Quantum types: going beyond qubits and quantum gates

Tamás Varga
Constructor Institute Schaffhausen
Schaffhausen, Switzerland
tamas.varga@constructor.org

Yaiza Aragonés-Soria
Constructor Institute Schaffhausen
Schaffhausen, Switzerland
yaiza.aragonessoria@constructor.org

Manuel Oriol
Constructor Institute Schaffhausen
Schaffhausen, Switzerland
Constructor University Bremen
Bremen, Germany
mo@constructor.org

ABSTRACT

Quantum computing is a growing field with significant potential applications. Learning how to code quantum programs means understanding how qubits work and learning to use quantum gates. This is analogous to creating classical algorithms using logic gates and bits. Even after learning all concepts, it is difficult to create new algorithms, which hinders the acceptance of quantum programming by most developers.

This article outlines the need for higher-level abstractions and proposes some of them in a developer-friendly programming language called Rhyme. The new quantum types are extensions of classical types, including bits, integers, floats, characters, arrays, and strings. We show how to use such types with code snippets.

CCS CONCEPTS

• **Software and its engineering** → **Imperative languages**; • **Computer systems organization** → *Quantum computing*; • **Hardware** → Quantum computation.

KEYWORDS

Quantum computing, Quantum programming, Quantum data types, High-level abstractions

1 INTRODUCTION

Why are there so few algorithms for quantum computing? In our opinion, the adoption of quantum computing is hindered by the lack of high-level quantum abstractions to code and reason about problems. Both current abstractions and tools are devised for the qubit level [10]. For example, platforms such as Qiskit [6] provide the means for developers to create quantum circuits using Python. On the conceptual level, this is similar to coding with bits and logic gates. Despite the limitations of the current hardware, it is already

possible to create the abstractions that allow regular developers to create their own algorithms.

This article proposes Rhyme, a high-level quantum programming language, that makes quantum programming accessible to software engineers by providing abstractions to develop quantum software in an intuitive manner. We believe this will encourage developers to join the quantum software engineering community, and lead to new quantum applications and algorithms faster than using gate-level languages.

The following sections informally describe the proposed language via simple code snippets, highlighting the main features which differentiate it from the state-of-the-art. For the first version of the language, the compiler should generate code following Open-QASM 2.0 [8], a machine-independent quantum assembly language for gate-based quantum computation. Several major quantum computing platforms have support for OpenQASM, including IBM Quantum [11], Azure Quantum [14], and Amazon Braket [1].

Section 2 explains data types. Section 3 presents basic quantum-related operations. Section 4 describes some properties and shows how to code the Grover algorithm. Section 5 evaluates related work. Section 6 concludes and presents future work.

2 DATA TYPES

The main intuition is to extend classical data types to quantum data types. The basis states of a quantum data type are then equal to the possible values of the corresponding classical data type. Programmers can express superposition states by mapping probability amplitudes to classical data values and further manipulate such quantum data by re-mapping the amplitudes. Such (re-)mapping of the amplitudes can be achieved by writing classical functions using familiar high-level classical data types.

Classical types. Supported classical types are shown in the code example below. Arrays of classical bits are allowed. The bit-lengths of numeric types and the maximum character-length of strings are compiler parameters.

```
bit b = 0;
bit[] bArr; // not yet initialized
int i = 15;
float f = cos(pi / 4); // common math is allowed
complex z = 0.6 + 0.8i;
char c = 'A'; // any ASCII character
string s = "Hello";
bool t = true;
ref r = &i; // holds the address of i
```

Quantum types. The quantum types are extensions of the classical types. On a qubit-based quantum computer, instances of quantum types are stored on as many qubits as the bit-length of their classical counterparts.

