Methods and Patterns for User-Friendly Quantum Programming

Alexandros Singh, Konstantinos Giannakis, Kalliopi Kastampolidou, and Christos Papalitsas

Abstract The power and efficiency of particular quantum algorithms over classical ones has been proved. The rise of quantum computing and algorithms has highlighted the need for appropriate programming means and tools. Here, we present a brief overview of some techniques and a proposed methodology in writing quantum programs and designing languages. Our approach offers "user-friendly" features to ease the development of such programs. We also give indicative snippets in an untyped fragment of the Qumin language, describing well-known quantum algorithms.

Keywords Quantum programming • Quantum programming language • Functional programming • Qumin

1 Introduction

With Moore's law reaching an apparent plateau, attention to unconventional computing paradigms is ever increasing. Quantum computation, that is, computing based on quantum mechanical principles, is among the most sought-after of these. While quantum computing is still in relative infancy, quantum algorithms show very promising results. For example Grover's algorithm, which can be used as a database search algorithm, offers a quadratic speed-up over its classical counterparts, outpacing any classical algorithm [1].

Grover's algorithm can also be used to brute force a symmetric cryptographic key with orders of magnitude more efficiency than any other classical algorithm. Another popular quantum algorithm is Shor's algorithm which can factor any integer N in polynomial time and could make many modern cryptographic systems (such as RSA) obsolete [2].

The above observations have attracted the attention, not only of academia, but also of the industry and various funding sources. The pursuit of novel and efficient

A. Singh (✉) • K. Giannakis • K. Kastampolidou • C. Papalitsas
Department of Informatics, Ionian University, Corfu, Greece
e-mail: p13sing@ionio.gr; kgiann@ionio.gr; p12kast@ionio.gr

P. Vlamos (ed.), *GeNeDis 2016*, Advances in Experimental Medicine and Biology 989, DOI 10.1007/978-3-319-57348-9_17

computing technologies has already led to the raise of investments and funding schemes that aim to profit from proposed current and future quantum computing systems, with D-Wave System being a notable example [3].

Quantum programming has experienced a surge of interest, with many theoretical models being proposed from quantum circuits to lambda calculi/type systems [4–8], quantum logics, and quantum assembly languages [9–11]. Quantum Programming Languages (QPL) allow us to argue about quantum algorithms beyond the hardware-like level of quantum circuits which they are frequently described in. Such a higher level description of quantum algorithms would include features such as data structures, procedures, and syntactic constructions such as control flow statements: recursion, loops, conditionals, etc.

Works like [12] have tried to formulate some basic requirements one would expect a QPL to fulfil. These vary accordingly to the underlying paradigm, with frequent requirements amongst others being: completeness, extensibility, abstracting away and being independent from the underlying machinery, and being expressive enough to allow one to define quantum data structures, oracles etc., handling of measurement, handling of quantum memory/registers. A QPL that fulfils the aforementioned requirements would open the road for the application of quantum computing in various areas such as networks, databases, cryptography and telecommunications, leading to revolutionary innovations in many fields crucial to our modern computing-intensive world.

In this work we present a brief overview of some techniques, algorithms and patterns we consider helpful in writing quantum programs and designing languages that offer "user-friendly" features to ease the development of such programs. We also give indicative code snippets in a fragment of the Qumin language.

Qumin has an experimental implementation in a Python programming language environment [13], using the libraries `numpy` for matrix/vector calculations and `parsimonious` for parsing. The implementation consists of the interpreter for the language, tools for parsing and typechecking and auxiliary tools for parsing type signatures and automatically generating various types.

This paper is organized as follows: Sect. 2 includes the related work. In Sect. 3 we describe an extension of the untyped lambda calculus, whereas Sect. 4 is our main contribution. Specifically, we discuss and illustrate by examples, the proposed techniques for programming in a quantum framework. Finally, a discussion of our results and plans for future work is included in Sect. 5.

2 Related Works

For a comprehensive introduction to quantum computing we refer the reader to the work of Nielsen and Chuang in [14]. Various models and paradigms have been defined for quantum programming and a handful of fully-fledged quantum programming languages have already been implemented, as we discuss below.

The field of quantum algorithms has produced a number of very interesting works. Among them, Shor's [2] and Grover's algorithms [1] are some of the better known ones. Quantum programming has experienced a surge of interest, with many theoretical models being proposed from the well-established quantum circuits to experimental (typed and untyped) lambda calculi, type systems [4, 6–8], and quantum assembly languages [9–11].

