Vol.2022-QS-7 No.19
2022/10/28

Entanglement Analysis of Quantum Programs in Q#

Shangzhou Xia[1,a)] Jianjun Zhao[1,b)]

Abstract: Quantum entanglement plays a crucial role in quantum computing. Entangling information has important implications for understanding the behavior of quantum programs and avoiding entanglement-induced errors. Entanglement analysis is a static code analysis technique that determines which qubit may entangle with another qubit and establishes an entanglement graph to represent the whole picture of interactions between entangled qubits. This paper presents the first static entanglement analysis method for quantum programs developed in the practical quantum programming language Q#. Our method first constructs an interprocedural control flow graph (ICFG) for a Q# program and then calculates the entanglement information not only within each module but also between modules of the program. The analysis results can help improve the reliability and security of quantum programs.

Keywords: Entanglement analysis, quantum programming, Q#

1. Introduction

In recent years, with the development of quantum computers, more and more quantum programming languages and environments have been developed to support programming quantum computers. However, since quantum programming requires exploiting unique quantum properties such as superposition and entanglement, it is more challenging than classical programming, which makes understanding the behavior of quantum programs very difficult. There is an urgent need, therefore, to develop methods and tools to support the analysis of quantum programs efficiently and automatically.

Quantum entanglement plays a crucial role in quantum computing. Entangling information has important implications for understanding the behavior of quantum programs and avoiding entanglement-induced errors. The quantum software at this stage is still in a mixed state of classical and quantum. During the program execution, it is inevitable that some quantum bits in the quantum software need to be measured. Due to the existence of entanglement phenomena, performing measurements without systematic entanglement information may lead to the destruction of the state of other quantum bits in the program as well, which leads to program errors and information loss. In addition, due to the no-cloning principle of quantum computing, ancilla qubits are often used to assist the operation during the program execution. Therefore, it is also one of the goals of entanglement analysis to ensure that the ancilla qubits are no longer entangled with the system after the auxiliary operation is completed.

Entanglement analysis is a static analysis technique that determines which qubit may entangle with another qubit in a quantum program. Several entanglement analysis methods [12, 13, 8, 19, 9, 14] have been proposed to support different types of quantum programming languages, but no entanglement analysis method for supporting the practical quantum programming language Q# has been available until now. As the first step toward the efficient analysis of entanglement information in practical quantum programs, this paper presents the first static entanglement analysis method for quantum programs developed in the practical quantum programming language Q#. Our method first constructs an interprocedural control flow graph (ICFG) for a Q# program and then calculates the entanglement information within each module and between modules of the program. Our method establishes an entanglement graph to represent the whole picture of interactions between entangled qubits in the program. The analysis results can help improve the reliability and security of quantum programs.

The rest of the paper is organized as follows. Section 2 introduces some basic concepts of quantum computation and Q#. Section 3 presents an example to illustrate how the entanglement analysis algorithm works. Section 4 presents the algorithm for entanglement analysis at both intraprocedural and interprocedural levels. Related work is discussed in Section 5, and conclusion is given in Section 6.

2. Backgroud Information

We briefly introduce some basics of quantum computing [11] and the quantum programming language Q#.

2.1 Basic Concepts of Quantum Computation

2.1.1 Quantum Bit

A classical bit is a binary unit of information used in classical computation. It can take two possible values, 0 or 1. A quantum bit (or *qubit*) is different from the classical bit in that its state is theoretically represented by a linear combination of two bases in the quantum state space (represented by a column vector of length 2). We can define two qubits $|0\rangle$ and $|1\rangle$, which can be described as

[1] Kyushu University, Fukuoka, Japan
[a)] xia.shangzhou.218@s.kyushu-u.ac.jp
[b)] zhao@ait.kyushu-u.ac.jp

$$|0\rangle = \begin{bmatrix}1\\0\end{bmatrix} \text{ and } |1\rangle = \begin{bmatrix}0\\1\end{bmatrix}$$

Qubits $|0\rangle$ and $|1\rangle$ are the computational basis state of the qubit. In other words, they are a set of the basis of quantum state space.

Any qubit $|e\rangle$ can be expressed as a linear combination of two bases:

$$|e\rangle = \alpha|0\rangle + \beta|1\rangle$$

where α and β are complex numbers, and $|\alpha|^2 + |\beta|^2 = 1$. This restriction is also called *normalization conditions*.

2.1.2 Quantum Gate and Circuit

Just as a logic gate in a digital circuit can modify the state of a bit, a quantum gate can change the state of a qubit. A quantum gate can have only one input and one output (transition of a single quantum state), or it can have multiple inputs and multiple outputs (transition of multiple quantum states). The number of inputs and outputs should be equal because the operators need to be reversible, which means no information can be lost in quantum computing.

NOT Gate. The NOT gate works on a single qubit, which can exchange the coefficients of two basis vectors: $NOT(\alpha|0\rangle + \beta|1\rangle) = \alpha|1\rangle + \beta|0\rangle$. The quantum NOT gate is an extension of the NOT gate in classical digital circuits.

