Seminar Report

Quantum Programming Language

Samuel Rupprechter
Samuel.Rupprechter@student.uibk.ac.at

20 June 2015

Supervisor: Julian Nagele

Abstract

This seminar report is about cQPL, a quantum programming language, in which it is possible to model quantum communication. After an introduction to the necessary physics involved, the language will be introduced in two steps. First the classical part of cQPL is described and then the quantum part is added. Finally, some example usecases are shown.

Contents

1 Introduction **1**

2 Quantum Computers **1**
2.1 Computational Power 1
2.2 Quantum Bits 2
2.2.1 Superpositions and Randomness 2
2.2.2 Parallelism and Entanglement 3

3 cQPL **3**
3.1 Classical Language Elements 4
3.1.1 Identifiers and Variables 4
3.1.2 Arithmetic and Logical Expressions 4
3.1.3 Procedures 5
3.1.4 Control Flow 5
3.2 Quantum Language Elements 6
3.2.1 Quantum Data Types 6
3.2.2 Quantum Gates 7
3.2.3 Dump and Measure 8
3.3 Communication via Modules 9
3.4 Runtime Errors 9

4 Possible Usecases **9**
4.1 Generating Random Numbers 9
4.2 Qubit Exchange 10

5 Conclusion **10**

Bibliography **12**

1 Introduction

Even though real quantum computers do not actually exist right now[1], several programming languages, which might someday be used on such computers, have already been proposed [3]. The focus of this report, `cQPL`, is such a quantum programming language. It was introduced by Wolfgang Mauerer in [5] (and therefore the majority of this report, including the examples, is based on that publication) and is an extension of `QPL` which was described by Peter Selinger in [9]. Even though quantum computing was nothing new at the time of Selingers publication, quantum algorithms were often programmed with quantum circuit models or quantum Turing machines. Naturally these techniques do not really lend themselves to structured programming or abstractions such as data types. This led him to describe both syntax and semantics of `QPL`, which was supposed to be a first step towards *structured* quantum programming languages. Mauerer then expanded this language with "quantum communication".

In Section 2, I will give a short introduction to quantum computers and some quantum physics. Then in the main part of this report `cQPL`, in Section 3, is introduced, followed by the description of some scenarios for which quantum programming might be used in the future in Section 4.

2 Quantum Computers

In this chapter we will have a look at the similarities and differences between classical and quantum computers and describe some of the physics involved. Since the primary focus of this report lies on a programming language, I kept this chapter as short as possible while still trying to cover all the necessary parts. Most of this chapter is based on Ömer [6, 7] and Mauerer [5].

2.1 Computational Power

Alan Turing and Alonzo Church both stated in their own words what would later become known as the Church-Turing thesis:

Hypothesis 2.1 (Church-Turing)**.** *Every 'function which would naturally be regarded as computable' can be computed by the universal Turing machine.*

David Deutsch published a stronger, physical version of the statement in 1985:

Hypothesis 2.2 (Church-Turing-Deutsch)**.** *Every finitely realizable physical system can be perfectly simulated by a universal model computing machine operating by finite means.*

He immediately realized that classical physics cannot be simulated by a Turing machine, because it makes use of *real numbers*, which are different from *computable reals*. Instead, he proposed that quantum computers might be able to obey to his principle [2].

[1] Experiments show that the theoretical concepts of quantum computing actually work, but at the moment only for a small number of quantum bits. Programs written in the language that is presented here can not be executed by any "quantum computer" at the moment.

There exist problems in NP that can be solved in polynomial time by quantum computers (e.g. prime factorization). Nevertheless, it is not proven that quantum computers have greater computational power (i.e. can solve NP-complete problems efficiently) than classical computers (since it is for example not known if prime factorization can be done in polynomial time by classical computers). Note that computations that do not rely on the quantum part of a quantum computer are actually likely to be slower (since there is still the overhead of managing the unneeded quantum part).

2.2 Quantum Bits

A quantum bit (*qubit*) is the quantum analogue of the classical bit and can for example be modeled by electrons, which have a *spin*. The spin-direction (up or down) can then be used to model the values 0 and 1.

If we measure a qubit there are, like for the classical bit, two different possible states, which are usually denoted by $|0\rangle$ and $|1\rangle$. However, qubits have very interesting additional properties:

2.2.1 Superpositions and Randomness

Before we measure a quantum bit, we have no way of knowing its state, because it can be in all states at the same time. If we toss a coin and catch it, but do not look at it, the coin is in exactly one of two states (either heads or tails). This can be described by a probability distribution, but the only thing preventing us from knowing for sure, is our own ignorance.

