УДК 519.62 DOi 10.30837/ bi.2020.2(95).02

Ivan Bozhko[1], Grygoriy Chetverykov[2], Oleksandr Karataiev[3]

[1] Khavkiv National University of Radio Electronics, Kharkiv, Ukraine,
ivan.bozhko@nure.ua

[2]Khavkiv National University of Radio Electronics, Kharkiv, Ukraine,
grigorij.chetverykov@nure.ua

[3]Khavkiv National University of Radio Electronics, Kharkiv, Ukraine,
oleksandr.karataiev@nure.ua

THE MODERN QUANTUM COMPUTING TOOLS INVSTIGATION

The current paper covers the investigation of the current state of the existing tools for quantum programming including QCL (Quantum Computation Language), quantum pseudocode, Q# programming language and Quipper. Since quantum computing is one of the main research areas today, the respective tools are being created quite often. They are aimed on simplifying the development of quantum programs, on the one hand, and provide some platform for testing and running them, on the other hand. So, the authors investigated the currently available tools and provided the results in the article.

QUANTUM COMPUTING, QUANTUM COMPUTER, Q#, QUANTUM COMPUTING TOOLS, MICROSOFT QUANTUM DEVELOPMENT KIT

Божко І.К., Четвериков Г.Г., Каратаєв О.А. Дослідження сучасних засобів квантових обчислень. Дана робота являє собою дослідження поточного стану існуючих інструментів квантових обчислень, особливо мови програмування Q# як найбільш розвиненого інструменту для цього в даний час. Оскільки квантові обчислення сьогодні є однією з основних областей досліджень, створюються відповідні інструменти. Вони покликані спростити розробку квантових програм, з одного боку, і надати платформу для тестування і запуску їх, з іншого боку. Тому автори дослідили наявні в даний час інструменти і представили результати в даній статті.

КВАНТОВІ ОБЧИСЛЕННЯ, КВАНТОВИЙ КОМП'ЮТЕР, Q#, ІНСТРУМЕНТИ КВАНТОВИХ ОБЧИСЛЕНЬ, MICROSOFT QUANTUM DEVELOPMENT KIT

Божко И.К., Четвериков Г.Г., Каратаев А.А. Исследование современных средств квантовых вычислений. Данная работа представляет исследование текущего состояния существующих инструментов квантовых вычислений, особенно языка программирования Q# как самого развитого инструмента для этого в настоящее время. Поскольку квантовые вычисления сегодня являются одной из основных областей исследований, создаются соответствующие инструменты. Они призваны упростить разработку квантовых программ, с одной стороны, и предоставить платформу для тестирования и запуска их, с другой стороны. Поэтому авторы исследовали имеющиеся в настоящее время инструменты и представили результаты в данной статье.

КВАНТОВЫЕ ВЫЧИСЛЕНИЯ, КВАНТОВЫЙ КОМПЮТЕР, Q#, ИНСТРУМЕНТЫ КВАНТОВЫХ ВЫЧИСЛЕНИЙ, MICROSOFT QUANTUM DEVELOPMENT KIT

Introduction

The research of the quantum computer area started for the first time back in 1980 by the Soviet mathematician Yu.I. Manin [1]. However, more interest in this type of calculation arose only in 1982, after the American theoretical physicist Richard Feynman noticed that not all quantum-mechanical operations can be accurately transferred to a classical computer and more efficiently carried out by quantum operations.

An additional relevance to the quantum computing problem was added by mathematician Peter Shore, who in 1994 proposed an algorithm that allows the expansion of a n-valued number to simple multipliers with polynomial complexity. On classical computers, this task is much more complex and does not allow you to get the result for a satisfactory time.

Since this task is the basis of many popular cryptographic algorithms (for example, RSA) [2], the creation of quantum computers may influence the security of the data exchange in the network, and after the appearance of a real prototype of a quantum computer, it can become a global security issue.

Consequently, quantum computing is relevant not only for scientific problems of quantum processes modeling, but also relevant to the world of information technology. In order to popularize them not only in the scientific world, but also among developers, tools are created to simplify the work with quantum algorithms, such as the programming language Q#, which is considered in this paper.

1. Existing quantum computing tools

Over the past few years, with the growing popularity of quantum computing research, tools and emulators of quantum computers have begun to appear that allow you to try calculations to practice. Here are some popular tools and their descriptions.