A single input-output quantum gate can be represented by a 2×2 matrix. The state of a quantum state after passing through the quantum gate is determined by the value of the quantum state vector left multiplied by the quantum gate matrix. The quantum gate matrix corresponding to the NOT gate is

$$X = \begin{bmatrix}0 & 1\\1 & 0\end{bmatrix}.$$

Therefore, the result of a qubit passing a NOT gate can be denoted as

$$X\begin{bmatrix}\alpha\\\beta\end{bmatrix} = \begin{bmatrix}0 & 1\\1 & 0\end{bmatrix}\begin{bmatrix}\alpha\\\beta\end{bmatrix} = \begin{bmatrix}\beta\\\alpha\end{bmatrix}.$$

Hadamard Gate. The Hadamard gate also works on a single qubit, which can decompose existing quantum states according to its coefficients as:

$$H(\alpha|0\rangle + \beta|1\rangle) = \tfrac{\alpha+\beta}{\sqrt{2}}|0\rangle + \tfrac{\alpha-\beta}{\sqrt{2}}|1\rangle.$$

This can be represented by a matrix:

$$H = \tfrac{\sqrt{2}}{2}\begin{bmatrix}1 & 1\\1 & -1\end{bmatrix}.$$

Although the Hadamard gate is not directly related to the AND and OR gates in classical digital circuits, it has important applications in many quantum computing algorithms.

2.1.3 Controlled NOT Gate

Computer programs are full of conditional judgment statements: if so, what to do, otherwise, do something else. In quantum computing, we also expect that the state of one qubit can be changed by another qubit, which requires a quantum gate with multiple inputs and outputs. The following is the controlled-NOT gate (CNOT gate). It has two inputs and two outputs. If the input and output are taken as a whole, this state can be expressed by

$$\alpha|00\rangle + \beta|01\rangle + \gamma|10\rangle + \theta|11\rangle,$$

where $|00\rangle$, $|01\rangle$, $|10\rangle$, $|11\rangle$ are column vectors of length 4, which can be generated by concatenating $|0\rangle$ and $|1\rangle$. This state also needs to satisfy the normalization conditions, that is $|\alpha|^2 + |\beta|^2 + |\gamma|^2 + |\theta|^2 = 1$.

The CNOT gate is a two-qubit operation, where the first qubit is usually referred to as the control qubit and the second qubit as the target qubit. When the control qubit is in state $|0\rangle$, it leaves the target qubit unchanged, and when the control qubit is in state $|1\rangle$, it leaves the control qubit unchanged and performs a Pauli-X gate on the target qubit. It can be expressed in mathematical formulas as

$$CNOT(\alpha|00\rangle + \beta|01\rangle + \gamma|10\rangle + \theta|11\rangle) = \alpha|00\rangle + \beta|01\rangle + \gamma|11\rangle + \theta|10\rangle.$$

The action of the CNOT gate can be represented by the matrix:

$$X = \begin{bmatrix}1 & 0 & 0 & 0\\0 & 1 & 0 & 0\\0 & 0 & 0 & 1\\0 & 0 & 1 & 0\end{bmatrix}$$

2.2 Quantum Entanglement

Quantum systems may exhibit *entanglement* [7, 15], a quantum mechanical phenomenon. A state is considered entangled if it cannot be broken down into more basic parts. The existence of entanglement relations makes mutually independent qubit systems connected, thus enabling the information interaction of different qubit systems. At the same time, the existence of entanglement relations among systems makes it impossible to be considered separable systems. When a measurement is made on some qubits in the system, it affects the state of other qubits.

Definition 1 For a state $|\varphi\rangle$ of a set S of qubits and a partition (A, B) of S, (A, B) is *entangled* iff there does not exist two states $|\varphi_A\rangle$ and $|\varphi_B\rangle$ of the respective parts A and B such that $|\varphi\rangle = |\varphi_A\rangle \otimes |\varphi_B\rangle$. (A, B) is *separable* iff (A, B) is not entangled.

For example, a GHZ state $\frac{\sqrt{2}}{2}(|000\rangle+|111\rangle)$ is entangled. When the result of observing the first qubit is 0, the other qubits must all be 0. Therefore, the measurement of some qubits in the absence of system entanglement information leads to the destruction of the system state in which it is located, triggering bugs and information loss of the program.

For a state $|\varphi\rangle$ of a set S of qubits and a partition (A, B, C) of S. The following are the three properties of the entanglement relation:

- **Transitive**: If (A, B) is entangled, (B, C) is entangled, then (A, C) is entangled.
- **Symmetric**: If (A, B) is entangled, then (B, A) is entangled
- **Eliminable**: If (A, B) is separable and for an operation U that $U(|\varphi_A\rangle \otimes |\varphi_B\rangle)$ is entangled, then there must exist operation V that $VU(|\varphi_A\rangle \otimes |\varphi_B\rangle)$ is separable.