Quantum bits are different: The state of a qubit can be described by a probability distribution, but before we look at it, it is actually in all states at the same time. This phenomenon is called the *superposition* of quantum bits and can be represented as a linear combination of $|0\rangle$ and $|1\rangle$:

$$|\psi\rangle = v_0|0\rangle + v_1|1\rangle \quad \text{with} \quad v_0, v_1 \in \mathbb{C} \quad \text{and} \quad |v_0|^2 + |v_1|^2 = 1$$

A common mathematical representation of qubits is as vectors of $\mathbb{C}^2$:

$$v_0|0\rangle + v_1|1\rangle \to \begin{bmatrix} v_0 \\ v_1 \end{bmatrix}, \text{ so that } |0\rangle \to \begin{bmatrix} 1 \\ 0 \end{bmatrix} \text{ and } |1\rangle \to \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

The probability that the outcome of our measurement will be $|0\rangle$ is $|v_0|^2$ and the probability that it will be $|1\rangle$ is $|v_1|^2$. Note that a measurement changes the state of the qubit and any consequent measurements will always yield the same result. For equal probabilities of the outcome being $|0\rangle$ or $|1\rangle$, we need a *symmetric* superposition ($v_0 = v_1 = \frac{1}{\sqrt{2}}$). It is important to emphasize that this means that qubits in symmetric superpositions possess a built-in *randomness*—there is no way of correctly predicting the outcome of the next measurement at all times (although the *probability* of the outcomes can be predicted). It is not hard to think of settings where this could be extremely useful and we will look at an example in Section 4.

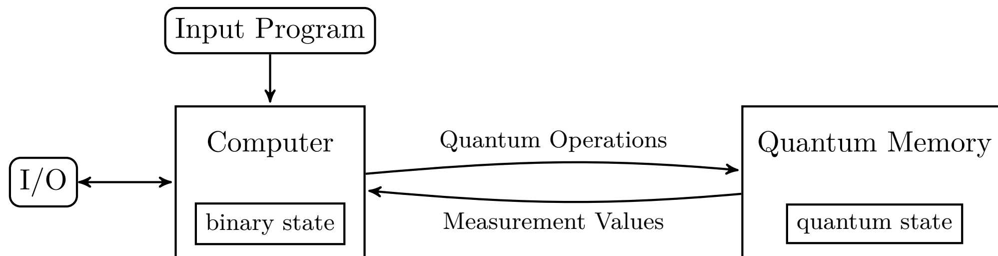

Figure 1: A possible quantum computer architecture.

We can extend our representation to two qubits by defining $|\alpha\beta\rangle$ as the state where the first qubit is in state α and the second one is in state β. Two qubits can then be represented as:

$$v_{00}|00\rangle + v_{01}|01\rangle + v_{10}|10\rangle + v_{11}|11\rangle \to \begin{bmatrix} v_{00} \\ v_{01} \\ v_{10} \\ v_{11} \end{bmatrix}$$

2.2.2 Parallelism and Entanglement

If we have a quantum register containing n quantum bits which are all in a symmetric superposition, the register contains *all* numbers from 0 to $2^n - 1$ at the same time. Whenever we manipulate the register, we therefore manipulate all these numbers *at the same time.* We call this *quantum parallelism.*

Qubits can be *entangled* with each other, which means that any state-changes of one of the qubits will influence the other qubit. This works regardless of the physical distance between the two qubits. This forms the basis of the communication part of `cQPL`, of which we will see an example later on.

3 cQPL

`cQPL` was designed to run on a special two-part machine—one part being a classical computer, the other one a *quantum memory.* This memory stores the quantum variables and can measure or manipulate them. The classical computer part does everything a normal computer would do (it controls the whole control flow of the machine), plus communicate with the quantum memory. Figure 1 shows an example of this architecture.

Even though some authors consider using such a machine-setup a loss of generality ([7]), because they argue that not every control flow can be described in a classical way, the author of `cQPL` disagrees. He argues that the language is centered on the assumption that the control flow of any program can be described by classical means, because all known quantum algorithms can be expressed in this framework.

Since at the time of the development of `cQPL` no real quantum memory actually existed, a quantum machine *simulator*, which was described in [6], was used to build a working

compiler for the language.