Such models include the popular QRAM model: a register machine capable of performing quantum operations (such as preparing a quantum state, unitary transformations and measurement of quantum registers), which is controlled by a classical computer. Some descriptions and/or implementations of QPLs include the functional languages Quipper [6], QML [4], QPL [11], QLISP [10], and the imperative languages QCL [12, 15] and LanQ [16].

In [17] the authors Sanders and Zuliani also present a quantum programming language, the qGCL based on the Guarded Command Language, along with its formal semantics. Additionally, the above work includes some examples of actual quantum algorithms expressed in the aforementioned language.

3 A Naive Extension of the Untyped Lambda Calculus

To prepare the ground for our upcoming discussion of algorithms and patterns, it would be beneficial to first discuss the theory of untyped lambda calculus, extended with some primitive operations and constants, in order to facilitate operations in Hilbert spaces H, which we will refer to as λ_H.

t :=	(term)
x	(variable)
v	(vector)
U	(operator)
$(U \cdot v)$	(operator application)
$(v \otimes v)$	(tensor product)
$measure(v)$	(measurement)
$\lambda x.t$	(abstraction)
$t\ t$	(application)

Where, for a given Hilbert space H,

- v belongs to the set of normalized vectors of H.
- U belongs to the set of matrix representations of unitary operators of H.
- $U \cdot v$ is operator application, by way of matrix multiplication: Uv.
- $v \otimes v$ is the tensor/Kronecker product of two vectors/matrices.
- $measure(v)$ is measurement of state v in the computational basis. (returns state after collapse)

In practice, the parentheses and the multiplication dot can be omitted when the meaning is clear. For example, Deutsch's algorithm is expressed in λ_H as such:

$$\lambda U_f.measure((H \otimes I)U_f(H \otimes H)(|0\rangle \otimes |1\rangle))$$

Where U_f is the matrix that corresponds to the oracle of a binary function:

$$f : \{0, 1\} \to \{0, 1\}$$
$$U_f(|x, y\rangle) = |x, y \oplus f(x)\rangle$$

4 Programming in Qumin

We will focus on the dynamically typed fragment of the language Qumin. The central construct we are interested in is that of an function as captured by the lambda abstraction. For example $\lambda x.x + 5$ is written in Qumin as such:

$$\texttt{lambda(x)\{(x + 5)\}}$$

Lambda abstractions can be invoked in-line by including arguments in a parenthesis as such:

$$\texttt{lambda(x, y)\{(x + 5)\}(3, 5)}$$

Which would evaluate to `8`.

Qumin, being a functional programming language, places great significance in the notion of functions. Functions are first-class citizens, in that they can be passed around and returned as any other primitive, like lists or numbers, and can be bound to identifiers. The returned value of a function is the last evaluated expression in its body. For example, a function that takes another function and applies it to an argument:

$$\texttt{lambda(f, x)\{f(x)\}(lambda(x)\{(x + x)\}, 5)}$$

Which evaluates to `10`.

To define a named function, we attach a lambda abstraction to an identifier. For example $f(x) = x + 5$ is written in Qumin as such:

```
let f = lambda(x){
        (x + 5)
}
```

And can be invoked as such:

$$\texttt{f(5)}$$

Which of course evaluates to 10.

Qumin also supports implicit partial application:

```
let f(x,y){
    (x + y)
}
let partiallyApplied = f(10)
partiallyApplied(30) => 40
```

Finally, specifically in the case of binary functions, we can also call them in infix notation: `(argument1 function argument2)`. For example:

```
let myOp = lambda(x,y){
    parindent (x + (3 * y))
}
(5 myOp 10) => 35
```

Arithmetic operators (`+,-,*/`) in Qumin are defined as any other function would be, we just call them infix for clarity.

4.1 Quantum Programming in Qumin

4.1.1 Vectors and Matrices

Vectors and matrices are of central importance in quantum computing, where they represent the state/qubits of a system and unitary operators/gates respectively. In Qumin vectors and matrices are implemented using lists and lists of lists. For example a state $|\psi\rangle = a|0\rangle + b|1\rangle$ in the two-dimensional space H, is written in Qumin as such:

```
let psi = [ab]
```

While, for example, the identity matrix that corresponds to the identity operator in H would be written as:

```
let identity = [[1 0]
                [0 1]]
```

Naturally, as the dimension of H increases, the process of writing matrices by hand quickly gets unwieldy. For example, for 4 qubits one would be expected to write a 16x16 (256 values) matrix by hand. To tackle this problem, we can eschew the use of matrix representations and work with linear operators as functions. This alleviates the aforementioned problem of having to manually define multi-dimensional matrices by hand. E.g. the identity operator is always $f(x) = x$, regardless of the space's dimension. Unfortunately this has the side-effect of making things like finding eigenvalues/eigenvectors much more difficult, while also introducing a severe slowdown in computations.