2.3 Quantum Programming and Q#

Quantum programming is the process of designing and building executable quantum computer programs to achieve a particular computing result [10, 18]. A quantum program consists of

blocks of code, each of which contains classical and quantum components. Quantum operations can be divided into *unitary* operations (reversible and preserve the norm of the operands) and *non-unitary* operations (not reversible and have probabilistic implementations). A quantum computer program uses a quantum register of qubits to perform quantum operations and a classical register of classic bits to record the measurements of the qubits' states and apply quantum operators conditionally [6]. Therefore, a typical quantum program usually consists of two types of instructions (or statements). One is called *classical instructions* that operate on the state of classical bits and apply conditional statements. Another is called *quantum instructions* that operate on the state of qubits and measure the qubit values.

Q# [16] is a scalable, multi-paradigm, domain-specific quantum programming language developed by Microsoft for quantum computing. Q# allows users to write code that can be executed on machines of various computing capabilities. We can use it to simulate a few qubits on a local machine or thousands of qubits for enterprise-level applications. Q# is a multi-paradigm programming language that supports both functional and imperative programming styles. Q# can be used to write algorithms and code snippets that execute on quantum processors. Figure 1 shows an example Q# program, which will be described in Section 3 in detail.

3. Example

We next present an example to illustrate how our entanglement analysis works. Figure 1 is a quantum program written in Q#. The main body of the program is composed of the QFT algorithm and the GHZ algorithm. Since researchers may modify and reorganize the existing algorithm in the process of developing the algorithm, we construct the structure of calling the GHZ algorithm in the QFT algorithm.

Based on the three properties of entanglement introduced in Section 2, we convert the entanglement relation into the structure of an entanglement graph. In the entanglement graph, nodes represent qubits in a superposition state, and edges represent entanglement relations. Two nodes in an entanglement graph are entangled if they are connected. Therefore, we can modify the entanglement graph step by step according to the interprocedural control flow graph (ICFG). The entanglement relation of the whole program will be generated automatically at the end of the ICFG-based analysis.

First, our analysis algorithm constructs the corresponding *control-flow graph* (CFG for short) for each module (`function` or `operation`) in the program. Due to the nature of quantum operations, we transform the statements in Q# into

```
line:  tuple(operation, object).
```

Among them, since the Q# language allows the structure of quantum arrays, the `use` statement has the option to create the number of qubits when creating a qubit, so the `use` statement is transformed to

```
line:  tuple(use, name, number).
```

Based on the call relationship, we generate the corresponding *interprocedural control flow graph* (ICFG for short). Then, we classify the state of a qubit into the *classical state* (denoted by C) and the *quantum state (superposition state)* (denoted by Q) according to whether the qubit state is in the superposition state or not. At the same time, due to the uncomputation mechanism, we create a stack data structure for the Q to record the operations. The following table indicates how the C state and Q state are transformed.

```
1   namespace NamespaceQFT {
2       open Microsoft.Quantum.Intrinsic;
3       open Microsoft.Quantum.Diagnostics;
4       open Microsoft.Quantum.Math;
5       open Microsoft.Quantum.Arrays;
6
7       operation GHZ(target:Qubit[]): Unit {
8           H(target[0]);
9           Controlled X(target[0], target[1]);
10          Controlled X(target[1],target[2]);
11      }
12
13      operation initialqubit(target:Qubit) : Unit {
14          // initial qubit state
15      }
16
17      @EntryPoint()
18      operation QFTfor3qubits() : Unit {
19          use qs=Qubit[3];
20          initialqubit(qs[0]);
21          initialqubit(qs[1]);
22          initialqubit(qs[2]);
23
24          H(qs[0]);
25          Controlled R1([qs[1]], (PI()/2.0, qs[0]));
26          Controlled R1([qs[2]], (PI()/4.0, qs[1]));
27
28          use newq=Qubit[1];
29          GHZ([qs[1],qs[2],newq]);
30
31          H(qs[1]);
32          Controlled R1([qs[2]], (PI()/2.0, qs[1]));
33
34          H(qs[2]);
35
36          SWAP(qs[2], qs[0]);
37      }
38  }
```

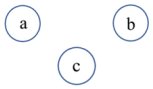
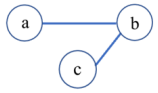
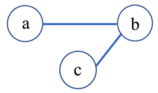
Fig. 1 An example Q# program.

$C \to Q$	Hadamard$(C) := Q, \{H\}$ CNOT$(C, C) := (Q, Q), (\{C_{line}\}, \{N_{line}\})$
$Q \to C$	If the stack of a qubit is empty, or there is no H and N_{line}

For the Q# code in Figure 1, the program starts executing from the `Entrypoint()` statement and creates three qubits with C states ($|0\rangle$ by default) at the time of the `use` statement (line 20). When passing through the `initialqubit` function, some qubits become Q states (superposition states), which may be assumed to be (Q, C, C). At this time, qs[0] becomes the Q state, so the `Uinit` operation is stored in the corresponding stack (if there is a specific operation, the specific operation is pushed into the stack).

