FJQuantum
A Quantum Object Oriented Language

Samuel S. Feitosa1, Juliana K. Vizzotto2, Eduardo K. Piveta3
Department of Languages and Computer Systems
Universidade Federal de Santa Maria
Santa Maria - RS, Brazil

Andre R. Du Bois4
Department of Computing - CDTec
Universidade Federal de Pelotas
Pelotas - RS, Brazil

Abstract
Several languages and libraries have been proposed to work with quantum programs, usually considering the imperative and functional paradigms. In this paper, we discuss the application of the FJQuantum language, an object-oriented language based on Featherweight Java to develop programs that handle quantum data and operations.

Keywords: Programming Languages, Quantum Computing.

1 Introduction

Quantum computing is a research field that investigates all aspects of computation considering the quantum nature of the physical world. Unlike conventional computers, a quantum computer presents some characteristics like superposition and entanglement, which enable quantum computers to consider and manipulate combinations of bits simultaneously, enabling a faster quantum information processing when compared with conventional computation [21]. Although there are no quantum computers for general purpose, there are several works [15, 29, 13, 14, 27, 18, 7] on different approaches to process quantum information.

1 Email: sfeitosa@inf.ufsm.br
2 Email: juvizzotto@inf.ufsm.br
3 Email: piveta@inf.ufsm.br
4 Email: dubois@inf.ufpel.edu.br

http://dx.doi.org/10.1016/j.entcs.2016.09.007
1571-0661/© 2016 The Author(s). Published by Elsevier B.V.
This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).
One challenging research area in quantum computing is the design of high-level quantum programming languages [17,22,23,6,25,3,28,32] suitable for describing and reasoning about quantum algorithms, and also providing tools to understand how quantum computing works in general. In this context, this work discusses the implementation of the \textit{FJQuantum} language, showing a set of source-code examples for handling quantum computing concepts, taking advantage of the object-oriented paradigm.

The \textit{FJQuantum} language is an extension of Featherweight Java (\textit{FJ}) [16], which is a small calculus, providing a formal semantics for the main aspects of the Java language. Thus, \textit{FJQuantum} provides all the features of FJ, adding several characteristics to simulate quantum behavior through a monadic approach. The \textit{FJ} was chosen as basis for this project because of Java’s acceptance, the simplicity of its semantics, and also because it provides an operational semantics, which facilitates the formal study of extensions to this language.

This work follows others proposals already developed using monads for quantum computing [30,32], and also the \textit{QJava} library [8], which provides mechanisms to simulate quantum computing in Java using \textit{closures}. Here we explore the capacity to write code in the \textit{FJQuantum} language, showing how to model some quantum computing aspects through its new constructions.

2 Quantum Computing

The basic information unit in classical computing is the traditional \textit{bit}, which represents the classical binary physical system, being able to represent only two states (\texttt{true} or \texttt{false}, 0 or 1). Every information is described as a combination of \textit{bits}.

In quantum computing, the basic information unit is called a \textit{quantum bit} or \textit{qubit}. The \textit{qubit} presents an essential difference if compared to the \textit{bit}, because it is not confined to the basic states of the classical \textit{bit}, it can be effectively in both states (0 and 1) at the same time [21]. Several researchers have studied ways to handle particles which are capable of providing the quantum characteristics. However, there are still many challenges in the physical manipulation of elements in microscopic scale.

The theory of quantum computing mathematically defines a \textit{qubit} as a vector \textsuperscript{5}

\[ |\psi\rangle = \alpha |0\rangle + \beta |1\rangle \]

where each of its positions stores the \textit{probability amplitude} \(\alpha\) and \(\beta\), representing each of its basic states. The probability amplitudes are represented as \textit{complex numbers}, such that \(|\alpha|^2 + |\beta|^2 = 1\).

Intuitively, one can imagine a \textit{qubit} as 0, 1 or both states occurring simultaneously, having a numeric coefficient which determines the probability of each pure state.