1.1. Microsoft Quantum Development Kit

Microsoft has released a preview version of Quantum Development Kit, which includes the new quantum programming language Q#, integration with the Visual Studio development environment, simulators that work with both the local system and their powerful Azure cloud platform, as well as libraries and code samples that can be used as constructive blocks.

1.2. IBM Quantum Experience

IBM has created an experimental quantum 5-qubit processor that is available to users through the Internet [2]. On the IBM Quantum Experience website, you can find a short tutorial that explains the basics of quantum computing and system usage instructions, the configuration of the queue access interfaces, a simulator that allows you to simulate their configuration before running it on the actual machine, and access to the machine itself that allows you to run the configuration and view the results.

1.3. Rigetti Forest

The Rigetti Forest package consists of the instructions based quantum language called Quil, an open Python library for building Quil programs called pyQuil, a quantum library called Grove, and a simulation environment called QVM (Quantum Virtual Machine). pyQuil and Grove are open source programs available on Github. Users can develop their applications using pyQuil and Grove on their own computer, and then transfer them to QVM for simulation through a web portal that is available to registered users.

1.4. ProjectQ

ProjectQ is an open source software for quantum computing, implemented in Python. This allows users to implement their quantum programs in Python using a powerful and intuitive syntax. ProjectQ can then broadcast these programs to any server part: a simulator that runs on a classic computer, or a quantum computer (for example, using IBM Quantum Experience). Other hardware platforms are currently not supported.

In addition to these tools, there are others such as Cirq, Quirk, QuTiP, but they are less powerful than those described above.

As can be seen from the description of existing solutions, most of them are complementary to existing programming languages (in particular, Python), but given the new paradigm of computations, this may cause difficulties in programming algorithms using these tools, therefore, a more flexible and powerful solution is a separate programming language, which currently only Microsoft offers.

Also, it is necessary to highlight the Intel solution with the existing experimental quantum computer, but it requires execution of queries to individual quibits through the API, which complicates its use.

The short comparison could be seen in the table 1.1.

Table 1.1

The comparison of the quantum computing tools

Feature\ Technology	**IBM Q Exp.**	**Rigetti Forest**	**ProjectQ**	**Q#**
Language	-	-	-	+
API Access	+	+	-	+
Integration with other languages	-	+	+	+
Availability for different platforms	+	-	-	-
Possibility to use real computer instead of emulators	+	-	-	-

2. The Q# languauge

As discussed above, one of the parts of Microsoft Quantum Development Kit is a Q# language specifically designed for quantum computing. In terms of software engineering, this solution is most interesting as it allows abstracting from the paradigm of classical computing and classical programming languages and describing a quantum algorithm using a special syntax.

Consider the language Q # in more detail.

2.1. Computing model

According to official Microsoft documentation [4], a natural model for quantum computation is to treat the quantum computer as a coprocessor, similar to that used for GPUs, FPGAs, and other adjunct processors. The primary control logic runs classical code on a classical "host" computer. When appropriate and necessary, the host program can invoke a sub-program that runs on the adjunct processor. When the sub-program completes, the host program gets access to the sub-program's results.

In this model, there are three levels of computation:

- Classical computation that reads input data, sets up the quantum computation, triggers the quantum computation, processes the results of the computation, and presents the results to the user.
- Quantum computation that happens directly in the quantum device and implements a quantum algorithm.
- Classical computation that is required by the quantum algorithm during its execution.

There is no intrinsic requirement that these three levels all be written in the same language. Indeed, quantum computation has somewhat different control structures and resource management needs than classical computation, so using a custom programming language allows common patterns in quantum algorithms to be expressed more naturally.

Keeping classical computations separate means that the quantum programming language may be very constrained. These constraints may allow better optimization or faster execution of the quantum algorithm.

Q# (Q-sharp) is a domain-specific programming language used for expressing quantum algorithms. It is to be used for writing sub-programs that execute on an adjunct quantum processor, under the control of a classical host program and computer.

Q# provides a small set of primitive types, along with two ways (arrays and tuples) for creating new, structured types. It supports a basic procedural model for writing programs, with loops and if/then statements. The top-level constructs in Q# are user defined types, operations, and functions.