4.1.2 Matrix Generators

The solution to the aforementioned dilemma is given by a group of functions called (matrix) generators. Generators allow us to make use of a linear operator in its function form where convenient, and in its matrix form otherwise. A generator is a function that when given an linear operator $f : H \rightarrow H$ and a basis $\{v_i\}$ of H, generates f's matrix representation on H with respect to the basis. This allows us to write linear operators as functions, composing them and manipulating them as one would expect to manipulate a mathematical operator, and when we want to make use of its matrix representation, all we have to do is invoke the generator on it.

Matrix Generator Algorithm.
Inputs: $f : H \rightarrow H, \{v_i\}$
Outputs: $M_{dim(H)\times dim(H)}$
0: M $\leftarrow$ [ ]
1: For v in $\{v_i\}$:
2: append $f(v)$ to M
3: transpose M

For example, the identity operator is defined as such:

```
let identity = lambda(vec){
        vec
}
```

Then generating, for example, the identity matrix on a 16-dimensional (4-qubit) Hilbert space, amounts to running:

```
generateMatrix(identity,16)
```

Apart from allowing us to avoid writing big matrices by hand, generators allow us to define operators in a mathematical, easily-understood, and general with respect to dimension, way. For example the Quantum Fourier Transform is written in Qumin as such:

```
--load generator

let omega = lambda(jj,k,N){
        exp((fold(*, [2 pi 0+1i jj k]) / N))
}

let qfSum = lambda(limit,vec,index,N){
        if ((limit = 0)){
                0
        }
        else {
                ((omega(index,limit,N) * car(vec)) +
```

```
                qfSum((limit - 1), cdr(vec), index, N))
        }
}

let outer = lambda(vec,index,N){
        if((N = index)){
                []
        }
        else{
          append(((1 / sqrt(N)) * qfSum(N,vec,index,N)),
                              outer(vec,(index + 1),N))
        }

}

let qft = lambda(vec){
        let N = len(vec)
        outer(vec,0,N)
}
```

As we can see, the Qumin implementation closely follows the mathematical expression of QFT:

$$y_k = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} x_j \omega^{jk}$$

Where:

$$\omega^{jk} = e^{2\pi i \frac{jk}{N}}$$

The function `omega` implements ω^{jk} (ie the N^{th} root of unity), `qfSum` implements the sum $\sum_{j=0}^{N-1} x_j \omega^{jk}$, and `outer` builds the transformed vector (y_k) by multiplying each result of `qfSum` by $\frac{1}{\sqrt{N}}$.

4.1.3 Deutsch's Algorithm

We will now proceed to show an implementation of Deutsch's algorithm. Once again, we look back to λ_H. Quantum computation in λ_H is based on three primitive operations: $\cdot$, $\otimes$ and *measure*, which in Qumin are defined as functions named $\cdot$, $\otimes$ and `measure` respectively. If one wishes to avoid using unicode, he can use

the aliases `apply` for $\cdot$ and `tensor` for $\otimes$ instead. For example, we already have presented Deutsch's algorithm in λ_H so let us present the Qumin version:

```
--load generator
--load operators

let fConstant = lambda(x){
        [1 0]
}

let fBalanced = lambda(x){
        x
}

let deutsch = lambda(f){
        let H = generateMatrix(hadamard,2)
        let I = generateMatrix(identity,2)
        let Uf = oracle(generateMatrix(f,2))
        let state = ([1 0]  ⊗ [0 1])
        measure(((H ⊗ I) · (Uf · ((H ⊗ H) · state))))
}
```

As we can see, the body of `deutsch` closely resembles the corresponding lambda version: $\lambda U_f.measure((H \otimes I)U_f(H \otimes H)(|0\rangle \otimes |1\rangle))$

Running Deutsch's algorithm on the first example function, $f(x) = 0$ gives us:

```
deutsch(fConstant)

=> Probability of state 0 is 0.5
Probability of state 1 is 0.5
Probability of state 2 is 0.0
Probability of state 3 is 0.0
System collapsed to state: 0
```

While it on the second example function, $f(x) = x$ gives us:

```
deutsch(fBalanced)

=> Probability of state 0 is 0.0
Probability of state 1 is 0.0
Probability of state 2 is 0.5
Probability of state 3 is 0.5
System collapsed to state: 3
```

As expected.