In the remainder of this section I will give a brief introduction to both the classical and quantum parts of the language.

3.1 Classical Language Elements

`cQPL` contains some basic features you would expect from an imperative programming language for classical computers. This is unsurprising, especially considering it is actually compiled to `C++` to run on the mentioned simulator. Even though `cQPL` only supports a small subset of the parts of `C++`, I will not describe everything in detail but only the most important parts. The language does, for example, not support user input and whole programs must be written inside a single file, because there is no import mechanism. Even though this sounds very restrictive, it is not a real restriction of `cQPL`, since it was never supposed to be used outside of academia and these features could easily be added in future versions. In general, every complete statement has to be concluded by a semicolon.

3.1.1 Identifiers and Variables

Variables and procedures can be given names consisting of characters `A-Z`, `a-z`, `_` and `0-9`, where no underscore and no digit must be at the beginning.

The available data types are `bit`, `int` and `float` (and some quantum data types which will be described later). A new variable can be declared using the keyword `new` and stating the data type. Assignments are done via the operator `:=`. It is mandatory to assign a value to a variable on declaration. Since integers and floats are internally represented by their corresponding data types in `C++`, they basically behave the same (e.g. they can overflow and be assigned to each other with the same automatic conversion). Note that this is actually not part of the language specification but rather a side-effect of using the mentioned quantum simulator.

```
new bit b := 0;
new int i := 42;
```

Example 1: Values are assigned to two newly declared variables.

3.1.2 Arithmetic and Logical Expressions

The basic mathematical operators `+`, `-`, `*`, `/` work as expected and can be used to form arbitrary arithmetic expressions (e.g. `a + (b - c) * (1/2)`). Even though all these operators work on `bits`, they may yield unexpected results: The bits are internally represented as integers and all the operations are executed on integers. Then it gets converted back to bits, with any result that is different from 0 converted to 1.

The logical operators `=`, `!=`, `>`, `<`, `>=` and `<=` yield one of the logical values `true` or `false`, which can be negated by using `!`. Any value different from 0 is seen as `false`.

Internally, `true` is just the value 0 and `false` is 1. You can assign logical expressions to variables, but the variable will then have result of the expression as value—either 0 or 1.

Even though the language definition states that two logical expressions can be combined using `&` (and) and `|` (or), this does not seem to work with our version of the `cQPL` complier.

3.1.3 Procedures

Procedures are declared with the keyword `proc`, followed by the name of the procedure and zero, one or more arguments, which are given as `name:type` pairs. The parameters are passed by value, meaning any change to the classical variables will not be visible outside the procedure.

The return value of a procedure is a tuple of the classical arguments given to the procedure at the end of its execution. The result can be ignored or assigned to a tuple of variables. Note that even though this means that tuples technically exist as a data type, they can not be assigned to variables. To call a procedure, the keyword `call` must be used.

```
proc my_procedure: arg1:int, arg2:int {
    arg1 := arg1 + arg2;
    arg2 := 0;
} in {
    new int var1 := 1;
    new int var2 := 2;
    (var1, var2) := call my_procedure(var1, var2);
};
```

Example 2: Declaring and calling a procedure.

Print Procedure

A special built-in procedure is called `print`, which unsurprisingly prints out a string (`print "Hello World!";`), the result of an arithmetic expression (`print 2 * 3;`) or the value of a variable (`print x;`) to the command line. Similar to tuples, strings can not be assigned to variables.

3.1.4 Control Flow

`cQPL` offers `while` loops and `if-then-else` segments to direct the control flow of a program. The `else`-block may be omitted. Unfortunately there is a bug in the compiler we used which would cause the `else` part to always be executed! A simple workaround would be to use a second `if` with the negated condition. The next example shows a simple program that will print out every third number between 0 and 100 (starting from 3) using the mentioned workaround.

```
new int i := 0;
new int d := 0;
while (i < 100) do {
    if (d >= 3) then {
        print i;
        d := 0;
    };
    if (d < 3) then {
        d := d + 1;
    };
    i := i + 1;
};
if (i >= 100) then {
    print "Success!";
};
```

Example 3: Prints every third number.

3.2 Quantum Language Elements

So far we have not seen any special elements of the language. If we stopped here, `cQPL` would just be a boring regular programming language, which could be used on any ordinary computer. This will change now as I introduce the quantum part of the language.