For initialization, the `||` operator(s) create equal superposition of compile-time constant classical values, whereas `<qtype>.all()` creates equal superposition of all possible classical values (basis states) of a type.

```
qbit b = 0 || 1;
qbit[] bArr = qbit.zeros(4);
qint i = 15; // basis state
qint j = 1 || 30 || 160;
qint k = qint.all(); // equal superposition of ALL
↪   int values
qfloat f = 2.5 || 3.5;
qcomplex z = (0.6 + 0.8i) || (sqrt(1/2) +
↪   sqrt(1/2)i);
qchar c = 'A' || 'B' || 'X' || 'Y';
qstring s = "Hello" || "World";
qbool tf = true || false;
qref r = &i || &j;
```

Since all quantum values have an encoding in qubits, the compiler can apply conditional qubit-rotation techniques for state preparation to implement the `||` operator at the gate level [15], and Hadamard gates to implement the `all()` method.

Measurement. When assigning a quantum variable to a classical variable, the quantum variable is measured in the classical basis and collapses. The classical variable holds the result.

Below is an example with the following steps: (1) the program initializes the variables, (2) measures a `qref` which references two `qfloat` variables simultaneously, (3) measures the `qfloat` corresponding to the resulting address from the previous step, which is stored in a `ref`, and (4) prints `float y`.

```
qfloat f = 2.5 || 3.5;
qfloat g = 3.14159 || 2.71828;
qref r = &f || &g;

ref x = r; // r collapses, x is either &f or &g
float y = *x; // *x is a qfloat, it collapses, y
↪   holds outcome

print(y);
```

3 COMPUTATION

Quantum types are compatible with two forms of computation: classical computation (arithmetic, modifications...), and quantum computation. Classical operations on quantum types are defined by their classical counterparts. This section explains how three basic common quantum-related operations are defined: phase shifts, interference, and conditional operations.

Adding phase. On every instance of a quantum type, programmers can call the `addPhase(shift,N)` method, where the first argument `shift` is a classical function that calculates a phase shift for each basis state and is expressed as an integer multiple of the angle $\frac{2\pi}{N}$. The second argument, `N`, is a positive integer, a compile-time constant. The example code below brings b from $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$ to $\frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle$.

The compiler can implement the `addPhase` method at the gate level by applying a generic phase kick-back technique [4]. As for the classical code of `shift`, a compilation pass can convert it to an equivalent reversible quantum circuit [12].

```
qbit b = 0 || 1;
b.addPhase(shift, 2);

def shift(bit b) -> bit {
  if (b == 1) {
    return 1;
  } else {
    return 0;
  }
}
```

Creating interference. On any instance of a quantum type, one can call `applyBipartiteInterference(split,pair)`, which makes the classical basis states pairwise interfere with each other. The first argument, `split`, is a classical function that divides the set of all possible classical values (basis states) of the quantum type into two disjoint subsets, marked by the return value `true` or `false`. The second argument `pair` is a classical bijective map between the "true" and "false" subsets.[1]

The interference caused is analogous to that of a Hadamard gate, H, for qubits: if a classical basis state, $|x\rangle$, in the "true" subset has amplitude a_x, while its pair $|y\rangle$ in the "false" subset has amplitude a_y, then $|x\rangle$ will end up with amplitude $\frac{1}{\sqrt{2}}(a_x + a_y)$, while $|y\rangle$ with amplitude $\frac{1}{\sqrt{2}}(a_x - a_y)$.

In the example code below, `applyBipartiteInterference` brings i from $a|80\rangle + b|81\rangle + c|144\rangle + d|145\rangle$ to $\frac{1}{\sqrt{2}}(a+b)|80\rangle + \frac{1}{\sqrt{2}}(a-b)|81\rangle + \frac{1}{\sqrt{2}}(c+d)|144\rangle + \frac{1}{\sqrt{2}}(c-d)|145\rangle$, so $|80\rangle$ interfered with $|81\rangle$, and $|144\rangle$ with $|145\rangle$.

```
qint i = 80 || 81 || 144 || 145;
...
// code that makes i = a|80>+b|81>+c|144>+d|145>
...
i.applyBipartiteInterference(split, pair);

def split(int i) -> bool {
  if (i % 2 == 0) {
    return true;
```