2.2. Q# type system

The Q# language provides a small set of primitive types, as well as two methods (arrays and corrections) for creating new types of data (it means that the language have a bit more constraints than classical languages, which was made for optimization purposes). In general, the language usually supports a procedural conditional programming model (if -this) and cycles.

Let us consider the primitive data types which other types consist of:

- The int type represents a 64- bit signed (two's complement) integer.
- The double type represents a double-precision floating-point number.
- The bool type represents a Boolean value, either true or false.
- The qubit type represents a quantum bit or qubit. They are opaque to the user; the only operation possible with them, other than passing them to another operation, is to test for identity (equality). Ultimately, actions on Qubits are implemented by calling operations in the Q# standard library.
- The Pauli type represents an element of the single-qubit Pauli group. This type is used to denote the base operation for rotations and to specify the basis of a measurement. This type is a discriminated union with four possible values: PauliI, PauliX, PauliY and PauliZ.
- The Result type represents the result of a measurement. This type is a discriminated union with two possible values: One and Zero. Zero indicates that the +1 eigenvalue was measured; One indicates the -1 eigenvalue.
- The Range type represents a sequence of integers.
- The String type is a sequence of Unicode characters that is opaque to the user once created. This type is used to report messages to a classical host.

It is also necessary to note that having the type system described above means a set of reserved keywords: true, false, PauliI, PauliX, PauliY, PauliZ, Zero and One.

Outside of the primitive types, there are also another types we will consider now in more detail.

Given any valid Q# type ‘T there is a type that represents an array of values of type ‘T. This array type is represented as ‘T[] for example, Qubit[] or Int[][].

In the second example, note that this represents a potentially jagged array of arrays, and not a rectangular two-dimensional array. Q# does not include support for rectangular multi-dimensional arrays.

Given any valid Q# types ‘T1, ‘T2, ‘T3, etc., there is a type that represents a tuple of values of types 'T1, 'T2, 'T3, etc., respectively. This tuple type is represented as ('T1, 'T2, 'T3, ...). Any number of types may be tupled together. The empty tuple, (), is equivalent to unit in F#.

It is possible to create arrays of tuples, tuples of arrays, tuples of sub-tuples, etc.

Tuple instances are immutable. Q# does not provide a mechanism to change the contents of a tuple once created.

It is also possible to create a singleton (single-element) tuple, ('T1), such as (5) or ([1;2;3]). However, Q# treats a singleton tuple as completely equivalent to a value of the enclosed type. That is, there is no difference between 5 and (5), or between 5 and (((5))), or between (5, (6)) and (5, 6).

This equivalence applies for all purposes, including assignment and expressions. It is just as valid to write (5)+3as to write 5+3, and both expressions will evaluate to 8. We refer to this property as singleton tuple equivalence.

There is also a possibility for creating user-defined types. A Q# file may define a new named type based on a standard type. Any legal type may be used as the base for a user-defined type.

User-defined types may be used anywhere any other type may be used. In particular, it is possible to define an array of a user-defined type and to include a user-defined type as an element of a tuple type.

It is not possible to create recursive type structures. That is, the type that defines a user-defined type may not be a tuple type that includes an element of the user-defined type. More generally, user-defined types may not have cyclic dependencies on each other.

The mutability of instances of user-defined types is the same as the mutability of instances of the base type of the user-defined type. Specifically, instances of user-defined types based on tuples are immutable; instances of user-defined types based on arrays are potentially mutable.

Effectively, a user-defined type is a subtype of the base type. Thus, a value of a user-defined type may be used anywhere a value of the base type is expected. This is applied recursively.

For example, suppose type IntPair is a user-defined type with base type (Int, Int), and type IntPair2 is a user-defined type with base type IntPair. A value of type IntPair2 may be used anywhere a value of type IntPair2, IntPair, or (Int, Int) is expected. A value of type IntPair may be used anywhere a value of type IntPair or (Int, Int) is expected.

Different user-defined types based on the same base type are treated as distinct and unrelated types. In the previous example, if IntPair3 is also a user-defined type with base type (Int, Int), then IntPair and IntPair3 are unrelated and a value of one may not be used where a value of the other is expected.

A Q# operation is a quantum subroutine. That is, it is a callable routine that contains quantum operations.