3.2.1 Quantum Data Types

In addition to the classical data types, `cQPL` offers two more data types—`qbit` and `qint`. The declaration and assignment syntax stays the same, but note that the only valid assignments to qbits are 0 or 1 (they can not be initialized with superpositions):

```
new qbit b := 0;
new qint i := 42;
```

Example 4: Values are assigned to two newly declared quantum variables.

The difference is of course that `qbit` and `qint` represent *quantum* bits and integers (and they are stored in the quantum memory). Just like their classical counterparts, quantum integers can internally be represented by quantum bits.

An important trait of quantum variables is that they can not be passed by value. When passing them to a procedure, they are always passed by reference, since it is not possible to copy quantum variables [9]:

Theorem 3.1 (No-Cloning)**.** *It is not, in any physically meaningful way, possible to duplicate a quantum bit.*

cQPL prevents this cloning *syntactically*, which is important for the prevention of runtime errors (see Section 3.4).

3.2.2 Quantum Gates

A quantum gate operates on one or more quantum bits. By applying a gate to a quantum bit, the state of the bit may change. With the representation of qubits as vectors it is common to represent gates as matrices, so that applications of gates to qubits are simple matrix multiplications. cQPL comes with several predefined gates, including:

Logical Negation (Not)
This gate, also called *Pauli-X gate*, simply "negates" the qbit is is applied to (it switches v_0 and v_1 in $\begin{bmatrix} v_0 \\ v_1 \end{bmatrix}$):

$$\text{Not} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Controlled Not (CNot)
Is applied to 2 qbits and can be used to *entangle* them. The first one changes to the XOR-product of itself with the second one. The second qbit remains unchanged: $\texttt{CNot} : |x, y\rangle \to |x \oplus y, y\rangle$

$$\text{CNOT} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Hadamard-Transform (H)
The Hadamard Transformation changes a vector in several ways. Its most important feature is that it brings the qubits $|0\rangle$ and $|1\rangle$ into even superpositions when applied to them:

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

The other predefined gates are *Fourier Transformation* and *Phase Shift*, but they will not be elaborated here. It is easy to define your own gates by specifying the wanted matrix as list of (complex) numbers enclosed in [[and]], but these user-defined gates can not be assigned to variables. Gates are applied to quantum variables via the *= operator, as we can see in this next example:

```
new qbit b1 := 0; new qbit b2 := 1;
b1 *= Not;
b1, b2 *= CNot;
b1 *= [[1, 0, 0i, 1]];
```

Example 5: Applying gates to quantum bits.

3.2.3 Dump and Measure

There are two built-in procedures which only work on quantum variables:

To measure a quantum variable we use `measure`, which returns a classical result. In general, running a program (which contains `measure`) multiple times, will yield different outputs. The result depends on the state of the quantum variable and this state is described by a probability distribution. The measurement is irreversible and will always return $|0\rangle$ or $|1\rangle$.

The `measure` procedure also allows for a new `measure-then-else` control flow segment, where the code in the then block will execute if the `qbit` measured is $|0\rangle$. Otherwise (whenever it is $|1\rangle$), the code in the else block will execute.

To output the probability distribution of one or more quantum variables we use `dump`.

```
new qbit q := 0;
print "State before Hadamard-Transform: "; dump q;
q *= H;
print "State after Hadamard-Transform: "; dump q;
```

Example 6: Example usage of dump.

The output of this program will always be:

```
State before Hadamard-Transform: 1 |0>
State after Hadamard-Transform: 0.5 |0>, 0.5 |1>
```

It tells us, what we expected: Before applying the Hadamard gate, `q` is in state $|0\rangle$ with a probability of 100%. Afterwards it is in an even superposition. Actually *measuring* a qbit has a different effect:

```
measure q then {
    print "q is |0>";
} else {
    print "q is |1>";
};
```

Example 7: Example usage of measure.

The output will be exactly one of these two lines (and might change in consecutive runs):

```
q is |0>
```

or

```
q is |1>
```

3.3 Communication via Modules

The fundamental part of communication is the *module*, which contains the code a communication-partner runs. Every module must have a unique identifier. Two new language pieces are added: `send var1, var2, ... to ModuleID` to send and `receive var1:type, var2:type, ... from ModuleID` to receive data. Received variables are declared implicitly.

```
module A {
    new qbit q := 0;
    send q to B;
    ...
};

module B {
    receive r:qbit from A:
    ...
};
```

Example 8: Communication between two modules.