One may notice that in the implementation of Deutsch's algorithm we made use of a function called `oracle`. The `oracle` function converts classical operators to unitary ones, allowing us to use them in our quantum computations. To do this,

`oracle` expects as input a binary function f and creates a new operator U that operates on a composite space, the tensor product of the domain of f as a qudit and an additional helper qudit. That is, for $f(x)$, `oracle` creates $U(x, y)$ defined as such: $U(x, y) = (x, y \otimes f(x))$.

5 Conclusion and Future Work

Notable works on quantum aspects of computing, like the well-known quantum algorithms of Shor and Deutsch have shown some prosperous signs. There is a need for deep understanding and examination of the computation processes that could be implemented. Works like this contribute in the field of quantum programming. Overall, since Quantum Computing is a quite new scientific field, the theoretical foundation of technologies and methodologies regarding this branch is still under research. Aiming to this direction, our work proposed a specific methodology to program and express quantum algorithms and computation processes.

As for future work, it would be of interest to use the language as a tool to study the computational aspects of quantum computation, such as using it to simulate variants of quantum automata that have interesting and useful properties, such as measure-once automata [18] and periodic quantum automata [19]. Apart from that, further implementations of quantum algorithms would also be of interest, serving to expose potential new features that are crucial to their implementation. Finally, the language could be used as an educational tool for familiarization with notions related to quantum computation and quantum algorithms.

References

1. Grover, L.K. 1997. Quantum mechanics helps in searching for a needle in a haystack. *Physical Review Letters* 79(2):325.
2. Shor, P.W. 1994. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science, 1994*, 124–134. Los Alamitos: IEEE.
3. D-Wave Systems, S. 2016. D-wave 2x. http://www.dwavesys.com/. [D-Wave 2X]
4. Altenkirch, T., and J. Grattage. 2005. A functional quantum programming language. In Logic in Computer Science, 2005. LICS 2005. Proceedings. 20th Annual IEEE Symposium on, 2005, 249–258. Chicago, IL, USA: IEEE.
5. Altenkirch, T., J. Grattage, J.K. Vizzotto, and A. Sabry. 2007. An algebra of pure quantum programming. *Electronic Notes in Theoretical Computer Science* 170:23–47.
6. Green, A.S., P.L. Lumsdaine, N.J. Ross, P. Selinger, and B. Valiron. 2013. Quipper: a scalable quantum programming language. In *ACM SIGPLAN Notices*, vol. 48, 333–342. New York: ACM.
7. Selinger, P., B. Valiron, et al. 2009. Quantum lambda calculus. In *Semantic Techniques in Quantum Computation*, 135–172. Cambridge: Cambridge University Press.
8. Van Tonder, A. 2004. A lambda calculus for quantum computation. *SIAM Journal on Computing* 33(5):1109–1135.

9. Blaha, S. 2002. Quantum computers and quantum computer languages: quantum assembly language and quantum c language. arXiv preprint quant-ph/0201082
10. Desmet, B., E. D'Hondt, P. Costanza, and T. D'Hondt. 2006. Simulation of quantum computations in lisp. In *3rd European Lisp Workshop, Co-Located with ECOOP*.
11. Selinger, P. 2004. Towards a quantum programming language. *Mathematical Structures in Computer Science* 14(04):527–586.
12. Ömer, B. 1998. A procedural formalism for quantum computing. Tech. rep., Department of Theoretical Physics, Technical University of Vienna.
13. QUIT Group, I.U. 2016. Qumin language project. https://github.com/wintershammer/QImp/. [Github repositiory, accessed 6/10/2016].
14. Nielsen, M.A., and I.L. Chuang. 2010. *Quantum Computation and Quantum Information*. Cambridge: Cambridge University Press.
15. Ömer, B. 2005. Classical concepts in quantum programming. *International Journal of Theoretical Physics* 44(7):943–955.
16. Mlnarik, H. 2007. Operational semantics and type soundness of quantum programming language lanQ. arXiv preprint arXiv:0708.0890.
17. Sanders, J.W., and P. Zuliani. 2000. Quantum programming. In *Mathematics of Program Construction*, 80–99. Berlin: Springer.
18. Moore, C., and J.P. Crutchfield. 2000. Quantum automata and quantum grammars. *Theoretical Computer Science* 237(1):275–306.
19. Giannakis, K., C. Papalitsas, and T. Andronikos. 2015. Quantum automata for infinite periodic words. In *6th International Conference on Information, Intelligence, Systems and Applications (IISA), 2015*, 1–6. Piscataway, NJ: IEEE.