A Q# function is a classical subroutine used within a quantum algorithm. It may contain classical code but no quantum operations. Functions may not allocate or borrow qubits, nor may they call operations. It is possible, however, to pass them operations or qubits for processing.

Together, operations and functions are known as callables.

All Q# callables are considered to take a single value as input and return a single value as output. Both the input and output values may be tuples. Callables that have no result return the empty tuple, (); callables that have no input take the empty tuple as input.

The basic signature for any callable is written as ('Tinput => 'Tresult) or ('Tinput -> 'Tresult), where both 'Tinput and 'Tresult are type specifiers. The first form, with =>, is used for operations; the second form, with ->, for functions. For example, ((Qubit, Pauli) => Result) represents the signature for a possible single-qubit measurement operation.

Function types are completely specified by their signature. For example, a function that computes the sine of an angle would have type (Double -> Double).

Operations – but not functions – may allow the application of one or more functors. Functors are meta-operations that generate a variant of a base operation.

Operation types are specified by the their signature and the list of functors they support. For example, the Pauli X operation has type (Qubit => () : Adjoint, Controlled). An operation type that does not support any functors is specified by its signature alone, with no trailing :.

Callable signatures may contain type parameters. Type parameters are indicated by a symbol prefixed by a single quote; for example, 'A is a legal type parameter. Type-parameterized functions and operations are similar to generic functions in many programming languages, but Q# does not provide a full generic type/function capability.

A type parameter may appear more than once in a single signature. For example, a function that applies another function to each element of an array and returns the collected results would have signature(('A[], 'A->'A) -> 'A[]). Similarly, a function that returns the composition of two operations might have signature ((('A=>'B), ('B=>'C)) -> ('A=>'C)).

When invoking a type-parameterized callable, all arguments that have the same type parameter must be of the same type, or be compatible with the same type; that is.

Q# does not provide a mechanism for constraining the possible types that might be substituted for a type parameter. Thus, type parameters are primarily useful for functions on arrays and for composing callables.

An operation with additional functors supported may be used anywhere an operation with fewer functors but the same signature is expected. For instance, an operation of type (Qubit=>():Adjoint) may be used anywhere an operation of type (Qubit=>()) is expected.

Q# is covariant with respect to callable return types: a callable that returns a type 'A is compatible with a callable with the same input type and a result type that 'A is compatible with.

Q# is contravariant with respect to input types: a callable that takes a type 'A as input is compatible with a callable with the same result type and an input type that is compatible with 'A.

A functor in Q# is a factory that defines a new operation from another operation. Functors have access to the implementation of the base operation when defining the implementation of the new operation. Thus, functors can perform more complex functions than traditional higher-level functions.

A functor is used by applying it to an operation, returning a new operation. For example, the operation that results from applying the Adjoint functor to the Y operation is written as (Adjoint Y). The new operation may then be invoked like any other operation. Thus, (Adjoint Y)(q1) applies the adjoint functor to the Yoperation to generate a new operation, and applies that new operation to q1. Similarly, (Controlled X)(controls, target).

The two standard functors in Q# are Adjoint and Controlled.

In quantum computing, the adjoint of an operation is the complex conjugate transpose of the operation. For operations that implement a unitary operator, the adjoint is the inverse of the operation. For a simple operation that just invokes a sequence of other unitary operations on a set of qubits, the adjoint may be computed by applying the adjoints of the sub-operations on the same qubits, in the reverse sequence.

Given an operation expression, a new operation expression may be formed using the Adjoint functor, with the base operation expression enclosed in parentheses, (and). The new operation has the same signature and type as the base operation. In particular, the new operation also allows Adjoint, and will allow Controlled if and only if the base operation did.

For instance, (Adjoint QFT) designates the adjoint of the QFT operation.

The controlled version of an operation is a new operation that effectively applies the base operation only if all of the control qubits are in a specified state. If the control qubits are in superposition, then the base operation is applied coherently to the appropriate part of the superposition. Thus, controlled operations are often used to generate entanglement.