When the `H` operation (line 25) is performed, it is put into the stack of `qs[0]`. If `H` is the inverse of `Uinit`, remove `Uinit`. At this point, the stack of `qs[0]` is empty, `qs[0]` changes from the Q state back to the C state, and deletes the node in the graph. If `H` is not the inverse of `Uinit`, push `H` onto the stack.

In the execution of the cphase (CP) operation (line 26), the statement has no effect on the entanglement relation because CP

GHZ (a, b, c)	State (Q, Q, Q)	Stack	Entanglement Graph
8: (H, a)	(Q, Q, Q)	a = {H}	
9: (CN, (a, b))	(Q, Q, Q)	a = {H, C_9} b = {N_9}	
10: (CN, (b, c))	(Q, Q, Q)	a = {H, C_9} b = {N_9, C_{10}} c = {N_{10}}	

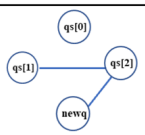
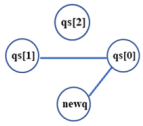
Entry	State	Stack	Entanglement Graph
20: (use, qs, 3)	(C, C, C)		
21~23: initial	(Q, C, C)	qs[0] = { Uinit }	
25: (H, qs[0])	(Q, C, C)	qs[0] = {Uinit, H}	
26: (CP, [qs[0], qs[1])	(Q, C, C)	qs[0] = {Uinit, H}	
27: (CP, [qs[0], qs[2])	(Q, C, C)	qs[0] = {Uinit, H}	
29: (use, newq, 1)	(Q, C, C, C)	qs[0] = {Uinit, H}	
30: GHZ	(Q, Q, Q, Q)	qs[0] = {Uinit, H} qs[1] = a = {H, C_9} qs[2] = b = {N_9, C_{10}} newq = c = {N_{10}}	
32: (H, qs[1])	(Q, Q, Q, Q)	qs[0] = {Uinit, H} qs[1] = {H, C_9, H} qs[2] = b = {N_9, C_{10}} newq = c = {N_{10}}	
33: (CP, [qs[1], qs[2])	(Q, Q, Q, Q)	qs[0] = {Uinit, H} qs[1] = {H, C_9, H, C_{33}} qs[2] = {N_9, C_{10}, P_{33}} newq = c = {N_{10}}	
35: (H, qs[2])	(Q, Q, Q, Q)	qs[0] = {Uinit, H} qs[1] = {H, C_9, H, C_{33}} qs[2] = {N_9, C_{10}, P_{33}, H} newq = c = {N_{10}}	
37: SWAP, qs[2], qs[1]	(Q, Q, Q, Q)	qs[0] = {N_9, C_{10}, P_{33}, H} qs[1] = {H, C_9, H, C_{33}} qs[2] = {Uinit, H} newq = c = {N_{10}}	

Fig. 2 The intraprocedural analysis based on the CFGs for the `GHZ` operation and `QFTfor3qubits` operation.

is not a state transformation operation, and the operation target is in the C state.

When executing the `GHZ` statement (line 30), we can use the result of the `GHZ` transformation and the entanglement relationship graph. Since three inputs are required when calling the `GHZ` function, we create the inputs `(a,b,c)` with the states (Q, Q, Q) for the three qubits. The same processing is used for the GHZ internal operations, which generate the corresponding stack and graph.

When `GHZ` is called, an alias relationship between the input qubits (`qs[1]` and `qs[2]`) and the `(a,b,c)` is created:

$$\{\ \texttt{qs[1]} \leftrightarrow \texttt{a},\ \texttt{qs[2]} \leftrightarrow \texttt{b},\ \texttt{newq} \leftrightarrow \texttt{c}\ \}.$$

Then the aliasing relation is lifted in turn, e.g., for `qs[1]`. Now the stack operation of `a` is passed to `qs[1]`. In the process of stack passing, as in the judgment of entering the stack, it is necessary to detect whether the top of the stack of `qs[1]` is the inverse operation of the bottom of the stack of `a`. The `qs[0]` in the example is the C state, and there is a state transition operation `H` in the stack of `a`, so it is possible for `qs[0]` to inherit the stack operation directly. Also, we connect the point connected to `a` in the entanglement graph to the node of `qs[1]` and delete the node of `a`, as shown in Figure 3.

For multiple calls to the `GHZ` function, we only need to perform the stack merge and graph transformation instead of repeatedly executing the `GHZ` function. The operations in the statements from line 32 to line 35 are not inverse operations on the top of the target stack and therefore enter the stack normally. Because the state-

