Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 324 (2016) 67–77

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

FJQuantum A Quantum Object Oriented Language

Samuel S. Feitosa[1] Juliana K. Vizzotto[2] Eduardo K. Piveta[3]

Department of Languages and Computer Systems
Universidade Federal de Santa Maria
Santa Maria - RS, Brazil

Andre R. Du Bois[4]

Department of Computing - CDTec
Universidade Federal de Pelotas
Pelotas - RS, Brazil

Abstract

Several languages and libraries has 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

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 *FJQuantum* language, showing a set of source-code examples for handling quantum computing concepts, taking advantage of the object-oriented paradigm.

The *FJQuantum* language is an extension of Featherweight Java (*FJ*) [16], which is a small calculus, providing a formal semantics for the main aspects of the Java language. Thus, *FJQuantum* provides all the features of FJ, adding several characteristics to simulate quantum behavior through a monadic approach. The *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 *QJava* library [8], which provides mechanisms to simulate quantum computing in Java using *closures*. Here we explore the capacity to write code in the *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 *bit*, which represents the classical binary physical system, being able to represent only two states (*true* or *false*, 0 or 1). Every information is described as a combination of *bits*.

In quantum computing, the basic information unit is called a *quantum bit* or *qubit*. The *qubit* presents an essential difference if compared to the *bit*, because it is not confined to the basic states of the classical *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 *qubit* as a vector [5]

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

where each of its positions stores the *probability amplitude* α and β, representing each of its basic states. The probability amplitudes are represented as *complex numbers*, such that $|\alpha|^2 + |\beta|^2 = 1$.

Intuitively, one can imagine a *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 α and β represents a *quantum superposition* of $|0\rangle$ and $|1\rangle$. These *superposition* states provide to quantum computing

[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 α and β) 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.

```
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*.

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

```
25                  if (b.2 == true) {
26                      mreturn {true, false}
27                  } else {
28                      mreturn {true, true}
29                  }
30              } else {
31                  mreturn {b.1,b.2}
32              };
33      }
34  }
```

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.

```
1   class QExec extends Object {
2     // Constructor and other methods
3
4     Vec<{boolean,boolean,boolean}> composedOperation() {
5       return let qop = new QOp() in
6         ({boolean,boolean,boolean} state) ->
7         ((qop.hadamard()).invoke(state.3)) >>=
8         (boolean b) ->
9         ((qop.controlledNot()).invoke({state.1,state.2})) >>=
10        ({boolean,boolean} tm) ->
11        ((qop.hadamard()).invoke(b)) >>=
12        (boolean ba) -> mreturn {tm.1,tm.2,ba};
13    }
14    Vec<{boolean,boolean,boolean}>
15      exec({boolean,boolean,boolean} ini) {
16        return new QState<{boolean,boolean,boolean}>(ini)
17              .transform(this.composedOperation());
18    }
19  }
20
21  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:

$$U_f |x\rangle |y\rangle = |x\rangle |y \oplus f(x)\rangle \tag{1}$$

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

```
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          mreturn {state.1,false}
9        }
10       else {
11         mreturn {state.1,true}
12       };
13     }
14  }
```

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.

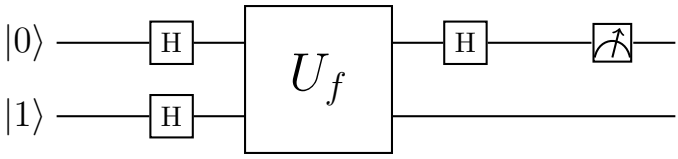

Fig. 1. Quantum circuit for *Deutsch* algorithm.

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.