---

[1]For convenience, it is also allowed that `pair` maps a value to itself, which causes the amplitude of the corresponding basis state to stay the same. If `split` and `pair` do not make up a one-to-one bipartition on the (sub)set of values not mapped to themselves, the behavior of `applyBipartiteInterference` is compiler-dependent.

```
    } else {
      return false;
    }
}

def pair(int i) -> int {
  if (i % 2 == 0) {
    return i + 1;
  } else {
    return i - 1;
  }
}
```

A more general interference, analogous to that of an arbitrary 1-qubit quantum gate, $U = \begin{bmatrix} u_{11} & u_{12} \\ u_{21} & u_{22} \end{bmatrix}$, can be achieved by adding the elements of U as parameters (see code snippet below). In the previous example, this variant would have brought i to $(u_{11}a + u_{12}b)|80\rangle + (u_{21}a + u_{22}b)|81\rangle + (u_{11}c + u_{12}d)|144\rangle + (u_{21}c + u_{22}d)|145\rangle$.[2]

```
i.applyBipartiteInterference(split, pair, u11,
↪  u12, u21, u22);
```

The compiler can implement applyBipartiteInterference at the gate level by making use of the fact that split and pair create a one-to-one bipartition. This bipartition pairs basis states, which can be transformed to each other in a straightforward way, on classical "branches" of a superposition state.

Conditional operation. Programmers can apply a classical reversible operation on a classical value (basis state) of a quantum variable, conditioned on a (classical) logical expression evaluated on the classical values (in superposition) of another quantum variable. The resulting effect of the operation is the creation of entanglement between the two quantum variables.

The code snippet below produces entanglement between two qchar variables. The if (condition) {...} block brings the composite quantum state of c and t from the product state $\left(\frac{1}{\sqrt{2}}|'A'\rangle + \frac{1}{\sqrt{2}}|'B'\rangle\right) \otimes |'A'\rangle$ to the entangled state $\frac{1}{\sqrt{2}}|'A'\rangle|'A'\rangle + \frac{1}{\sqrt{2}}|'B'\rangle|'B'\rangle$.

```
qchar c = 'A' || 'B';
qchar t = 'A';

if (c == 'B') {
  t.increment(); // 'A' to 'B', t becomes entangled
  ↪  with c
}
```

The compiler stores the result of the logical expression in an ancilla qubit. Conditioned on the ancilla state, the compiler first

---

[2]We note that the same interference can be reproduced (approximately, up to the precision allowed by the int data type) by using only $U = H$ and addPhase steps.

executes the body of the if statement, and then the uncomputation of the ancilla qubit.

4 EVALUATION

Universality. The quantum data types are represented as qudits, typically representing quantum systems with more than two dimensions. For example, since char is encoded with 7-bit ASCII characters, a qchar is a qudit of dimension $d = 128$. Applying addPhase and applyBipartiteInterference can reproduce any single-qudit unitary transformation [13].[3] Together with the if statement, which can create entanglement between single qudits, the programming language is universal with respect to any given quantum data type [3].

Example: Grover algorithm with strings. Similarly to classical programming, in certain situations it is useful to write a subroutine using low-level (assembly) instructions to achieve a highly optimized piece of code. To demonstrate this point, below is the code for a simple Grover search, where the goal is to find the string "ABC", marked by the oracle(string s) function.

```
qstring s = qstring.all();

int dim = qstring.dimension();
int numIter = ceil(pi / 4 * sqrt(dim));

// Grover iteration
for(int i = 0; i < numIter; i++) {
  s.applyOracle(oracle);
  s.invertAboutMean();
}

string res = s; // measure to get solution with
↪  high probability
print(res);

def oracle(string s) -> bool {
  if (s == "ABC") {
    return true;
  } else {
    return false;
  }
}
```