Any other state with different values for \(\alpha\) and \(\beta\) represents a \textit{quantum superposition} of \(|0\rangle\) and \(|1\rangle\). These \textit{superposition} states provide to quantum computing

\textsuperscript{5} Na notao de Dirac.
a characteristic called quantum parallelism. Essentially, because of state superposition, a qubit can assume the 0 and 1 values at the same time. This property is explored on quantum algorithms, which can obtain an exponential speedup (on theory), considering the characteristic that allow to handle several possibilities in parallel.

The classical bit can be examined to determine its current value (0 or 1), and this is what happens at every moment in classical computers, when handling the memory contents. In the case of qubits, it is impossible to visualize their values to determine their current states (amplitudes $\alpha$ and $\beta$) without interfering in the system. Reading the quantum state performs a measurement operation.

On quantum computing, one can perform two forms of operations: measurement operations and unitary transformations. The measurement operation is related to a way of extracting information from the quantum state. Unitary transformations refer to operations that transform the current quantum state into another, similarly to when we apply a function in classical computing.

Quantum computing is different from classical computing because it is usually probabilistic, so, the measurement operation works over the probability amplitudes of a quantum state. When a measurement is performed, the probability amplitudes collapse and just one of their basic states is returned, like $|0\rangle$ or $|1\rangle$. In other words, after a measurement, the qubit stays on a known state and the probability amplitudes are destroyed. The measurement operation usually is performed to obtain information after the entire processing of a quantum algorithm.

Similarly to the way of processing information in classical systems, on quantum computing an algorithm is designed as a series of unitary transformations, also known as quantum gates [21]. These gates, applied to qubits, modify their initial value transforming them into the desired output.

3 The FJQuantum Language

FJQuantum is an object-oriented language developed as an extension of Featherweight Java (FJ) [16], adding several constructions to allow the development of programs with features to handle quantum data and operations through a monadic layer. This language aims to formalize a previous work [8] considering the use of quantum monads in Java language.

FJ introduces a lightweight version of Java, providing a formalization for its core parts, offering the language’s main operations, providing mechanisms to represent classes, methods, attributes, inheritance and dynamic casts with similar semantics of the original one [16], described through an operational semantics. Besides providing Java’s main characteristics, it focuses on a functional view of that language, without side effects and several constructions, for example: it is not allowed to use assignments, interfaces, overload, null pointers, primitive types, static methods, etc.

Therefore, an FJ program has as entry point a term, arranged after the class definitions (as showed through examples in the next section) and, in its original version it allows the use of only five different terms: object creation, method invocation,
attribute access, casts and variables.

Considering that the simulation of quantum computing involves mathematical operations over complex numbers, and also requires a series of control mechanisms, we added several extensions to FJ, with the primary purpose of enabling a quantum monadic layer. Among these extensions are:

- Features to handle basic types (booleans and complex numbers).
- Mathematical operations over complex numbers.
- Conditional control structures.
- Functional tuples as primitive types.

The use of monads in quantum computing has been explored in several works [20,30,33,32], usually applied to functional languages. A different approach was used in Calegaro and Vizzotto [8], where concepts of monads were applied in Java through the use of closures. Closures enable the use of anonymous functions (or lambda expressions) and we also added them as an extension to FJ, adapting the proposal of Bellia and Occhiuto [5], which is slightly different from the Java implementation. This approach was used for simplicity.

The modeling of quantum bits can be thought of as a type of side effect, since their non-deterministic nature. More specifically, qubits can be modeled as a type of monad [20]. The idea behind this monad is to build the space of quantum states, mathematically represented by a vector of complex numbers holding the probability amplitudes of qubits, enabling the transformation of states through quantum gates, which are represented as an unitary matrix and can be applied through the bind monadic operator [12].

For the FJQuantum language be able to handle quantum computing concepts, we proposed several syntactical constructions, each one with a specific purpose, such as: the feature to create quantum states, to handle probability amplitudes through the scalar product operator, to handle superpositions using the monadic sum operator, as well the possibility to create functions responsible to transforming the quantum state, through the bind monadic operator.