3.4 Runtime Errors

Selinger proved that `QPL` can avoid all runtime errors (programs would never crash or enter an invalid state) by detecting them all at compile time [9], which he attributes mostly to the strong static type system of the language. The most important part that is that the No-Cloning theorem is enforced by the syntax. He claims that this is the only (quantum) programming language in which this is possible right now.

Unfortunately, with the introduction of communication to `cQPL`, it was not possible to keep this nice property, because some programs can not be checked at compile time. However, it would still be possible to guarantee no runtime errors by restricting the code to the limited subset of `QPL`, although this would not be very practical.

4 Possible Usecases

Now that we have seen the special properties of a quantum programming language, it is time to look at some scenarios where these characteristics might be useful.

4.1 Generating Random Numbers

As already mentioned, there is an intrinsic randomness connected to quantum bits in a superposition. This can be used to easily produce *real* random numbers as opposed to the random numbers generated by pseudo random number generators that are mostly in use today. In the following code-example we see how a `qbit` is brought to a symmetric

superposition through the Hadamard gate and then measured to simulate a random coin-toss.

```
new qbit q := 0;
q *= H;
measure q then {
    print "Tossed head";
} else {
    print "Tossed tail";
};
```

Example 9: Random Coin Toss.

4.2 Qubit Exchange

This example shows how a single qubit can be exchanged between two modules. The two qubits `a` and `b` are first entangled in Module `A` and then one of them is sent to Module `B`. The two measurements will always yield the same result: either $|0\rangle$ or $|1\rangle$ (with equal probability), because each measurement can affect the states of both entangled qubits.

```
module A {
    new qbit a := 0;
    new qbit b := 0;
    b *= Not;
    b *= H;
    a, b *= CNot;
    send b to B;
    measure a then { print "A has a = |0>"; }
              else { print "A has a = |1>"; };
};

module B {
    receive b:qbit from A;
    measure b then { print "B has b = |0>"; }
              else { print "B has b = |1>"; };
};
```

Example 10: Quantum bit exchange.

5 Conclusion

This report gave a short introduction to `cQPL`, which is just one of several quantum programming languages that have been proposed so far (see surveys and comparisons

in [5] and [3]). After Deutsch [2] introduced the quantum Turing machine in 1985, Ömer [6, 7] developed *QCL*, the first real quantum programming language. It does not support communication and it is possible to write programs in it that would create "unphysical" states (which are impossible in the real world). The *Q Language*, which was presented in [1] by Bettelli et al., has the same problems. The programming language *qGCL*, which was introduced by Sanders and Zuliani in [8], supports communication, but still does not respect physics, while *QML* prevents unphysical states, but communication can not be expressed in it [4].

A lot of research is currently ongoing for both the design of new languages and the physical implementation of quantum computers. Concerning language design, a possible next step would be to develop complex quantum data structures, which would make high-level programming easier. We have seen in this report that working quantum computers will have very nice features and could someday replace classical computers in many usecases, but probably not for quite a few years.

References

[1] S. Bettelli, T. Calarco, and L. Serafini. Toward an architecture for quantum programming. *Eur. Phys. J. D, Vol. 25, No. 2, pp. 181-200 (2003)*, 2001. `arXiv:cs/0103009`.

[2] D. Deutsch. Quantum theory, the church-turing principle and the universal quantum computer. In *Proceedings of the Royal Society of London A 400*, pages 97–117, 1985.

[3] S. J. Gay. Quantum programming languages: survey and bibliography. *Mathematical Structures in Computer Science*, 16:581–600, 2006.

[4] J. Grattage and T. Altenkirch. QML: Quantum data and control. submitted for publication, February 2005.

[5] W. Mauerer. Semantics and simulation of communication in quantum programming. Diploma Thesis, University Erlangen-Nuremberg, 2005.

[6] B. Ömer. A procedural formalism for quantum computing. Master's thesis, Vienna University of Technology, 2003.

[7] B. Ömer. *Structured Quantum Programming.* PhD thesis, Vienna University of Technology, 2003.

[8] J. Sanders and P. Zuliani. Quantum programming. In R. Backhouse and J. Oliveira, editors, *Mathematics of Program Construction*, volume 1837 of *Lecture Notes in Computer Science*, pages 80–99. Springer Berlin Heidelberg, 2000.

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