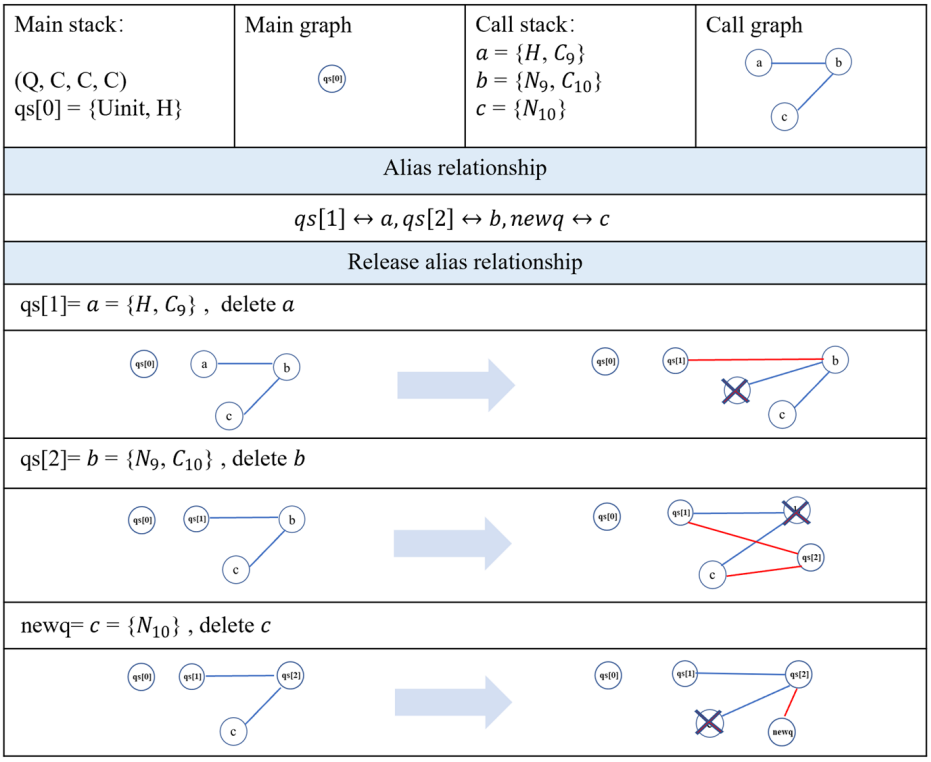


Fig. 3 The interprocedural analysis based on the ICFGs for calling the `GHZ` operation.

ment in line 37 performs a SWAP operation, the stacks of `qs[0]` and `qs[2]` are swapped, as well as the names of the nodes in the graph. The final entanglement relation returned by our algorithm is shown in Figure 2.

4. Entanglement Analysis

We next present our entanglement analysis for Q#. Sections 4.1 and 4.2 introduce the computation mechanism and the classical and quantum states. Section 4.3 introduce the intraprocedural entanglement analysis. Section 4.4 extends the intraprocedural analysis to interprocedural analysis.

4.1 Uncomputation Mechanism

The uncomputation mechanism is a unique mechanism of quantum computing. Since quantum operations are unitary operations, for each step of the operation, there must be a corresponding inverse operation. Uncomputation is the mechanism by which an operation cancels out when the operation and the corresponding inverse operation occur at the same time. The description using the matrix is that the multiplication of two operation matrices results in an identity matrix. Due to the no-cloning nature of qubits, the information in a qubit can only be used and cannot be copied. Therefore, each step of the operation modifies the information in the original qubit, and the uncomputation mechanism is used to restore it after the execution of the operation in order to ensure that the original information remains unchanged. In addition to this, for the intermediate results of operations in quantum programs, a new qubit is needed to save the result temporarily, that is, the *ancilla qubit*, and the ancilla qubit needs to be restored after the operation to ensure that the ancilla qubits does not affect the program. This process also uses the uncomputation mechanism. For some common gate operations, their corresponding inverse operations are shown in Table 1.

Table 1 Some quantum gates and their corresponding inverse gates.

Operation	Inverse Operation
Hadamard	Hadamard
NOT	NOT
Phase(a)	Phase(-a)
CNOT(a, b)	CNOT(a, b)

Moreover, any unitary operation can be composed of these four basic operations (Hadamard, NOT, Phase, Control). For the convenience of description, we use (H, N, P, C_{line}) to represent each of them, as shown in Table 2. To align multi-qubit operations, we use $_{line}$ to keep track of the operations.

Table 2 Examples for describing the quantum operations.

Operation	Description
CCNOT(a, b, c)	(C_{line}, C_{line}, N)
CPhase(a, b, c)	(C_{line}, C_{line}, P)

The Q# language allows user-defined operations, such as the GHZ function in the example in Figure 1. The syntactic structure of the customs operations is shown in Figure 4. For the Name function, Q# will have a corresponding adjoint Name function. When the Name function and the adjoint Name function act on the same qubit in succession, the uncomputation mechanism is implemented to cancel each other out.

Due to the existence of the uncomputation mechanism, the entanglement relationship in the program also has the possibility of

```
operation Name ( Q : Qubit ) : Unit is Adj+Ctl {

    body(...) {
       //opeartion
    }

    adjoint(...) {
        //inverse operation
    }

    controlled(cs, ...) {
       //controlled operation
    }
}
```

Fig. 4 The template of Q# operation.

elimination. Therefore, in order to improve the accuracy of program entanglement, the uncomputation mechanism cannot be ignored. Due to the strict requirements of the uncomputation mechanism for the order of operations, we use the stack structure to record operations. When an operation is pushed onto the stack, first determine the relationship between the push operation and the stack top operation. If they are mutually inverse operations, the uncomputation mechanism is implemented to delete the stack top operation. Otherwise, the push operation will be used as a new stack top operation to be judged with the next push operation. For entangled statements, we also create a stack to record entanglement operations, and when implementing the uncomputation mechanism, delete the entanglement operations on the top of the stack. When the stack is empty, it means that there is no entanglement between qubits.

