A research on quantum programming languages

Tomás Carneiro*
Universidade do Minho
(Dated: June 21, 2021)

The goal of the presented essay is to provide an introduction to two different Quantum Programming Languages, and to illustrate their respective use. As there are two main groups of QPLs: imperative and functional, one of each will be described thoroughly. The QPLs chosen for this research were: the imperative higher-level *Silq* and functional *QML*.

I. INTRODUCTION

As quantum computing emerges as one of the most enigmatic and challenging fields of computer science, it has been accompanied by a crescent number of unique and resourceful new quantum programming languages.

Considering the existence of well-known quantum algorithms such as Shor's and Grover's, there has always been keen interest in building languages that were capable (and easy enough to understand) of expressing these kinds of algorithms.

Similarly to classical computation, we are faced with two different language paradigms: imperative and functional, which have quite different focuses.

Functional languages focus mainly on what information is desired and what transformations are required. State changes are non-existent and flow control is assured by function calls, including recursion.

Imperative languages, on the other hand, work by using sequential sets of steps in order to perform tasks. State changes must be tracked and flow control is done using loops, conditionals and function calls.

II. QML LANGUAGE

QML is a quantum programming language with a structure familiar to functional programmers (resembling *Haskell*), which supports reasoning and and algorithm design. Though functional (programs are expressions), QML is first order and finitary; all datatypes are finite.

A. FCC's and FQC's

This language is presented with semantics in which terms are interpreted as morphisms in the category of finite quantum computations (**FQC**'s). This notion was developed by analogy with finite classical computations (**FCC**'s) [1]

In short, given finite sets A and B, a morphism $(H, h, G, \phi) \in FCC\ A\ B$ is given by: [6]

- a finite set of initial heaps H,
- an initial heap $h \in H$,
- a finite set of garbage states G,
- a bijection $\phi \in A \times H \approx B \times G$,

while a morphism $(H, h, G, \phi) \in FQC\ A\ B$ is given by:

- a finite set H, the basis of the space of initial heaps,
- a heap initialisation vector $h \in \mathbb{C}^H$,
- a finite set G, the basis of the space of garbage states,
- a unitary operator $\phi \in A \otimes H \multimap B \otimes G$.

These types of morphisms can be represented by the diagram:

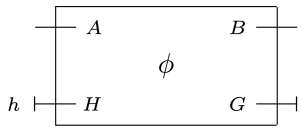

Figure 1. FCC / FQC diagram

Given two computational systems we can compose them by combining initial and final heaps: [6]

* Correspondence email address: tomasini20@gmail.com

[1] FCC may be viewed as a categorical account of a finite version of Bennet's results [7]

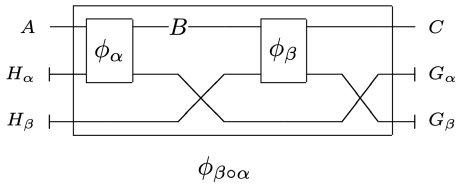

Figure 2. FCC / FQC diagram for 2 computational systems

More formally, given the morphisms α and β:

$$\alpha = (H_\alpha, h_\alpha, G_\alpha, \phi_\alpha) \in FCC\ A\ B$$
$$\beta = (H_\beta, h_\beta, G_\beta, \phi_\beta) \in FCC\ A\ B$$

the composite morphism $\beta \circ \alpha = (H, h, G, \phi)$ is:

$$H = H_\alpha \times H_\beta$$
$$h = h_\alpha \times h_\beta$$
$$G = G_\alpha \times G_\beta$$
$$\phi = (G_\alpha \times \phi_\beta) \circ (H_\beta \times \phi_\alpha)$$

In the analogous quantum case, given morphisms:

$$\alpha = (H_\alpha, h_\alpha, G_\alpha, \phi_\alpha) \in FQC\ A\ B$$
$$\beta = (H_\beta, h_\beta, G_\beta, \phi_\beta) \in FQC\ A\ B$$

the composite morphism $\beta \circ \alpha = (H, h, G, \phi)$ is:

$$H = H_\alpha \otimes H_\beta$$
$$h = h_\alpha \otimes h_\beta$$
$$G = G_\alpha \otimes G_\beta$$
$$\phi = (G_\alpha \otimes \phi_\beta) \circ (H_\beta \otimes \phi_\alpha)$$

Classical and Quantum morphisms can thus be compared in terms of their own operators and state spaces:

FCC (Classical case)	FQC (Quantum case)
Finite sets	Finite dimensional Hilbert spaces
Cartesian product	Tensor product
Bijections	Unitary operators

