The *Toffoli* gate is a universal logic gate, also known as the "controlled-controlled-not" gate. It works over a three qubits quantum state as shown in Figure 3.

The gates labeled *H*, *V*, *VT* and *NOT* represent respectively the quantum operations *Hadamard*, *Phase*, *Adjoint Phase* and *NOT*. The bullets connecting another wire represent controlled operations.

This quantum circuit can be implemented in the library QJava as the following method:

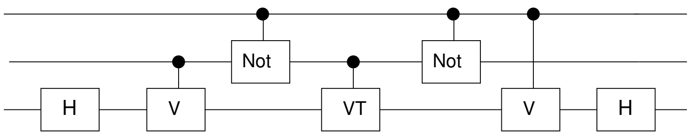

Figure 3. Circuit for the Toffoli gate.

```
public static Lin<Tuple, Tuple> Toffoli() {
 //Auxliary type constructors
 Boolean bool = Boolean.TRUE;
 Tuple bool_bool = Tuple.createTuple(Boolean.TRUE, Boolean.TRUE);
 Tuple bool_bool_bool = Tuple.createTuple(
  Boolean.TRUE, Boolean.TRUE, Boolean.TRUE);

 //Instance the required quantum gates
 Lin<Tuple, Tuple> cnot = Lin.controlled(Lin.qnot());
 Lin<Tuple, Tuple> cphase = Lin.controlled(Lin.qphase());
 Lin<Tuple, Tuple> caphase = Lin.controlled(Lin.adjoint(Lin.qphase()));
 Lin<Boolean, Boolean> hadamard = Lin.qhadamard();

 return new Lin<Tuple, Tuple> (bool_bool_bool, (Tuple t0m0b0) -> {
  return QMonad.bind(hadamard.invoke(b0),
    new Lin<Boolean, Tuple>(bool, (Boolean b1) -> {
      return QMonad.bind(cphase.invoke(mo,b1),
        new Lin<Tuple,Tuple> (bool_bool, (Tuple m1b2) -> {
          return QMonad.bind(cnot.invoke(t0,m1),
            new Lin<Tuple, Tuple>(bool_bool, (Tuple t1m2) -> {
              return QMonad.bind(caphase.invoke(m2,b2),
                new Lin<Tuple, Tuple>(bool_bool, (Tuple m3b3) -> {
                  return QMonad.bind(cnot.invoke(t1,m3),
                    new Lin<Tuple, Tuple>(bool_bool, (Tuple t2m4) -> {
                      return QMonad.bind(cphase.invoke(t2,b3),
                       new Lin<Tuple, Tuple>(bool_bool, (Tuple t3b4) -> {
                        return QMonad.bind(hadamard.invoke(b4),
                         new Lin<Boolean, Tuple>(bool, (Boolean b5) -> {
                          return QMonad.qreturn(Tuple.createTuple(t3, m4,b5));
                         }));
                       }));
                    }));
                }));
            }))
        }));
    }));
 });
}
```

We use the names top, middle and bottom to refer the qubits in the quantum circuit. The input state is written as `t0m0b0`, i.e., a tuple of <boolean, boolean, boolean>. We implement a class `Tuple` to deal with tuples in Java with default constructor to create tuples and projections. To simplify the reading of the code above we omit the call of the methods `createTuple` and the projection `getTthValue`. One can observe in the code how to use the monadic bind to program quantum algorithms with many qubits and how to apply specific operations to specific qubits. The advantage of using the monad is that it allows the programmer to work in parts of the global quantum state. For instance, the code starts applying the `hadamard` gate only to the third qubit of the system through `hadamard.invoke(b0)`. The new value for the bottom qubit is now called `b1` and then is passed around through the bind to be used in the operation `cphase.invoke(m0,b1)`. The computation follows with this same reasoning until the last `hadamard` application and then returns.

Finally, this method representing the *Toffoli* circuit can be applied to a three qubit state as $|111\rangle$:

```
Vec<Tuple> test = QMonad.qreturn( Tuple.createTuple(true,true,true));
test = QMonad.bind( test, Toffoli());
```

5 Conclusion

In this paper we have presented a monadic library for quantum programming in Java using closures called QJava: Quantum monad in Java. The library can be used to program pure quantum algorithms in Java. We believe that one of the main advantages of the library is that it allows to elegantly express quantum algorithms in Java giving to the programmer the ability to work in parts of the global quantum state.

Besides this technical contribution the present paper has a detailed explanation of monads in the context of functional programming languages and its application to quantum computing. Monads and its variants are a very important concept in the semantic of programming languages. We hope this work can help readers also interested in monads in general.

As future work we intend to implement a parser to use the a syntax like the Haskell **do**-notation for monads. The investigation and implementation of a type system and then the definition of a *domain specific language* (DSL) for quantum programming inside Java is also a future work.

Due to well know problems of performance on simulation of quantum algorithms in classical computers, an implementation of the library using threads and General-Purpose Computation on Graphics Hardware (GPGPU) is also an idea of future work.

References

[1] R. Feynman and P. W. Shor, "Simulating physics with computers," *SIAM Journal on Computing*, vol. 26, pp. 1484–1509, 1982.

[2] D. Deutsch, "Quantum theory, the church-turing principle, and the universal quantum computer," in *Proceedings of the Royal Society of London*, vol. 47, pp. 97–117, 1985.

[3] P. W. Shor, "Algorithms for quantum computation: discrete logarithms and factoring," in *In Proc. IEEE Symposium on Foundations of Computer Science*, pp. 124–134, 1994.

[4] C. Pomerance, "A tale of two sieves," *Notices Amer. Math. Soc.*, vol. 43, pp. 1473–1485, 1996.

[5] J. K. Vizzotto, T. Altenkirch, and A. Sabry, "Structuring quantum effects: Superoperators as arrows," *Journal of Mathematical Structures in Computer Science: special issue in quantum programming languages*, vol. 16, pp. 453–468, 2006.

[6] E. Moggi, "Computational lambda-calculus and monads," in *Proceedings of the Fourth Annual Symposium on Logic in computer science*, pp. 14–23, IEEE Press, 1989.

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

[8] A. Sabry, "What is a purely functional language?," *Journal of Functional Programming*, vol. 8, pp. 1–22, 1998.

[9] P. Wadler, "Monads for functional programming," in *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, (London, UK, UK), pp. 24–52, Springer-Verlag, 1995.

[10] P. Wadler, "The essence of functional programming," in *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '92, (New York, NY, USA), pp. 1–14, ACM, 1992.

[11] P. Hudak, S. Peyton Jones, and P. Wadler (editors), "Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2)," *ACM SIGPLAN Notices*, vol. 27, May 1992.

[12] E. Moggi, "An Abstract View of Programming Languages," tech. rep., Edinburgh University, 1989.

[13] M. Barr and C. Wells, *Toposes, Triples and Theories.* Springer Verlag, 1895.

[14] M. Felleisen, M. Felleisen, R. Hieb, and R. Hieb, “The revised report on the syntactic theories of sequential control and state,” *Theoretical Computer Science*, vol. 103, pp. 235–271, 1992.

[15] S.-C. Mu and R. S. Bird, “Functional quantum programming,” in *APLAS*, pp. 75–88, 2001.

[16] J. P. Landin, “The mechanical evaluation of expressions,” in *The Computer Journal*, 1964.