4.2 Quantum State System

In quantum computing, the state of a qubit will be only in $|0\rangle$, $|1\rangle$, or superposition state, where only the superposition state leads to uncertainty in the results. Therefore, we denote the superposition state as the Q state and $|0\rangle$ and $|1\rangle$ as the C state. For single-qubit operations, NOT and Phase gates do not implement the transition between Q states and C states. Hadamard gates can change C states to Q states. If the Hadamard gate changes the Q state to the C state, it must satisfy the uncomputation requirement. For multi-qubit operations, the C state of the controlled bit is changed to a Q state only if the control bit is a Q state. The execution of the corresponding inverse operation can realize the transition from Q state to the C state.

For quantum operations, the state that the target qubit is in affects the results of the operation execution, especially the entanglement relation. By definition, the effect of entanglement arises from the difference in the measurement results of the target qubit, that is, the difference in the observation caused by the uncertainty of the operation. Thus, Q and C states will produce different results when faced with an entangled statement. In a quantum program, the statements that produce entanglement relations can be expressed as the relation between control and controlled bits. When the control bit of an entangled statement is a C state, the operation of the controlled bit is deterministic, and no branching occurs. If the control bit is in the Q state, the operation of the controlled bit depends on the uncertainty of the Q state, and thus the entanglement relation is expressed in the quantum state. Therefore, the nodes in the entanglement graph are created, and entanglement relations are constructed only when the qubit is in the Q state.

When the state of a qubit is changed from a Q state to a C state, the system can be converted to an expression of the tensor product from the point of view of separability. Therefore, the qubit will be disentangled from other qubits.

Table 3 show the state transitions for some single-qubit and multi-qubit operations.

Table 3 State transition for single-qubit and multi-qubit operations.

Single-qubit Operation	Multiple-qubit Operation
N(C) := C	CNOT(C, Q) := (C, Q)
P(C) := C (global phase)	CNOT(C, C) := (C, C)
H(C) := Q	CNOT(Q, C) := (Q, Q)
H(H(C)) := C	CNOT(Q, C) := (Q, Q)
H(Q_1) := Q_2 ($Q_1 \neq$ H(C)	CP(C, Q) := (C, Q)
P(Q) := Q	CP(Q, C) := (Q, C)
N(Q) := Q	CP(C, C) := (C, C)
	CP(Q, Q) := (Q, Q)

N: NOT gate, P: Phase gate, H: Hadamard gate
CNOT: Controlled NOT gate, CP: Controlled Phase gate

4.3 Intraprocedural Analysis

At the beginning of the quantum program execution, the created qubit will default to the $|0\rangle$ state, which is preset to the C state. During the execution of the program (which can be represented by a control flow graph of the program), the state of the qubit changes continuously. The qubit in the C state will not generate entanglement relations and will not affect subsequent operations. When a qubit in the C state encounters a state transition operation, we modify its state to Q state and create a stack for recording subsequent operations. At this point, the qubit in state Q is already capable of generating entanglement relations, so we create the corresponding nodes in the entanglement graph. When the entanglement statement is executed, we connect the corresponding nodes, thus transforming the entanglement relationship into a connected relationship of the graph. When an operation is executed at the qubit in state Q, if the corresponding stack is empty or there is no N operation in the stack, the state of the qubit is transformed to state C, and the node is deleted. Meanwhile, due to the transferability of the entanglement relation, the entanglement relation previously constructed by this qubit is passed on, so the original entanglement relation is passed on to the qubit connected to it in the entanglement graph.

4.4 Interprocedural Analysis

We perform an interprocedural analysis to deal with the problem of calls between functions or operations. There are many function-specific modular functions in existing quantum programs, such as GHZ, QFT, and Amplitude Amplification. For modular functions used at high frequencies, we generate the corresponding stack and entanglement graphs in advance. Unlike regular processing, the Q state is sensitive to all quantum operations since the call to the function requires input. The Q state behaves differently from the C state in the face of quantum operations. Therefore, we presuppose that all qubits of inputs are in