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

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

$$|+\rangle\,|-\rangle = \frac{1}{2}(|00\rangle - |01\rangle + |10\rangle - |11\rangle) \tag{2}$$

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

$$|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \qquad \text{e} \qquad |-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \tag{3}$$

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 $|0\rangle$ and $|1\rangle$.

$$\begin{aligned} U_f\,|x\rangle\,|0\rangle &= |x\rangle\,|f(x)\rangle \\ U_f\,|x\rangle\,|1\rangle &= |x\rangle\,|1 \otimes f(x)\rangle \end{aligned} \tag{4}$$

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

$$\begin{aligned} U_f\,|x\rangle\,|-\rangle &= \tfrac{1}{2}\,|x\rangle\,(|0\rangle - |1\rangle) \text{ if } f(x) = 0 \\ U_f\,|x\rangle\,|-\rangle &= \tfrac{1}{2}\,|x\rangle\,(|1\rangle - |0\rangle) \text{ if } f(x) = 1 \end{aligned} \tag{5}$$

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

$$U_f\,|x\rangle\,|-\rangle = (-1)^{f(x)}\,|x\rangle\,|-\rangle \tag{6}$$

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

$$U_f\,|+\rangle\,|-\rangle = \begin{cases} |+\rangle\,|-\rangle & \text{if } f(0) = f(1) \\ |-\rangle\,|-\rangle & \text{if } f(0) \neq f(1) \end{cases} \tag{7}$$