To allow the creation of quantum states, there is the monadic operator mreturn, which is used as a constructor that acts over the basis states. The basis states can be built from booleans or tuple of booleans, as we can see in the following example:

```
1 mreturn false // Create the state |0>
2 mreturn true  // Create the state |1>
3 mreturn {false,false} // Create the state |00>
4 mreturn {false,true}  // Create the state |01>
```

As a way to enable the manipulation of probability amplitudes and the creation of states in superposition, we created the scalar product $*$ and the monadic sum mplus. The following piece of code shows the construction of a state in superposition.

```
1 ComplexHalf $* mreturn false mplus ComplexMHalf $* mreturn true
```

In the example above, the keyword ComplexHalf represents the complex number $\frac{1}{\sqrt{2}}$ and the keyword ComplexMHalf represents the number $-\frac{1}{\sqrt{2}}$. The presented
state in the code above defines the quantum state in superposition $\frac{|0\rangle - |1\rangle}{\sqrt{2}}$. Next section examples show the use of these operators in the hadamard method.

After preparing the language with the necessary tools to create quantum states and handle its probability amplitudes, it is possible to define the bind operator, syntactically represented in FJQuantum as >>=, which is responsible for applying transformations over the quantum state. The quantum transformations can be seen as quantum gates applications on qubits, similarly to information processing in classical computing. The next example shows how to use this operator.

```java
1 (mreturn false) >>=
2 qop.hadamard() >>=
3 qop.not() >>=
4 qop.hadamard();
```

In this piece of code, we can see the application of hadamard on the state $|0\rangle$, after that applying the not operator on the result of the first processing, and then applying again the hadamard, after the processing of not.

In addition to the definition of FJQuantum, we developed an interpreter aiming to test the rules and to write quantum algorithms. The interpreter was developed in Haskell, and it implements the lexer, the parser, the semantics and the type system.

## 4 Examples

This section presents some examples of FJQuantum programs.

First, we show a class that implements a series of universal reversible quantum gates. Line (2) shows the not method representing the quantum version of the classical operator, which is applied over one qubit. Line (11) shows the hadamard method, which represents an operator responsible for transforming a qubit from a basic state into a state superposition. Line (21) shows the controlledNot method representing a conditional not.

```java
class QOp extends Object {
1    (boolean -> Vec<boolean>) not() {
2        return (boolean i) ->
3            if (i == false) {
4                mreturn true
5            } else {
6                mreturn false
7            };
8    }
9
10    (boolean -> Vec<boolean>) hadamard() {
11        return (boolean b) ->
12            if (b == false) {
13                (ComplexHalf $* mreturn false) mplus
14                (ComplexHalf $* mreturn true)
15            } else {
16                (ComplexHalf $* mreturn false) mplus
17                (ComplexMHalf $* mreturn true)
18            };
19    }
20
21    ({boolean,boolean} -> Vec<{boolean,boolean}>) controlledNot() {
22        return ({boolean,boolean} b) ->
23            if (b.1 == true) {
24            }
25        }
26}
```
if (b.2 == true) {
    mreturn {true, false}
} else {
    mreturn {true, true}
}
} else {
    mreturn {b.1,b.2}
};

Emphasizing the creation of a superposition state, in the hadamard method, on line (14) and (17) we can note the use of the mplus and scalar product operators, explaining the reasons to create these operators in the proposed language. In the case of controlledNot method we can see the language performing operations over more than one qubit. It is important to note the way that the operator was created, returning a lambda expression, allowing to work with the >>= composition operator similarly to what happens with functional languages.

The next example shows the code with complex operations, aiming to perform transformations over an initial state with several qubits, considering the previously defined classes.

class QExec extends Object {
    // Constructor and other methods
    Vec<{boolean,boolean,boolean}> composedOperation() {
        return let qop = new QOp() in
            ({boolean,boolean,boolean} state) ->
                ((qop.hadamard()).invoke(state.3)) >>=
                (boolean b) ->
                    ((qop.controlledNot()).invoke({state.1,state.2})) >>=
                    ({boolean,boolean} tm) ->
                        ((qop.hadamard()).invoke(b)) >>=
                        (boolean ba) -> mreturn {tm.1,tm.2,ba};
    }
    Vec<{boolean,boolean,boolean}>
        exec({boolean,boolean,boolean} ini) {
            return new QState<{boolean,boolean,boolean}>(ini)
                .transform(this.composedOperation());
        }
    }
    new QExec().exec({true,true,true});

This example shows a way to apply partial transformations over the quantum state, and also how to compose operations through the bind operator in line (7), (9) and (11). The composedOperation method acts over a quantum state with three qubits and performs in sequence the operator hadamard to the third qubit, the operator controlledNot to the first and second, and again applying the hadamard to the first qubit, to finally return the result of the algorithm. The method exec shows an entry point to class processing.

