量子程序的静态纠缠分析 s

Shangzhou Xia、Jianjun Zhao

九州大学{xia.shangzhou.218@s,zhao@ait}.
kyushu-u.ac.jp

arXiv:2304.05049v1 [cs.SE] 11 Apr 2023

摘要 量子纠缠在量子计算中发挥着至关重要的作用。纠缠信息对于理解量子程序的行为以及避免由纠缠引起的错误具有重要意义。纠缠分析是一种静态代码分析技术，用于确定哪些量子比特可能与其他量子比特发生纠缠，并构建纠缠图，以呈现纠缠量子比特之间交互的整体情况。本文提出了首个面向实用量子编程语言 Q# 所开发的量子程序静态纠缠分析方法。我们的方法首先为 Q# 程序构建过程间控制流图（ICFG），随后计算程序各模块内部及模块之间的纠缠信息。分析结果有助于提升量子程序的可靠性和安全性。

1 引言

近年来，越来越多的量子编程语言和环境（如 [13,14],）得到开发，以支持量子计算机编程。由于量子编程需要利用叠加和纠缠等独特的量子特性，因此被认为比经典编程更具挑战性，这使得理解量子程序的行为变得十分困难。因此，亟需开发方法和工具，以高效、自动地支持量子程序分析。

量子纠缠在量子计算中发挥着至关重要的作用。量子程序中量子比特之间的纠缠关系，对于理解量子程序的行为以及避免程序中由纠缠引发的错误具有重要意义。在量子程序执行过程中，不可避免地需要对一些量子比特进行测量。然而，在不考虑量子比特之间纠缠信息的情况下执行测量，可能会导致程序中某些量子比特的状态发生意外破坏，从而造成程序错误和信息丢失。此外，由于量子计算中的不可克隆原理，辅助量子比特常被用于协助程序执行过程中的操作。完成辅助操作后，必须确保辅助量子比特不再与程序中的其他量子比特发生纠缠。

此前已有若干纠缠分析方法 [5,6,8–10]，但直到现在，还没有适用于实用量子编程语言 Q# 的纠缠分析方法。本文介绍了*JiuChan*，一种针对Q# 程序的高效静态纠缠分析方法。我们的方法首先为 Q# 程序构建过程间控制流图（ICFG），然后计算程序各模块内部以及模块之间的纠- 缠信息。我们的方法建立了一个纠缠图，用于表示程序中纠缠量子比特之间交互的整体情况。分析结果有助于提高量子程序的可靠性和安全性。

本文的主要贡献是提出了一种面向 Q# 中量子程序的新型过程间纠缠分析算法。据我们所知，该算法是首个针对使用 Q# 这类实用量子编程语言编写的量子程序的纠缠分析算法。

本文其余部分的组织如下。第2 节介绍量子计算和 Q#的一些基本概念。第 3节通过一个示例说明纠缠分析算法的工作原理。第 4 节介绍分析框架和算法。第5节讨论一些实现问题。第 6 节讨论相关工作，第 7 节给出结论。

2 背景信息

我们简要介绍一些量子计算的基础知识 [7]。

2.1 量子计算的基本概念

2.1.1 量子比特

经典比特是经典计算中使用的二进制信息单位类型。它可以取两个可能值，即 0 或 1。量子比特（或量子比特）不同于经典比特，因为从理论上讲，它的状态由量子态空间中两个基的线性组合表示（由长度为 2 的列向量表示）。我们可以定义两个量子比特 $|0\rangle$ 和 $|1\rangle$，它们可以描述为

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

量子比特 $|0\rangle$ 和 $|1\rangle$ 是量子比特的计算基状态。换句话说，它们是量子态空间的一组基。

任意量子比特 $|e\rangle$ 都可以表示为两个基的线性组合：

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

其中 α 和 β 是复数，且 $|\alpha|^2 + |\beta|^2 = 1$。这种限制也称为归一化条件。

2.1.2 量子门和电路

正如数字电路中的逻辑门可以改变比特的状态一样，量子门也可以改变量子比特的状态。量子门可以只有一个输入和一个输出（单个量子态的转移），也可以有多个输入和多个输出（多个量子态的转移）。输入和输出的数量应相等，因为运算符需要可逆，这意味着量子计算中不能丢失任何信息。

非门。 非门作用于单个量子比特，可以交换两个基向量的系数： $NOT(\alpha|0\rangle + \beta|1\rangle)$ $= \alpha|1\rangle + \beta|0\rangle$。量子非门是经典数字电路中非门的扩展。