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.

```
1   class QExec<T extends Object> extends Object {
2    Vec<T> state;
3    // Constructor and other methods
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.

$$
\begin{aligned}
&|0\rangle\,|-\rangle \text{ if } f(0) = f(1) \text{ (constant)} \\
&|1\rangle\,|-\rangle \text{ if } f(0) \neq f(1) \text{ (balanced)}
\end{aligned}
\tag{8}
$$

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 λ-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

[1] S. Abramsky. High-level methods for quantum computation and information. In *Logic in Computer Science, 2004 in 19th Annual IEEE Symposium*, pages 410–414, July 2004.

[2] T. Altenkirch and Grattage J. A functional quantum programming language. In *Proceedings of the 20th Annual IEEE Symposium on Logic in computer science*, 2005.

[3] Thorsten Altenkirch, Jonathan Grattage, Juliana K. Vizzotto, and Amr Sabry. An algebra of pure quantum programming. *Electron. Notes Theor. Comput. Sci.*, 170:23–47, 2007.

[4] P. Arrighi and G. Dowek. Linear-algebraic Lambda-calculus: higher-order, encodings and confluence. *eprint arXiv:quant-ph/0612199*, December 2006.

[5] M. Bellia and M.E. Occhiuto. Java: Proving type safety for java simple closures. *CSP2010*, pages 61–72, 2010.

[6] S. Bettelli, Luciano Serafini, and T. Calarco. Toward an architecture for quantum programming. *CoRR*, cs.PL/0103009, 2001.

[7] S. Boixo, T. F. Rønnow, S. V. Isakov, Z. Wang, D. Wecker, D. A. Lidar, J. M. Martinis, and M. Troyer. Evidence for quantum annealing with more than one hundred qubits. *Nature Physics*, 10:218–224, March 2014.

[8] Bruno Crestani Calegaro and Juliana Kaizer Vizzotto. Quantum monad using java closures. In *Theoretical Computer Science (WEIT), 2013 2nd Workshop-School on*, pages 34–39, Oct 2013.

[9] Bob Coecke and Ross Duncan. Interacting quantum observables. In *Automata, Lang. and Prog.*, volume 5126 of *Lec. Notes in Computer Science*, pages 298–310. Springer, 2008.

[10] David Deutsch. Quantum theory, the church-turing principle and the universal quantum computer. 400:97–117, 1985.

[11] R. Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21:467–488, 1982.

[12] Jonathan James Grattage, James Chapman, Alex Green, Mark Jago, Wouter Swierstra, and Mauro Jaskelioff. A functional quantum programming language. In *In: Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science*, pages 249–258, 2005.

[13] S. Hallgren, C. Moore, M. Rtteler, A. Russel, and P. Sen. Limitations of quantum coset states for graph isomorphism. In *STOC'06: Proceedings of the Thirty-eight anual ACM symposium on Theory of Computing*, pages 604–617, New York,USA, 2006.

[14] P.A. Hiskett, D. Rosenberg, C.G. Peterson, R.J. Hughes, S. Nam, A.E. Lita, A.J. Miller, and J.E. Nordholt. Long-distance entanglement-based quantum key distribution over optical fiber. *New Journal of Physics*, 8(9), 2006.

[15] D. Hucul, M. Yeo, W.K. Hensinger, J. Rabchuk, S. Olmschenk, and C. Monroe. On the transport of atomic ions in linear and multidimensional ion trap arrays. Quantum Physics e-prints, Feb. 2007.

[16] Atsushi Igarashi, Benjamin C Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for java and gj. *ACM Trans. on Prog. Lang. and Systems*, 23(3):396–450, 2001.

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

[18] T. Lanting, J. Przybysz, A. Yu. Smirnov, A. M. Spedalieri, F. H. Amin, M. J. Berkley, A. R. Harris, F. Altomare, S. Boixo, P. Bunyk, N. Dickson, C. Enderud, P. Hilton, J. E. Hoskinson, W. Johnson, M. E. Ladizinsky, N. Ladizinsky, R. Neufeld, T. Oh, I. Perminov, C. Rich, C. Thom, M. E. Tolkacheva, S. Uchaikin, B. Wilson, A. and G. Rose. Entanglement in a quantum annealing processor. *Phys. Rev. X*, 4:021041, May 2014.

[19] N. D. Mermin. *Quantum Computer Science: An Introduction*. Cambridge University Press, New York, USA, 2007.

[20] Shin-Cheng Mu and Richard Bird. Functional quantum programming. In *Asian Workshop on Programming Languages and Systems*, KAIST, Dajeaon, Korea, dec 2001.

[21] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, New York, NY, USA, 10th edition, 2011.

[22] Bernhard Omer. A procedural formalism for quantum computing. Technical report, 1998.

[23] J. W. Sanders and P. Zuliani. Quantum programming. In *In Mathematics of Program Construction*, pages 80–99. Springer-Verlag, 1999.

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

[25] Peter Selinger. Dagger compact closed categories and completely positive maps: (extended abstract). *Electronic Notes in Theoretical Computer Science*, 170(0):139 – 163, 2007. Proceedings of the 3rd International Workshop on Quantum Programming Languages (QPL 2005).

[26] Peter Selinger and Benoît Valiron. A lambda calculus for quantum computation with classical control. *J. Mathematical Structures in Computer Science*, 16(3):527–552, 2006.

[27] R. Laflamme Y. Nakamura C. Monroe T. D. Ladd, F. Jelezko and J. L. OBrien. Quantum computers. *Nature Physics*, 464/online, 2010.

[28] Andre van Tonder. A Lambda calculus for quantum computation. *SIAM J.Comput.*, 33:1109–1135, 2004.

[29] L. M. K. Vandersypen, M. Steffen, G. Breyta, C. Yannoni, M. H. Sherwood, and I.L. Chuang. Experimental realization of shor's quantum factoring algorithm using nuclear magnetic resonance. *Nature*, 414:883–887, 2001.

[30] Juliana K. Vizzotto, Thorsten Altenkirch, and Amr Sabry. Structuring quantum effects: Superoperators as arrows. *J. Mathematical Structures in Computer Science*, 16:453–468, 2006.

[31] Juliana Kaizer Vizzotto. *Structuring General and Complete Quantum Computations in Haskell: The Arrows Approach*. PhD thesis, Universidade Federal do Rio Grande do Sul, 2006.

[32] Juliana Kaizer Vizzotto, Bruno Crestani Calegaro, and Eduardo Kessler Piveta. A double effect λ-calculus for quantum computation. In *Programming Languages*, volume 8129 of *Lecture Notes in Computer Science*, pages 61–74. Springer Berlin Heidelberg, 2013.

[33] Juliana Kaizer Vizzotto, André Rauber DuBois, and Amr Sabry. The arrow calculus as a quantum programming language. In *Logic, Language, Information and Computation*, volume LNAI 5514 of *Lecture Notes in Computer Science*, pages 379–393. Springer, 2009.