

Programming language for unified computing with classical and quantum bits

Gergely Gálfi¹, Tamás Kozsik², Zoltán Zimborás³

Abstract: As the capabilities of quantum computers - regarding the numbers of physical qubits, the number of consecutive coherence-keeping steps, or the noise of the gates - are continuously improving, so they enable the implementation of complex algorithms, including ones which combine classical and quantum-enabled steps. Such computations will need control and data shared between the two worlds (quantum and classical), and for improving code quality by eliminating redundancies they may also need a form of classical/quantum polymorphism. This paper proposes a high-level programming language, Qubla to support this type of polymorphism, allowing not only genericity, but also mixed classical and quantum arithmetics. In this language, arithmetical and logical operations can be performed on data formed of both classical bits and qubits. The current implementation of the language works as a interpreter/compiler hybrid: all the classical steps of the program are executed consecutively and together with them the quantum steps are added as building blocks to a quantum logic definition over qubits.

1 Introduction

Many of the quantum algorithms have steps which are quantized version of some common classical procedures. Typical example is Shor's algorithm[Sh94], where to prepare the incoming state of the QFT, controlled modular multiplications of a quantum and classical integers are employed. These steps are usually considered as trivial, in the sense that they require "only" the reimplementing of classical logical circuits with quantum gates. On the other hand, with the increasing number of logical qubits this originally straightforward task of quantum circuit design is becoming far too complex to effectively done manually. This was our main motivation for developing a programming language which allows users to specify their algorithms through constructions similar to the ones in the classical procedural programming languages. Also we wanted to make it transparent as much as possible whether in a given algorithm, classical or quantum bits are used. Certainly there are other programming languages which provide this type of transparency, e.g. Silq[Bi19]. However, we had some specific needs which made us to develop a compiler from scratch instead of using or tweaking an existing one. It is best to compare our language with Silq to shed light on our drivers during the development process:

¹ ELTE Eötvös Loránd University, Budapest, Hungary, galfi@inf.elte.hu, 0000-0002-6568-877X

² ELTE Eötvös Loránd University, Budapest, Hungary, kto@inf.elte.hu, 0000-0003-4484-9172

³ Wigner Research Centre for Physics, Budapest, Hungary, zimboras.zoltan@wigner.hu, 0000-0002-2184-526X

- There are simulation methods or other numerical analysis tasks around quantum computing which require to pass multiple times through the chain of unitary transformations making up the quantum logic itself. In these cases it is more effective to calculate all the transformations in advance. That's why we needed a system which does not simulate or directly run the quantum steps on a physical hardware during compilation time - as Silq does -, but rather builds up the quantum logic as a whole. It was our definite intention to separate the task of building up the logical structure of a quantum circuit based on a specified algorithm and the task of running that quantum circuit either on a real hardware or on a simulator. As our framework focuses on the former task, the main final product of our compiler is a quantum logic definition, so it was also practical to create it as a library and not a command-line tool. Further on we will refer to this framework as Qubla[22]. Our choice of environment for implementing it was Python, as it gives many opportunity to feed the resulting quantum logics into the very extensive QC ecosystem in Python.
- Also we wanted to focus our attention on transformations which are boolean or - from classical logical engineering standpoint - could be described by truth tables. There are theorems which ensure that any boolean function could be built up from some universal gates (e.g. NAND-gates), but even with a more practical or ergonomical approach, it is common to define these functions as a sequence of some custom few-bit boolean operations. So one of our goal was to get even the most basic operations as addition, multiplication, division, etc. defined by a sequence of truth table operations, and making these operations partially or entirely quantized by substituting qubits in place of classical bits appropriately. Turning a classical boolean transformation into a quantum one requires to fulfill some mathematical constraints (like the unitarity of the operation), but we've seen this "housekeeping" as a task should be done automatically by the compiler and wanted to create the framework as transparent as possible for this classical-quantum transformation.
- Having the truth table transformations as a separate entity group - and not seeing them as just some general unitary transformations - helps us on one hand to produce a more optimized quantum logic (regarding the number of qubits and steps). On the other hand it could also help the physical implementation of these transformations: Solovay-Kitaev theorem[Ki97] only ensures the existence of an approximate implementation for general unitary transformations however keeping in eyesight that these are boolean ones it is guaranteed that they could be generated exactly by a finite number of elementary operations.

Summarily, we wanted to develop a framework which can process the source code in a hybrid manner: should the users operate with classical quantities (bits, fixed length words or integers), then the instructions are executed on the spot. However, when a user-defined function encounters a qubit-based quantity, it adds a new steps to the quantum logic definition. In the typical case of mixed-type arguments, it is up to the compiler to do in advance as many precomputations as possible, and add only the minium necessary qubit-operations to

the quantum logic definition. So, the final result after processing the source code is the usual output of the classical steps (e.g. lines printed on the console), but additionally a quantum logic definition is generated.

2 Object types

To understand how Qubla works as a unified classical-quantum language, it is worth taking a look at first some of the available data types. Qubla is a dynamically typed language. Converting object to a given type is easy, could be done with applying the desired type as a single-argument function. For example, `int(x)` will convert object `x` to an `int`, provided by the original type of `int(x)` allows that.

2.1 Integer type

Similar to other programming languages, Qubla uses an integer type. Our intention was to avoid any bit number constraint in this language, so the integer type is arbitray-length. To avoid the quirks of indefinite length quantum objects, integer type don't have quantum version, for that purpose the fixed length word type should be used.