B. Syntax

The symbols σ, τ are used to vary over QML types, given by $\sigma = \mathcal{Q}_1 \mid \mathcal{Q}_2 \mid \sigma \otimes \tau$, where the type constructor is the tensor product $\otimes$ corresponding to a product type and $\mathcal{Q}_2$ is a qubit type. x, y, z are used to vary over names. Constants $\kappa, \eta \in \mathbb{C}$ are also used to define the syntax of expressions. The vector notation γ is used for sequence variables to be measured. QML's grammar can thus be defined:

- **Variables:** $x, y, \cdots \in Vars$
- **Probability amplitudes:** $\kappa, \eta, \cdots \in \mathbb{C}$
- **Patterns:** $p, q ::= x \mid (x, y)$
- **Terms:**

$$\begin{aligned} t, u ::= & \ x \mid x^\gamma \mid () \mid (t, u) \mid \texttt{let}\ p = t\ \texttt{in}\ u \\ & \mid \texttt{if}\ t\ \texttt{then}\ u\ \texttt{else}\ u' \mid \texttt{if}^\circ\ t\ \texttt{then}\ u\ \texttt{else}\ u' \\ & \mid \texttt{qfalse}^\gamma \mid \texttt{qtrue}^\gamma \mid \kappa \times t \mid t + u \end{aligned}$$

Quantum data is modelled using the constructs $\kappa \times t$ and $t + u$. The term $\kappa \times t$, associates the probability amplitude κ with the term t. The term $t + u$ describes a quantum superposition of t and u. Quantum superpositions are first class values, and can be used in conditional expressions to provide quantum control. For example: `if° (qtrue+qfalse) then t else u` evaluates both t and u and combines the results in a quantum superposition.[6]
Finally, a QML program is a sequence of function definitions, where a function definition is given by $f\Gamma = t : \tau$. A Haskell-style syntax is used to present program examples. For example, the QML function below:

$$f(x_1 : \sigma_1, x_2 : \sigma_2, .., x_n : \sigma_n,) = t : \tau$$

is equivalent to the following Haskell-like code:

$$f : \sigma_1 \multimap \sigma_2, .. \multimap \sigma_n \multimap \tau$$
$$f\ x_1\ x_2\ ..\ x_n\ =\ t$$

C. Decoherence control

Reversibility plays a more central role in quantum computation due to the fact that forgetting information leads to decoherence, which destroys entanglement, and hence negatively affects quantum parallelism. Thus one of the central features of this language is control of decoherence, which is achieved by keeping track of weakening through the use of strict logic and by offering different if-then-else (or, alternatively, case) operators, one that **measures the qubit**, `if`, and a second, `if°` , that **doesn't** – but which can only be used in certain situations. These operators are explicitly different: one observes a qubit and thus leads to decoherence; the other is free of decoherence but requires that we derive that the two alternatives live in orthogonal spaces.
Consider the following expression: `if°` x `then` t `else` u. This should only be accepted if $t \perp u$. This notion intuitively ensures that the conditional operator does not implicitly discard any information about x during the evaluation. The branches of a superposition should

also be orthogonal for similar reasons. Mathematically, two terms, t, u, are orthogonal if their inner-product is equal to zero, $\langle t|u\rangle = 0$. If this is the case then the judgement $t \perp u$ is true, but if the inner-product yields any other value then t is not orthogonal to u. [6] This can be demonstrated with some simple examples:

```
forget : Q₂ ⊸ Q₂
forget x = if x
           then true
           else true
```

This program works, and x is measured, and thus, "forgotten".

```
forget : Q₂ ⊸ Q₂
forget x = if° x
           then true
           else true
```

This program does not compute, as true $\not\perp$ true.
We can, however, use the `if°` operator in the definition of a Hadamard gate:

```
had : Q₂ ⊸ Q₂
had x = if° x
        then { false | (−1) true }
        else { false | true }
```

Since these alternatives are orthogonal, our program typechecks.

D. Other examples

Using the decoherence-free operator `if°`, other standard reversible and hence quantum operations can be implemented, such as qnot, CNOT and Toffoli gates:

```
qnot : Q₂ ⊸ Q₂
qnot x = if° x
         then false
         else true

cnot : Q₂ ⊸ Q₂ ⊸ Q₂ ⊗ (Q₂ ⊗ Q₂)
cnot c x = if° c
           then (true, qnot x)
           else (false, x)

ccnot : Q₂ ⊸ Q₂ ⊸ Q₂ ⊸ Q₂ ⊗ Q₂
ccnot  c x y = if° c
               then (true, cnot x y)
               else (false, (x,y))
```

III. SILQ LANGUAGE

Silq is a high-level programming language for quantum computing with a strong static type system. Since this language is strongly static-typed, this means that the type of the variable is known at the compile time of the program and the variables defined in the program can only be of a specific data type, i.e., string data cannot be added with integer data types. It allows expressing quantum algorithms more safely and concisely than existing quantum programming languages, while typically using only half the number of quantum primitives. This way, programs become simpler, shorter and usually easier to read.