One option to implement the invertAboutMean() method, which is available for any quantum type, is to convert the quantum variable to qbit[] and call a generic library method written for qubit arrays using quantum gates:

```
def invertAboutMean(qbit[] arr) {
  int L = arr.length;
  qbit[] anc = qbit.zeros(L);
  for(int i = 0; i < L; i++) {
    arr[i].H();
    arr[i].X();
```

---

[3]Up to the precision allowed by the numeric data types and the hardware.

```
  }
  anc[0].CNOT(arr[0]);
  for(int i = 1; i < L; i++) {
    anc[i].CCNOT(arr[i], anc[i-1]);
  }
  anc[L-1].Z();
  for(int i = L - 1; i > 0; i--) {
    anc[i].CCNOT(arr[i], anc[i-1]);
  }
  anc[0].CNOT(arr[0]);
  for(int i = 0; i < L; i++) {
    arr[i].X();
    arr[i].H();
  }
}
```

Another possibility is not to expose the gate-level implementation at all, declaring the function as `native`, as shown below for the `qstring` type. In general, hiding the quantum gates forces the programmer to think about the impact of quantum operations using high-level classical data structures. For example, binary trees can be used to capture the overall impact of Hadamard gates applied in parallel at the qubit level.

```
def native invertAboutMean(qstring s);
```

Finally, a third approach consists in applying `addPhase`, `applyBipartiteInterference` and `if` statements in the body of `invertAboutMean()`, instead of low-level gates. This way, each quantum type would need a bespoke implementation.

5 RELATED WORK

In today's commonly used quantum programming languages, high-level abstraction features help developers express quantum algorithms operating on qubits using quantum gates. Such features include classical control flow, gate modifiers to automatically generate inverse and controlled versions of unitary subroutines, as well as automatic uncomputation facilities [2, 6, 7, 16, 17]. Other approaches include synthesizing large quantum circuits from high-level (template) functional models [18] or the parsing of classical code to automatically detect quantum speedups and replace the relevant parts with quantum circuits [5].

Regarding data types, several quantum programming languages introduce the concept of "quantum integers". In Silq [2], a quantum integer is an array of qubits and it is manipulated directly at the qubit level. In Quipper [9], an arithmetic library defines a fixed-size quantum integer type called QDInt, that supports classical arithmetic operations, but without QDInt-level operations to holistically manipulate QDInt instances in a universal manner. There is also a similar library that introduces the FPReal type. There are currently no genuine quantum data types other than qubits and arrays of qubits, and programming mostly consists of gate-level manipulations.

6 CONCLUSIONS

The lack of high-level abstractions for quantum programs is one of the major issues for software developers who want to write quantum code. It hinders developers' ability to produce new quantum algorithms.

This article proposes Rhyme, a quantum programming language that includes high-level quantum types based on familiar classical numeric and text data types. In that language, the amplitudes of quantum superposition states can be manipulated by writing solely classical functions. We presented results in the form of code snippets. We plan to develop a full-fledged programming language based on the ideas presented here, hosted on https://quantumpl.org.