In Q#, controlled versions always take an array of control qubits, and the specified state is always for all of the control qubits to be in the computational (PauliZ) One state, |1>|1>. Controlling based on other states may be achieved by applying the appropriate Clifford operations to the control qubits before the controlled operation, and then applying the inverses of the Cliffords after the controlled operation. For example, applying an Xoperation to a control qubit before and after a controlled operation will cause the operation to control on the Zero state (|0>|0>) for that qubit; applying an H operation will control on the PauliX Zero state |+>:=(|0>+|1>)/√2|+>:=(|0>+|1>)/2 rather than the PauliZ Zero state.

Given an operation expression, a new operation expression may be formed using the Controlled functor, with the base operation expression enclosed in parentheses, (and). The signature of the new operation is based on the signature of the base operation. The result type is the same, but the input type is a two-tuple with a qubit array that holds the control qubit(s) as the first element and the arguments of the base operation as the second element. If the base operation took no arguments, (), then the input type of the controlled version is just the array of control qubits. The new operation allows Controlled, and will allow Adjoint if and only if the base operation did.

If the base function took only a single argument, then singleton tuple equivalence will come into play here. For instance, Controlled(X) is the controlled version of the X operation. X has type (Qubit => () : Adjoint, Controlled), so Controlled(X) has type ((Qubit[], (Qubit)) => () : Adjoint, Controlled); because of singleton tuple equivalence, this is the same as ((Qubit[], Qubit) => () : Adjoint, Controlled).

Similarly, Controlled(Rz) is the controlled version of the Rz operation. Rz has type ((Double, Qubit) => () : Adjoint, Controlled), so Controlled(Rz) has type((Qubit[], (Double, Qubit)) => () : Adjoint, Controlled). For example, ((Controlled(Rz))(controls, (0.1, target)) would be a valid invocation of Controlled(Rz).

As another example, CNOT(control, target) can be implemented as (Controlled(X))([control], target). If a target should be controlled by 2 control qubits (CCNOT), we can use (Controlled(X))([control1;control2], target) statement.

2.3. Q# expressions

Expressions is an important part of any programming language and Q# is not an exception. Let us consider some of the important expressions.

Given any expression, that same expression enclosed in parentheses is an expression of the same type. For instance, (7) is an Int expression, ([1;2;3]) is an expression of type array of Ints, and ((1,2)) is an expression with type (Int, Int).

The equivalence between simple values and single-element tuples removes the ambiguity between (6) as a group and (6) as a single-element tuple.

The name of a symbol bound or assigned to a value of type 'T is an expression of type 'T. For instance, if the symbol count is bound to the integer value 5, then count is an integer expression.

Numeric expressions are expressions of type Int or Double. That is, they are either integer or floating-point numbers.

Int literals in Q# are identical to integer literals in C#, except that no trailing "l" or "L" is required (or allowed). Hexadecimal integers are supported with a "0x" prefix.

Double literals in Q# are identical to double literals in C#, except that no trailing "d" or "D" is required (or allowed).

Given an array expression of any element type, an Int expression may be formed using the Length built-in function, with the array expression enclosed in parentheses, (and). For instance, if a is bound to an array, then Length(a) is an integer expression. If b is an array of arrays of integers, Int[][], then Length(b) is the number of sub-arrays in b, and Length(b[1]) is the number of integers in the second sub-array in b.

Given two numeric expressions, the binary operators +, -, *, and / may be used to form a new numeric expression. The type of the new expression will be Double if both of the constituent expressions are Double, or will be an Int expression if both are integers.

Given two integer expressions, a new integer expression may be formed using the % (modulus), ^ (power), &&& (bitwise AND), ||| (bitwise OR), ^^^ (bitwise XOR), <<< (arithmetic left shift), or >>> (arithmetic right shift) operations. The second parameter to either shift operation must be greater than or equal to zero. The behavior for shifting negative numbers is undefined.

Given any numeric expression, a new expression may be formed using the - unary operator. The new expression will be the same type as the constituent expression.

Given any integer expression, a new integer expression may be formed using the ~~~ (bitwise complement) unary operator.

The only Qubit expressions are symbols that are bound to Qubit values or array elements of Qubit arrays. There are no Qubit literals.

The four Pauli values, PauliI, PauliX, PauliY, and PauliZ, are all valid Pauli expressions.

Other than that, the only Pauli expressions are symbols that are bound to Pauli values or array elements of Pauli arrays.

The two Result values, One and Zero, are valid Result expressions.