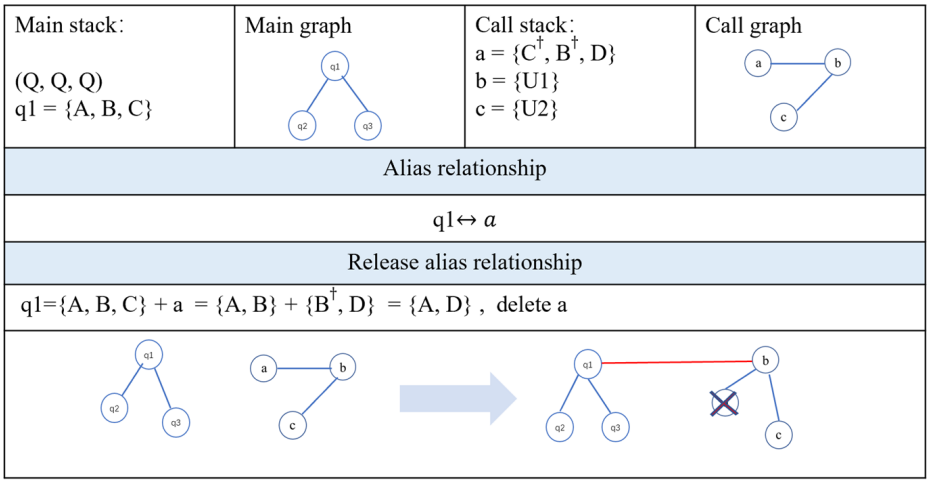


Fig. 5 An example for the interprocedural analysis

the Q state when processing the calling function. When the main program executes to the calling function, we create an alias relationship between the qubit as input in the main function and the qubit in the preprocessing calling function. Then the two stacks are merged, and the stack in the calling function is stacked from the bottom of the stack to the stack in the main function in order. If the inverse operation is satisfied, the offset is performed, otherwise, the stack is entered. For the entanglement graph, the qubit of the main function inherits the concatenation relation in the calling function and then removes the nodes in the calling function graph.

If the qubit of the main function cannot maintain the Q state after the merging of two stacks is finished, the concatenation relation is passed, and the node is deleted.

In Figure 5, qubit `q1` in the main operation is used as input to call other operations. We construct the {`q1`↔`a`} alias relationship. When the aliasing relation is lifted, the stack of `a` is merged into the stack of `q1`. During the merge, operations C and B are uncomputed by the inverse operations in the call stack. Therefore, at the end of the call, the stack of `q1` retains only operations A and D. For the entanglement graph, `q1` inherits the concatenation relation of `a` and removes node `a`.

5. Related Work

This section discusses some related work in the areas of entanglement analysis of quantum programs.

As a first step in dealing with entanglement analysis, Perdrix [12] proposed a type system reflecting the entanglement and separability between quantum bits that can approximate the entanglement relations of an array of quantum bits. Prost and Zerrari [14] proposed a logical entanglement analysis method that can deal with functional programming languages with higher-order functions. They followed the idea of classical aliasing analysis proposed by Berger *et al.* [4] and applied it to quantum entanglement analysis. Their logical framework can analyze more complex quantum programs, but not quantum programs without annotations and considering only pure quantum states. In [13], Perdrix further proposed an approach to entanglement analysis based on abstract interpretation [5]. In this approach, a correlation between concrete quantum semantics and a simple quantum programming language is established based on super operators, abstract semantics is introduced, and approximations are justified. Honda [8] proposed an alternative approach to entanglement analysis that considers the possibility of unitary gates withdrawing entanglement and measurement operations that may separate multiple quantum bits. The approach borrows some ideas from the work of Perdrix and uses the stabilizer formalism [11, 1] to improve the reasoning about the separability of quantum variables in quantum programs. However, these methods only analyze fine-grained entanglement between specific quantum bits and are therefore limited in their analytical scale to handle complex programs, such as teleportation. In comparison to these methods, our method aims to analyze the entanglement relations in large-scale quantum programs written in Q#.

ScaffCC [9] is a compiler framework for quantum programming language Scaffold [2]. ScaffCC supports conservative entanglement analysis by identifying each pair of quantum bits that may be entangled together in a program. The resulting entanglement information can help programmers design algorithms and perform debugging. ScaffCC uses data flow analysis techniques to obtain entanglement information in programs. Our entanglement analysis targets Q#, which is different from Scaffold in nature that needs unique entanglement analysis techniques.

Recently, Yuan *et al.* [19] formalized purity as a central tool for automatically reasoning about entanglement problems in quantum programs. A pure expression is one whose evaluation is not affected by measurements of qubits it does not own, meaning no entanglement with any other expression in the computation. They also designed *Twist*, the first language with a type system for reasoning about purity. Unlike their work, we focus on the analysis of entanglement information in Q# quantum programming language, which has more language features than *Twist* language.

6. Conclusion

This paper has presented a static entanglement analysis method for quantum programs developed in the practical quantum programming language Q#. To perform the analysis, our method first constructs an interprocedural control flow graph (ICFG) for

a Q# program and then calculates the entanglement information within each module and between modules of the program. We believe that our analysis approach can help improve the reliability and security of quantum programs by uncovering entanglement-induced errors in the programs.

As for future work, we would like to handle more language features in Q#, such as classical-quantum mixed programs, and apply our analysis approach to other quantum programming languages, such as Qiskit [3] and Cirq [17].

References

[1] Scott Aaronson and Daniel Gottesman. Improved simulation of stabilizer circuits. *Physical Review A*, 70(5):052328, 2004.