4.1 The Deutsch Algorithm

The Deutsch algorithm is the simplest example that demonstrates the power of quantum parallelism. Its first version was presented by David Deutsch [10], and
in addition with Richard Feynman work [11], they launched the field of quantum computing [19].

This algorithm aims to determine whether a boolean function is balanced or constant. If \( f(0) = f(1) \), then the function is constant, otherwise is balanced. In a classical algorithm, to solve this problem it is necessary to evaluate the function \( f \) twice, i.e. \( f(0) \) and \( f(1) \), and then comparing the results [31]. Using the quantum approach, the problem is solved consuming only one verification, taking half of time when compared to the classical version [19] by using the principle of superposition, which allows the evaluation of two entries at the same time.

For the quantum version of this algorithm, first it is necessary to build a quantum version of the function \( f \), which represents a unitary transformation \( U_f \) performing the same computation of \( f \). All the quantum computations must be reversible, thus we need to model the function \( U_f \) using two qubits, wrapping the function \( f \), as we can see on the next expression:

\[
U_f |x\rangle |y\rangle = |x\rangle |y \oplus f(x)\rangle
\]

The following piece of code models a method called blackbox between line (5) to (13), which represents the \( U_f \) function, written using FJQuantum.

```java
1 class QExec<T extends Object> extends Object {
2     Vec<T> state;
3     // Constructor and other methods
4     
5     ({boolean,boolean} -> Vec<{boolean,boolean}>) blackbox((boolean -> boolean) f) {
6         return ({boolean,boolean} state) ->
7             if (state.2 == (f).invoke(state.1)) {
8                 return {state.1,false}
9             } else {
10                return {state.1,true}
11             }
12         }
13     }
```

This method receives as parameter a closure, representing the classical function \( f \), and returns another closure representing the unitary transformation \( U_f \). It returns a closure to allow the use of the bind operator. We can note the use of two qubits, represented by a tuple of two booleans.

Considering the unitary transformation \( U_f \) previously modeled in the blackbox method, we can follow the explanation about the quantum circuit for this algorithm, as Figure 1 presents.

![Quantum Circuit](image)

Fig. 1. Quantum circuit for Deutsch algorithm.

The first step in this circuit is to create the quantum state, where the first qubit starts with the pure value \(|0\rangle\) and the second with the value \(|1\rangle\). Using FJQuantum syntax, we write `mreturn {false, true}`.
After the creation of quantum basic states, the trick is to apply the hadamard gate on both qubits, to create a superposition state. To accomplish this task, we create the method hadtb, which applies the previously defined method hadamard over the top and bottom qubits, as we can see in the next code example.

```java
class QExec<T extends Object> extends Object {
  Vec<T> state;
  // Constructor and other methods
  ({boolean,boolean} -> Vec<{boolean,boolean}>) hadtb() {
    return let qop = new QOp() in
      ({boolean ,boolean} state) ->
        ((qop.hadamard()).invoke(state.1)) >>=
          (boolean ta) ->
            ((qop.hadamard()).invoke(state.2)) >>=
              (boolean ba) -> mreturn {ta,ba};
  }
}
```

The result of processing the method hadtb is mathematically represented as:

\[
|+\rangle |-\rangle = \frac{1}{2}(|00\rangle - |01\rangle + |10\rangle - |11\rangle)
\]

The states \(|+\rangle\) and \(|-\rangle\) represent the state superposition of |0⟩ and |1⟩ respectively, as presented in the next equation:

\[
|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \quad \text{e} \quad |-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)
\]