单个输入输出量子门可以用一个 2×2 矩阵表示。量子态经过量子门后的状态由量子态向量左乘量子门矩阵的值决定。与非门对应的量子门矩阵是

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

因此，量子比特通过非门后的结果可表示为

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

*Hadamard*门。Hadamard门同样作用于单个量子比特，可以根据其系数将现有量子态分解为：

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

这可以用一个矩阵表示：

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

尽管 Hadamard门 与经典数字电路中的与门和或门没有直接关系，但它在许多量子计算算法中有着重要应用。

量子比特有两个属性：幅度和相位。因此，根据作用范围，量子门可分为幅度门和非幅度门。上述非门和 Hadamard门属于幅度门，因为它们能够改变目标量子比特的幅度，从而改变状态输出的概率。此外，幅度门还可以改变控制关系。相反，非幅度门只能改变量子比特的相位，例如 T 门或 Z 门。相位信息无法反映在状态概率中；因此，有些算法会使用 Hadamard门的组合来处理相位信息。

2.2 量子纠缠

量子系统可能会表现出纠缠 [3,12], 这一量子力学现象。如果一个状态无法分解为更基本的部分，就被视为纠缠态。纠缠关系的存在使彼此独立的量子比特系统相互连接，从而实现不同量子比特系统之间的信息交互。同时，系统间纠缠关系的存在意味着这些系统无法被视为可分离系统。当对系统中的一些量子比特进行测量时，会影响其他量子比特的状态。

定义 1 对于一组 S 量子比特的状态 $|\varphi\rangle$，以及 S 的一个划分 (A, B)，当且仅当不存在两个状态 $|\varphi_A\rangle$ 和 $|\varphi_B\rangle$，分别属于两部分 A 和 B，使得 $|\varphi\rangle = |\varphi_A\rangle \otimes |\varphi_B\rangle$，$(A, B)$ 是纠缠的。(A, B) 可分，当且仅当 (A, B) 不纠缠。

例如，GHZ 状态 $\frac{\sqrt{2}}{2}(|000\rangle + |111\rangle)$ 是纠缠的。当观测第一个量子比特的结果为 0 时，其他量子比特必须全部为 0。因此，在缺乏系统纠缠信息的情况下，对某些量子比特进行测量会导致其所在的系统状态被破坏，从而在程序中触发错误并造成信息丢失。

根据 [8],，对于量子比特集合 S的状态 $|\varphi\rangle$ 以及 S的划分（A， B， C），纠缠关系具有以下三个性质。

- **传递性**：如果（A， B）纠缠，并且（B， C）纠缠，那么（A， C）纠缠。
- **对称**：如果（A， B）是纠缠的，那么（B， A）也是纠缠的
- **可消除**：如果（A， B）是可分离的，且对于某个操作 U， $U(|\varphi_A\rangle \otimes |\varphi_B\rangle)$ 是纠缠的，那么必然存在一个操作 V，使得 $VU(|\varphi_A\rangle \otimes |\varphi_B\rangle)$ 是可分离的。

2.3 反计算机制

由于量子操作是幺正操作，因此每个操作步骤都必须有对应的逆操作。当操作与对应的逆操作同时发生时，操作会相互抵消，这就是反计算机制。用矩阵描述时，两个操作矩阵相乘会得到一个单位矩阵。由于量子比特具有不可克隆性，量子比特中的信息只能被使用而不能被复制。因此，每个操作步骤都会修改原始量子比特中的信息，而反计算机制则用于在操作执行后将其恢复，以确保原始信息保持不变。此外，对于量子程序中的操作中间结果，需要一个新的量子比特来暂时保存结果，即辅助量子比特；操作完成后需要恢复辅助量子比特，以确保辅助量子比特不会影响程序。此过程同样使用反计算机制。对于一些常见的门操作，其对应的逆操作如表1所示。

Q# 语言允许用户定义操作，例如图2 中示例里的 GHZ操作。用户定义操作的语法结构如图1所示。对于 Name操作，Q#将有一个相应的 Name伴随操作。当 Name操作和 Name伴随操作

表1。一些量子门及其对应的逆门。

操作	逆操作
Hadamard门	Hadamard门
NOT	NOT
相位门(a)	相位门(-a)
受控非门(a, b)	受控非门(a, b)

依次作用于同一个量子比特时，反计算机制会使它们相互抵消。需要注意的是，由于非幅度操作不会改变量子比特的幅度，因此对控制位量子比特使用非幅度操作不会改变控制关系。因此，在满足反计算机制的操作对之间，即使对控制位量子比特使用了非幅度操作，反计算机制仍然成立。

```
操作 名称(Q:量子比特) : 单位类型为伴随+控制 {主体
(...) {//操作}伴随(...) {//逆操作}受控(cs,...) {//受
控操作}}
```

图1. Q# 操作的模板。

由于存在反计算机制，程序中的纠缠关系也有可能被消除。因此，必须考虑反计算机制，以提高程序纠缠分析的准确性。

3 示例

接下来，我们通过一个示例来说明纠缠分析的工作方式。图 2 展示了一个用 Q# 编写的量子程序。该程序的主体由两个操作 Entangle test 和 GHZ 组成。由于研究人员在开发算法的过程中可能会修改并重组现有算法，因此我们在用户开发的程序中构建了调用 GHZ 算法的结构。此外，该程序允许用户输入参数 a，以控制是否调用 GHZ 算法。EntryPoint（第 13 行）告知 Q# 编译器从何处开始执行程序。