2.2 Bit and qubit type

The most important atomic data types of the language are the bit and qubit types. The bits are the usual binary values, the two distinct values could be accessed by `bit(0)` and `bit(1)`. The quantum counterpart of bit objects are the `qbit` objects. These are representing the qubits of the quantum computer. Technically a `qbit` object contains only a reference to a certain qubit, e.g. `qbit[5]` is a reference to the fifth qubit of the quantum computer. As it is clear from this definition, the compiler - apart from the initialization - doesn't keep track of the time evolution of qubits as we consider it a task for a simulator or for a real hardware implementation. However, the compiler guarantees that when referenced multiple times, the qubits wouldn't change their values. This may sounds too restrictive, however this condition is the quantum correspondent of the classical principle that for example `bit(1)` ought to have the same meaning while the program is running. We say that a qubit is *invariable* under a series of quantum steps, if in the case these steps applied on any pure canonical base state, then the density matrix of the qubit doesn't change.

2.3 Word types

To handle more complex (semi)quantum integer-like numbers, Qubla provides fixed length word types, specifically `word` and `uword`. Both of them refers to a finite list of bits and

qubits, and the only difference between the two is how certain functions - e.g. the “less than” function - handle them: `uword` is an unsigned integer, `word` should be considered as a two’s complement signed value. Managing of word objects are relatively easy in Qubla, even when it is a mix of classical bits and qubits. For example, `uword{[bit,bit,qbit,qbit]}(13)` will result in a word object similar to `uword{[1,0,qbit[4],qbit[5]]}` (actual indexing of qubits depends on the preceding part of the source code and also on the state of the compiler), where `qbit[4]` and `qbit[5]` are newly allocated (not used before) qubits and both of them are initialized to the state $|1\rangle$.

3 Quantum logic steps

The following building blocks are employed by Qubla:

- **Initialization steps** As the name suggests, this type of step is responsible for setting one or more qubits to a given quantum state.
- **Truth table steps** These are special permutation transformations where the mapping of the canonical basis states could be defined through the usual truth tables. Certainly not every truth table leads to a permutation, and henceforth a unitary transformation (e.g. the two bit AND operation). However, it is guaranteed by the Qubla compiler, that for any truth table, with arbitrary inputs and outputs, it will be extended by the inclusion of input qubits into the output till unitarity is achieved required by any reversible computing model[To80].
- **Copy steps** Qubla also guarantees the invariability of qubits in the sense of 2.2. To achieve that goal it duplicates the input qubits certainly only in the classical sense, as could be done by a CNOT gate with the controlled input set to zero. So copy steps would well fit into the group of general truth table transformations, however during reduction phase some of these copy steps are to be removed, and it is easier to find them if they are already belong to their own group.

4 Test case

Despite this use case doesn’t have any practical usage in Quantum Computing (it’s output could be simply found out), it was able to drive the Qubla framework through all it’s compilation and reduction steps in a non-trivial way, so giving it an end-to-end testing. Additionally it is a good example on how easily a complex quantum logic could be defined in Qubla. In this text case we define a quantum logic to calculate all the possible products of two 3-bit unsigned word (practically the numbers 0-7). The actual Qubla-code which defines this logic is the following:

```
H = {0: 1/sqrt(2), 1: 1/sqrt(2)};
```

```

n = 3;
arrqbx = alloc(n); arrqby = alloc(n);
for(i : seq(n)){
    arrqbx[i] = qstate(H)[0];
    arrqby[i] = qstate(H)[0];
}
x = uword{n}(arrqbx); y = uword{n}(arrqby);
z = x*y;
output(z);

```

For variables x and y we create 2×3 qubits, all being in equal superposition (so all possible input numbers has equal probability). The statement $z = x*y$ calculates the 6-bit arithmetic product of these inputs. The ultimate line `output(z)` makes it sure that the qubits corresponding to z and all other steps leading up to it will not be removed in the reduction phase. After compiling and reducing the quantum logic, we ended up in a 21-qubit system. This is well within the capability of a simple state vector simulation. So after simulating this system and then calculating the probabilities of the possible values of z , we were able to compare it with the theoretically predictable distribution (the distribution of the product of two numbers between 0 and 7, picked in an uniformly random way). Having an exact match give us a strong evidence that each individual steps in our Qubla framework work correctly.

4.0.1 Acknowledgements

This project has received funding from the Ministry of Innovation and Technology and the National Research, Development and Innovation Office within the Quantum Information National Laboratory of Hungary and through grant No. FK135220.

References

- [22] Qubla source code, 2022, URL: <http://absimp.org/qubla>.
- [Bi19] Bichsel, B.; Baader, M.; Gehr, T.; Vechev, M.: Silq: A High-level Quantum Programming Language, Sept. 2019, URL: <http://silq.ethz.ch/>, visited on: 09/22/2019.
- [Ki97] Kitaev, A. Y.: Quantum computations: algorithms and error correction. Russian Mathematical Surveys 52/6, pp. 1191–1249, Dec. 1997, URL: <https://doi.org/10.1070/rm1997v052n06abeh002155>.
- [Sh94] Shor, P. W.: Algorithms for Quantum Computation: Discrete Logarithms and Factoring, 1994.
- [To80] Toffoli, T.: Reversible computing. International Colloquium on Automata, Languages, and Programming/, pp. 632–644, 1980.