Other than that, the only Result expressions are symbols that are bound to Result values or array elements of Result arrays. In particular, note that One is not the same as the integer 1, and there is no direct conversion between them. The same is true for Zero and 0.

Given any three Int expressions start, step, and stop, start .. step .. stop is a range expression whose first element is start, second element is start+step, third element is start+step+step, etc., until stopis passed. A range may be empty if, for instance, step is positive and stop < start. The last element of the range will be stop if the difference between start and stop is an integral multiple of step; that is, the range is inclusive at both ends.

Given any two Int expressions start and stop, start .. stop is a range expression that is equal to start .. 1 .. stop. Note that the implied step is +1 even if stop is less than start; in such a case, the range is empty. For example, 1..3 is the range 1, 2, 3.

A callable literal is the name of an operation or function defined in the compilation scope. For instance, X is an operation literal that refers to the standard library X operation, and Message is a function literal that refers to the standard library Message function.

If an operation supports the Adjoint functor, then (Adjoint op) is an operation expression. Similarly, if the operation supports the Controlled functor, then (Controlled op) is an operation expression.

Q# callables are allowed to be directly or indirectly recursive. That is, an operation or function may call itself, or it may call another callable that directly or indirectly calls the callable operation.

3. A quantum program creating example (Q#)

In order to illustrate the possibilities of quantum computing via Q# language we will create a simple program in Visual Studio. As a precondition, Microsoft Quantum Development Kit should be installed on the computer.

Firstly, we should open Visual Studio 2017 and create a solution of the type "Q# Application". Afterwards, optionally we should update the necessary NuGet packages. Now, it is possible to create a program.

Consider a simple program of setting a qubit state (see Fig. 1):

```
test.qs   ×   Extension: Microsoft Quantum Development Kit for Visual Studio Code
test.qs
1   open Microsoft.Quantum.Canon;
2   open Microsoft.Quantum.Primitive;
3
4   operation Set (desired: Result, q1: Qubit) : ()
5   {
6       body
7       {
8           let curr = M(q1);
9           if (desired != curr)
10          {
11              X(q1);
12          }
13      }
14  }
```

Fig. 1. The qubit state setting function

As it can be seen from the code, the standard tools allow us to do everything via the standard library. Thus, we include the necessary standard tools on the first and the second lines and create a Set function accepting two parameters of Result and Qubit types respectively. Then, we get the current qubit state via M() function and compare the current state with the desired one. If they are not equal, we set the necessary state.

It should be clear from this example that having such a dedicated tool for a special computing paradigm as a language simplifies programming dramatically, since using other tools (e.g. calling an API) would require much more code.

Conclusion

In scope of the article the basic tools of quantum computing have been investigated. . It included different languages (imperative (QCL), functional (Quipper), multi-paradigm (Q#)), quantum pseudocode and frameworks used with the existing languages (e.g. ProjectQ for Python).

According to the retrieved data, the accurate performance cannot be determined using the given tools because of the absence of a fully functional quantum computer prototype (the results obtained for emulators cannot be considered precise). The tools cover most of the basic quantum computing units including qubits and operations with them, but differ in terms of the used syntax and the standard library.

References

[1] *Nielsen, M. A., Chuang, I. L.* Quantum Computation and Quantum Information. Quantum Computation and Quantum Information. UK: Cambridge University Press. 2010.

[2] *Simon J Gay.* Quantum programming languages: Survey and bibliography. Mathematical Structures in Computer Science. 2006. № 16 (04). P. 581-600.

[3] *C. C. Moran.* Quintuple: A tool for introducing quantum computing into the classroom. / Christine C. Moran – Frontiers in Physics 6(JUL). – p. 69

[4] *Bernhard Ömer.* Quantum Programming in QCL. Master thesis computing science. TU Vienna. 2000.

[5] *Emmanuel Knill.* Conventions for quantum pseudocode. Technical Report. Los Alamos National Lab., NM (United States). 1996.

[6] Q#: Enabling scalable quantum computing and development with a high-level domain-specific language. arXiv:1803.00652 [quant-ph], 2018.

[7] Quipper: A Scalable Quantum Programming Language. Alexander S. Green and others. ACM SIGPLAN Notices 48(6). P. 333-342. 2013

The article was delivered to editorial stuff on the 20.10.2020