REFERENCES

[1] Amazon. 2019. Amazon Braket - AWS. Retrieved Dec 7, 2023 from https://aws.amazon.com/braket/
[2] B. Bichsel, M. Baader, T. Gehr, and M. Vechev. 2020. Silq: a high-level quantum language with safe uncomputation and intuitive semantics. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. Association for Computing Machinery, London, UK, 286–300. https://doi.org/10.1145/3385412.3386007
[3] J.-L. Brylinski and R. Brylinski. 2002. *Mathematics of Quantum Computation*. Chapman & Hall/CRC, Boca Raton, FL, USA, Chapter 4: Universal quantum gates, 101–116. https://mathscinet.ams.org/mathscinet/relay-station?mr=2007944
[4] R. Cleve, A. Ekert, C. Macchiavello, and M. Mosca. 1998. Quantum algorithms revisited. In *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, Vol. 454. The Royal Society, UK, 339–354. https://doi.org/10.1098/rspa.1998.0164
[5] Horizon Quantum Computing. 2023. Pioneering classical to quantum programming. Retrieved Dec 7, 2023 from https://www.horizonquantum.com/technology
[6] A. Cross. 2018. The IBM Q experience and QISKit open-source quantum computing software. In *APS March Meeting Abstracts*, Vol. 2018. American Physical Society, College Park, MD, USA, L58.003. http://meetings.aps.org/link/BAPS.2018.MAR.L58.3
[7] A. Cross, A. Javadi-Abhari, T. Alexander, N. De Beaudrap, L. S. Bishop, S. Heidel, C. A. Ryan, P. Sivarajah, J. Smolin, J. M. Gambetta, and B. R. Johnson. 2022. OpenQASM 3: A Broader and Deeper Quantum Assembly Language. *ACM Trans. Quant. Computing* 3, 3 (2022), 1–50. https://doi.org/10.1145/3505636
[8] A. W. Cross, L. S. Bishop, J. A. Smolin, and J. M. Gambetta. 2017. Open Quantum Assembly Language. arXiv:1707.03429 [quant-ph]
[9] A. S. Green, P. L. Lumsdaine, N. J. Ross, P. Selinger, and B. Valiron. 2013. Quipper: A Scalable Quantum Programming Language. *SIGPLAN Not.* 48, 6 (2013), 333–342. https://doi.org/10.1145/2499370.2462177
[10] B. Heim, M. Soeken, S. Marshall, C. Granade, M. Roetteler, A. Geller, M. Troyer, and K. Svore. 2020. Quantum programming languages. *Nature Reviews Physics* 2, 12 (2020), 709–722. https://doi.org/10.1038/s42254-020-00245-7
[11] IBM. 2016. IBM Quantum. Retrieved Dec 7, 2023 from https://quantum.ibm.com
[12] A. Litteken, Y.-C. Fan, D. Singh, M. Martonosi, and F. T. Chong. 2020. An updated LLVM-based quantum research compiler with further OpenQASM support. *Quant. Sci. and Tech.* 5, 3 (2020), 034013. https://doi.org/10.1088/2058-9565/ab8c2c
[13] M. Luo and X. Wang. 2014. Universal quantum computation with qudits. *Science China Physics, Mechanics & Astronomy* 57, 9 (2014), 1712–1717. https://doi.org/10.1007/s11433-014-5551-9
[14] Microsoft. 2020. Azure Quantum cloud service. Retrieved Dec 7, 2023 from https://azure.microsoft.com/en-gb/products/quantum/
[15] F. Mozafari, H. Riener, M. Soeken, and G. De Micheli. 2021. Efficient Boolean Methods for Preparing Uniform Quantum States. *IEEE Transactions on Quantum Engineering* 2 (2021), 1–12. https://doi.org/10.1109/TQE.2021.3101663
[16] R. Seidel, N. Tcholtchev, S. Bock, and M. Hauswirth. 2023. Uncomputation in the Qrisp High-Level Quantum Programming Framework. In *Reversible Computation*. Springer Nature Switzerland, Cham, 150–165. https://doi.org/10.1007/978-3-031-38100-3_11
[17] K. Svore, A. Geller, M. Troyer, J. Azariah, C. Granade, B. Heim, V. Kliuchnikov, M. Mykhailova, A. Paz, and M. Roetteler. 2018. Q#: Enabling Scalable Quantum Computing and Development with a High-Level DSL. In *Proceedings of the Real World Domain Specific Languages Workshop 2018*. Association for Computing Machinery, New York, NY, USA, Article 7, 10 pages. https://doi.org/10.1145/3183895.3183901
[18] Classiq Technologies. 2021. The Classiq Platform. Retrieved Dec 7, 2023 from https://platform.classiq.io