[2] Ali J Abhari, Arvin Faruque, Mohammad J Dousti, Lukas Svec, Oana Catu, Amlan Chakrabati, Chen-Fu Chiang, Seth Vanderwilt, John Black, and Fred Chong. Scaffold: Quantum programming language. Technical report, Department of Computer Science, Princeton University, 2012.

[3] Gadi Aleksandrowicz, Thomas Alexander, Panagiotis Barkoutsos, Luciano Bello, Yael Ben-Haim, David Bucher, Francisco Jose Cabrera-Hernández, Jorge Carballo-Franquis, Adrian Chen, Chun-Fu Chen, Jerry M. Chow, Antonio D. Córcoles-Gonzales, Abigail J. Cross, Andrew Cross, Juan Cruz-Benito, Chris Culver, Salvador De La Puente González, Enrique De La Torre, Delton Ding, Eugene Dumitrescu, Ivan Duran, Pieter Eendebak, Mark Everitt, Ismael Faro Sertage, Albert Frisch, Andreas Fuhrer, Jay Gambetta, Borja Godoy Gago, Juan Gomez-Mosquera, Donny Greenberg, Ikko Hamamura, Vojtech Havlicek, Joe Hellmers, Łukasz Herok, Hiroshi Horii, Shaohan Hu, Takashi Imamichi, Toshinari Itoko, Ali Javadi-Abhari, Naoki Kanazawa, Anton Karazeev, Kevin Krsulich, Peng Liu, Yang Luh, Yunho Maeng, Manoel Marques, Francisco Jose Martín-Fernández, Douglas T. McClure, David McKay, Srujan Meesala, Antonio Mezzacapo, Nikolaj Moll, Diego Moreda Rodríguez, Giacomo Nannicini, Paul Nation, Pauline Ollitrault, Lee James O'Riordan, Hanhee Paik, Jesús Pérez, Anna Phan, Marco Pistoia, Viktor Prutyanov, Max Reuter, Julia Rice, Abdón Rodríguez Davila, Raymond Harry Putra Rudy, Mingi Ryu, Ninad Sathaye, Chris Schnabel, Eddie Schoute, Kanav Setia, Yunong Shi, Adenilton Silva, Yukio Siraichi, Seyon Sivarajah, John A. Smolin, Mathias Soeken, Hitomi Takahashi, Ivano Tavernelli, Charles Taylor, Pete Taylour, Kenso Trabing, Matthew Treinish, Wes Turner, Desiree Vogt-Lee, Christophe Vuillot, Jonathan A. Wildstrom, Jessica Wilson, Erick Winston, Christopher Wood, Stephen Wood, Stefan Wörner, Ismail Yunus Akhalwaya, and Christa Zoufal. Qiskit: An Open-source Framework for Quantum Computing. jan 2019.

[4] Martin Berger, Kohei Honda, and Nobuko Yoshida. A logical analysis of aliasing in imperative higher-order functions. In *Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 280–293, 2005.

[5] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, 1977.

[6] Andrew W Cross, Lev S Bishop, John A Smolin, and Jay M Gambetta. Open quantum assembly language. *arXiv preprint arXiv:1707.03429*, 2017.

[7] Albert Einstein, Boris Podolsky, and Nathan Rosen. Can quantum-mechanical description of physical reality be considered complete? *Physical review*, 47(10):777, 1935.

[8] Kentaro Honda. Analysis of quantum entanglement in quantum programs using stabilizer formalism. *arXiv preprint arXiv:1511.01572*, 2015.

[9] Ali JavadiAbhari, Shruti Patil, Daniel Kudrow, Jeff Heckey, Alexey Lvov, Frederic T Chong, and Margaret Martonosi. ScaffCC: Scalable compilation and analysis of quantum programs. *Parallel Computing*, 45:2–17, 2015.

[10] Jarosław Adam Miszczak. *High-level structures for quantum computing*, volume 4. Morgan & Claypool Publishers, 2012.

[11] Michael A Nielsen and Isaac Chuang. Quantum computation and quantum information, 2002.

[12] Simon Perdrix. Quantum patterns and types for entanglement and separability. *Electronic Notes in Theoretical Computer Science*, 170:125–138, 2007.

[13] Simon Perdrix. Quantum entanglement analysis based on abstract interpretation. In *International Static Analysis Symposium*, pages 270–282. Springer, 2008.

[14] Frédéric Prost and Chaouki Zerrari. Reasoning about entanglement and separability in quantum higher-order functions. In *International Conference on Unconventional Computation*, pages 219–235. Springer, 2009.

[15] Erwin Schrödinger. Discussion of probability relations between separated systems. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 31, pages 555–563. Cambridge University Press, 1935.

[16] Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. Q#: enabling scalable quantum computing and development with a high-level dsl. In *Proceedings of the Real World Domain Specific Languages Workshop 2018*, pages 1–10, 2018.

[17] Google AI Quantum team. Cirq. 2018.

[18] Mingsheng Ying. *Foundations of quantum programming*. Morgan Kaufmann, 2016.

[19] Charles Yuan, Christopher McNally, and Michael Carbin. Twist: sound reasoning for purity and entanglement in quantum programs. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–32, 2022.