A. The problem of uncomputation

Analogously to the classical setting, quantum computations often produce temporary values. However, as a key challenge specific to quantum computation, removing such values from consideration induces an implicit measurement collapsing the state [1]. In turn, collapsing can result in unintended side-effects on the state due to the phenomenon of entanglement. This problem cannot be prevented, as due to the quantum principle of deferred measurement [1], preserving values until computation ends is equivalent to measuring them immediately after their last use. To remove temporary values from consideration without inducing an implicit measurement, algorithms in existing languages must **explicitly** *uncompute* all temporary values, i.e., modify their state to enable ignoring them without side-effects. *Silq* allows the uncomputation of temporary values to be an **automatic process**. This is, however, only applicable if (i) the original evaluation of the uncomputed value *can be* described classically, and (ii) the variables used to evaluate it *are preserved* and can thus be leveraged for uncomputation. [3]

B. Annotations and Data types

Silq supports the following data types: [2]

- **B** or $\mathbb{B}$: Boolean data type used to denote 0 and 1 binary digits
- **N** or $\mathbb{N}$: The classical natural numbers set, $[0, 1, \dots]$
- **Z** or $\mathbb{Z}$: The classical integers set, $[\dots, -1, 0, 1, \dots]$
- **Q** or $\mathbb{Q}$: The classical rational numbers set
- **R** or $\mathbb{R}$: The classical real numbers set
- **1** or $\mathbb{1}$: The singleton type, which contains a single element

- `uint[n]`: n-bit unsigned integers, where $n \in \mathbb{N}$
- `int[n]`: n-bit signed integers in two's complement, where $n \in \mathbb{N}$
- τ^n : vector of length n, where $n \in \mathbb{N}$
- $\tau[\,]$: array of dynamic length
- $!\tau$: type τ, but restricted to classical values
- $\tau_{in,0} \times \cdots \times \tau_{in,n} \to \tau_{out}$: functions, whose input and output types can be annotated

Variable assignment makes use of two different symbols: `":="` and `":"`, which assign data to a variable and assign a particular data type to a variable, respectively. Suppose we want to create a qubit initialized as 0 and apply a Hadamard gate to it. The correct syntax should be as follows:

```
x := 0:B;
x := H(x);
```

Data type **conversion** is possible, and pretty straightforward. Let's see the example below:

```
a := vector(5, 0:B);
b := a as int[5];
```

Using the keyword `as`, we can safely convert variable `a`'s type from a quantum 5 bit vector to a quantum 5 bit integer.

Silq also allows for the use of different **annotations** [2] such as:

- $!\tau$: the `"!"` is used to indicate that the type τ may only hold classical types (and not superpositions); i.e., `42` has type $!\mathbb{N}$, which is classical, while `H(0)` contains the result $\frac{1}{\sqrt{2}}(|0\rangle+|1\rangle)$, which is not classical, and therefore has type $\mathbb{B}$ and not $!\mathbb{B}$;
- `qfree`: used to denote the evaluation of functions or expressions that do not create or destroy superpositions. This annotation helps with automatic uncomputation and it ensures that if `qfree` is being evaluated on classical expressions, then it should only yield classical outputs; i.e., `H(0)` is not `qfree` as it creates the superposition $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$, as we've seen, but `X` gate, however, is `qfree`, as it neither introduces nor destroys superpositions; in this case, it would map $\sum_{b=0}^{1} \gamma\,|\psi\rangle \xrightarrow{X} \sum_{b=0}^{1} \gamma\,|1-\psi\rangle$
- `mfree`: used to denote that a function can be evaluated without using any measurement operator; i.e. function `f` defined by:

```
def f(x:B) mfree:B { return H(x); }
```

  is `mfree`, since it doesn't measure the state `x`;

- `const`: indicates that a variable will not be changed in the given context. Concretely, each parameter of a function and each variable in the context may be annotated as const; i.e., int the following function:

```
def eval(const x:B, f: const B! -> B ) {
    return f(x);
}
```

  `const` keyword is used for the `x` variable and the `f` function to make them constant throughout the `eval()` function scope.

- `lifted`: `qfree` functions with constant arguments are known as `lifted` functions; i.e., boolean operators `&&` (and) and `||` (or), `+,-,*,/` operators, $\oplus$ (bitwise-xor), `sin, cos` are all `lifted` operators. In the function:

```
def Or(x:B, y:!B) lifted{ return x || y; }
```

  `x` and `y` variables are implicitly `const`.