As soon as the quantum states are in superposition, we can apply our blackbox function. The next expression shows the quantum state after processing the \(U_f\) function for the entries |0⟩ and |1⟩.

\[
U_f |x\rangle |0\rangle = |x\rangle |f(x)\rangle
\]

(4)

\[
U_f |x\rangle |1\rangle = |x\rangle |1 \otimes f(x)\rangle
\]

Considering these equations, for processing the \(U_f\) function over a superposition state, we have the following result.

\[
U_f |x\rangle |-\rangle = \frac{1}{2} |x\rangle (|0\rangle - |1\rangle) \text{ if } f(x) = 0
\]

(5)

\[
U_f |x\rangle |-\rangle = \frac{1}{2} |x\rangle (|1\rangle - |0\rangle) \text{ if } f(x) = 1
\]

The next expression represents the quantum transformation to determine the result of that algorithm.

\[
U_f |x\rangle |-\rangle = (\neg f(x)) |x\rangle |-\rangle
\]

(6)

Then, we can see below the interpretation for the quantum transformation application.

\[
U_f |+\rangle |-\rangle = \begin{cases} |+\rangle |-\rangle \text{ if } f(0) = f(1) \\ |-\rangle |-\rangle \text{ if } f(0) \neq f(1) \end{cases}
\]

(7)
In the end, we apply the hadamard function over the first qubit (and applies a measurement) to see if the function is constant or balanced [31]. The deutsch method holds the entire processing of that algorithm.

```java
1  class QExec<T extends Object> extends Object {
2    Vec<T> state; // Constructor and other methods
3  
4  Vec<T> deutsch((boolean -> boolean) f) {
5    return let qop = new QOp() in
6      (((this.state) >>=
7        this.hadtb()) >>=
8        this.blackbox(f)) >>=
9      {{boolean,boolean} tb} ->
10        mreturn {{{(qop.hadamard()).invoke(tb.1)), tb.2};
11    };
12  }
```

Then, the result is presented in the following form.

\[ |0\rangle |\rangle \text{ if } f(0) = f(1) \text{ (constant)} \]

\[ |1\rangle |\rangle \text{ if } f(0) \neq f(1) \text{ (balanced)} \]

The presented examples show how to express quantum algorithms in FJQuantum, taking advantage of the object-oriented paradigm, and demonstrate how the monadic quantum layer fits in the original FJ as well.

5 Related Work

A quantum programming language is an important tool to work and to formally reason about quantum algorithms. For this reason, there is an effort on investigating semantic models and quantum programming languages, despite the absence of quantum hardware. Quantum languages are proposed, in general, using the imperative or the functional paradigm.

The first quantum programming language was developed considering the imperative paradigm, and was proposed by Knill [17]. More complete programming languages in this paradigm were proposed by Omer [22], Sanders and Zuliani [23], and Bettelli et al. [6], among others. Considering the functional paradigm, Selinger [24] has been seen as a pioneer, working together with Valiron [26]. In this paradigm, one can cite the work of Altenkirch and Grattage which introduced a functional programming language for pure quantum computations [2] and the proposal of Van Tonder, which works with a \(\lambda\)-calculus also considering pure quantum computations [28], among others [4], [1], [9].

The work of Vizzotto et al. [30,32] has inspired the approach used in this work, through the use of monads for simulating quantum computing. Besides that, the work of Calegaro and Vizzotto [8] presents a starting point to use monads in object-oriented languages.
6 Conclusion and Future Work

This paper presents a high-level description of $FJQuantum$, an object-oriented language, showing the relevant concepts of its construction, as well as several examples of programs that handle quantum computing concepts, through a monadic layer extending Featherweight Java.

We believe that this language can be used to facilitate the learning efforts about quantum computing concepts by conventional programmers, reusing their previous knowledge about object-oriented languages. Beyond that, it is possible to simulate quantum algorithms through the developed interpreter.

As future work, it is possible to develop syntactical adjustments to improve the visualization of quantum states in the source-code, implement a syntactic-sugar to write code using the monadic layer similarly to imperative languages, and also add the measurement operation in the proposed language.

References