C. Control flow

Similar to other programming languages, *Silq* provides classical and quantum control flow statements, which decide the order of execution of the code we write. Control flow is also known as conditionals, where a condition is provided and executed when some criteria are fulfilled. The statements are executed whenever a certain condition is satisfied.
In *Silq*, conditionals are particullarly useful for making controlled gates, such as CNOT, CZ, etc. Syntax for a controlled Hadamard gate is given as follows:

```
if (x) { y := H(y); }
```

The code shows that if `x` qubit is true (meaning it is 1), then a Hadamard operation will be applied to the `y` qubit, which is the controlled Hadamard operation. For quantum control flow, however, *Silq* does enforce some restrictions: [4]

- The branches `if` and `else` must be `mfree`, which means no measures can be done in either of these;

- The condition can be implicitly uncomputed at the end of the two branches: the condition is `lifted` and all variables occurring in it are left `const` by both branches;
- Both branches cannot perform classical operations such as defining variables, arrays, or functions of the classical type. This is to prevent accidental superposition of classical variables, arrays or functions.

Iterations in computer programming are the repeated execution of code statements until they meet a certain condition. These are also known as **loops** in computer programming. *Silq* offers two different kinds of iteration statements, namely `while` and `for` loops. These loops can be used accordingly to the following syntax:

- `while x { ... }` : `x` must be classical;
- `for i in [a..b) { ... }` : for-loop from `a` (inclusive) to `b` (exclusive); `a` and `b` must be classical;
- `for i in (a..b] { ... }` : for-loop from `a` (exclusive) to `b` (inclusive); `a` and `b` must be classical;

Suppose we want to implement a function that creates an equal superposition of qubits using Hadamard gate. The code can be written as follows:

```
def uniformSuperposition[n:!N]():uint[n] {
   x := 0:uint[n];
   for i in [0..n) { x[i] := H(x[i]); }
   return x;
}
```

In this function, classical input `n` is used to initialize a `n`-bit unsigned integer, `x`, after which a for-loop iterates `x`'s bits, applying Hadamard to each of them and thus, creating an equal superposition.

D. Program structure

Similar to the classical programming language *C*, a program written in *Silq* consists of a `main` function, where code will be executed when the program runs, and optionally other additional functions.
A simple program which makes use of our previously created `uniformSuperposition` function can be written as follows:

```
def uniformSuperposition[n:!N]():uint[n] {
   x := 0:uint[n];
   for i in [0..n) { x[i] := H(x[i]); }
   return x;
}

def main() {
    return uniformSuperposition[1]();
}
```

For which the output obtained will be:

```
(0.707107+0i)·|1⟩+(0.707107+0i)·|0⟩
```

Which correponds to:

$$\frac{1}{\sqrt{2^n}}\sum_{k=0}^{2^n-1}|k\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle).$$

If we call function `uniformSuperposition` for 2 qubits, instead, the output is:

```
(0.5+0i)·|1⟩+(0.5+0i)·|2⟩+(0.5+0i)·|3⟩+(0.5+0i)·|0⟩
```

Which correponds to:

$$\frac{1}{\sqrt{2^n}}\sum_{k=0}^{2^n-1}|k\rangle = \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle).$$

IV. CONCLUSION

Along this essay, we learned about the different semantics and uses of both *QML* and *Silq* languages. We covered several important aspects in quantum programming such as FCC's and FQC's, decoherence control and uncomputation. We were also able to verify the big difference in level between the two languages, since *Silq* is most likely the most *high-level* quantum programming language developed to the day.

An implementation of the *Quantum Counting* algorithm using `Silq` was developed in the practical assignment of this course, where we'll cover more of `Silq`'s quantum programming functionalities.

[1] M. A. Nielsen, I. L. Chuang, *Quantum Computation and Quantum Information* (Cambridge University Press, Cambridge).
[2] Benjamin Bichsel, Timon Gehr, Maximilian Baader,

Martin Vechev, *Silq: A High-Level Quantum Language with Safe Uncomputation and Intuitive Semantics* (ETH Zurich, Switzerland).

[3] Srinjoy Ganguly, Thomas Cambier, *Quantum Computing with Silq Programming* (Packt, Birmingham).

[4] Silq Documentation, `https://silq.ethz.ch/documentation`.

[5] Giacomo Nannicini, *An Introduction to Quantum Computing, Without the Physics* (IBM T.J. Watson, Yorktown Heights, NY).

[6] Thorsten Altenkirch, Jonathan Grattage, *A functional quantum programming language*, (School of Computer Science and IT, Nottingham University)

[7] C. H. Bennett, *Logical reversibility of computation*, (IBM Journal of Research and Development)