基于第 2 节介绍的三种纠缠属性，我们使用纠缠图表示纠缠关系。在纠缠图中，节点表示处于叠加态的量子比特，边表示纠缠关系。若纠缠图中的两个节点相连，则它们处于纠缠态。因此，我们可以根据过程间控制流图（ICFG）逐步修改纠缠图。在基于 ICFG 的分析结束时，整个程序的纠缠关系将自动生成。

首先，我们的分析算法为程序中的每个操作模块构建相应的 控制流图 （简称控制流图）。由于量子操作的特性，我们将Q# 中的语句转换为以下结构，我们称之为操作行。

```
(Functor, Operation, Control, Target)
```

```
1 namespace NamespaceQFT {2openMicrosoft.Quantum.Intrinsic; 3
openMicrosoft.Quantum.Diagnostics; 4
openMicrosoft.Quantum.Math;5openMicrosoft.Quantum.Arrays;67
operationGHZ(target:Qubit[]): Unit {8 H(target[0]);9Controlled X(target
[0],target[1]); 10Controlled X(target[1],target[2]);11 }12 13
@EntryPoint() 14 operation Entangle_test(a:Int): Unit {15 use qs=Qubit[4];
16 17 H(qs[0]); 18 X(qs[1]); 19 H(qs[3]); 20 Controlled R1([qs[0]],(PI()/2.0,
qs[2])); 21 Controlled X(qs[0],qs[2]); 22 if a==1: 23 GHZ([qs[0],qs[1],qs
[2]); 24 ControlledR1([qs[1]], (PI()/4.0,qs[3])); 25 Controlled X(qs[1],
qs[0]); 26 if a==1: 27 Controlled R1([qs[1]], (-PI()/4.0, qs[3])); 28
Controlled X(qs[1],qs[0]); 29 H(qs[3])30 }31 }
```

图 2. 一个 Q# 程序示例。

根据调用关系，我们生成相应的过程间控制流图（简称 ICFG）。然后，根据量子比特的状态是否处于叠加态，将其状态分为经典态（以 0 态和 1 状态表示）和量子态（叠加态）（以 $\mathcal{Q}$-态表示）。同时，由于反计算机制，我们为 $\mathcal{Q}$ 创建一个栈数据结构来记录操作。第 IV 节将详细介绍量子比特的状态系统和状态转移规则。

对于图 2 中的 Q# 代码，程序从 Entrypoint() 语句开始执行，并在 use 语句（第 0 行）处创建四个处于 0 态（$|0\rangle$ 默认）的量子比特。收到第 1 行操作时，由于 qs[0] 处于 0 态，因此执行状态转移规则 0。该操作满足从 0 态变为 $\mathcal{Q}$-态的要求，因此我们将 qs[0] 的状态改为 $\mathcal{Q}$-态，并在纠缠图中创建相应的栈和节点。对于第 2 行和第 3 行操作，由于操作的目标量子比特不处于 $\mathcal{Q}$- 态，因此使用状态转移规则 0 或 1，按照与第一条操作行类似的方式处理。对于第 4 行操作，目标量子比特处于 0 态，且该操作不是幅度操作，因此可以忽略。对于第 5 行操作，由于它满足状态转移规则 0 的切换条件，也满足纠缠生成条件，因此我们将 qs[2] 的状态转换为 $\mathcal{Q}$-态，并创建相应的栈和节点。由于存在纠缠关系，我们将第 5 行操作添加到 Entangle 中，并在它们之间创建一条边。

执行 GHZ 语句（第 6 行）时，我们可以使用 GHZ 变换的结果和纠缠图。由于调用 GHZ 函数需要三个输入，因此我们为三个量子比特创建输入 (a,b,c)，其状态分别为 ($\mathcal{Q}$, $\mathcal{Q}$, $\mathcal{Q}$)。GHZ 内部操作也采用相同的处理方式，从而生成相应的栈和图。

当调用GHZ时，会在输入量子比特（qs[0], qs[1]和 qs[2]）与（a,b,c）之间创建别名关系:

$$\{\, \texttt{qs[0]} \leftrightarrow \texttt{a}, \texttt{qs[1]} \leftrightarrow \texttt{b}, \texttt{qs[2]} \leftrightarrow \texttt{c} \,\}.$$

然后 a 依次提升别名关系，例如，对于 qs[0]。现在，a 的栈操作被传递了下去。

到 qs[0]。在栈传递过程中，与进入栈的判断一样，需要检测 qs[0] 的栈顶是否为 a 的栈底的逆操作。需要确定 qs[0] 的状态是否在每次传递中发生变化，因为这将影响选择哪条转换规则。此外，我们将纠缠图中与 a 相连的点连接到 qs[0] 的节点，并删除 a 的节点，如图 3 所示。

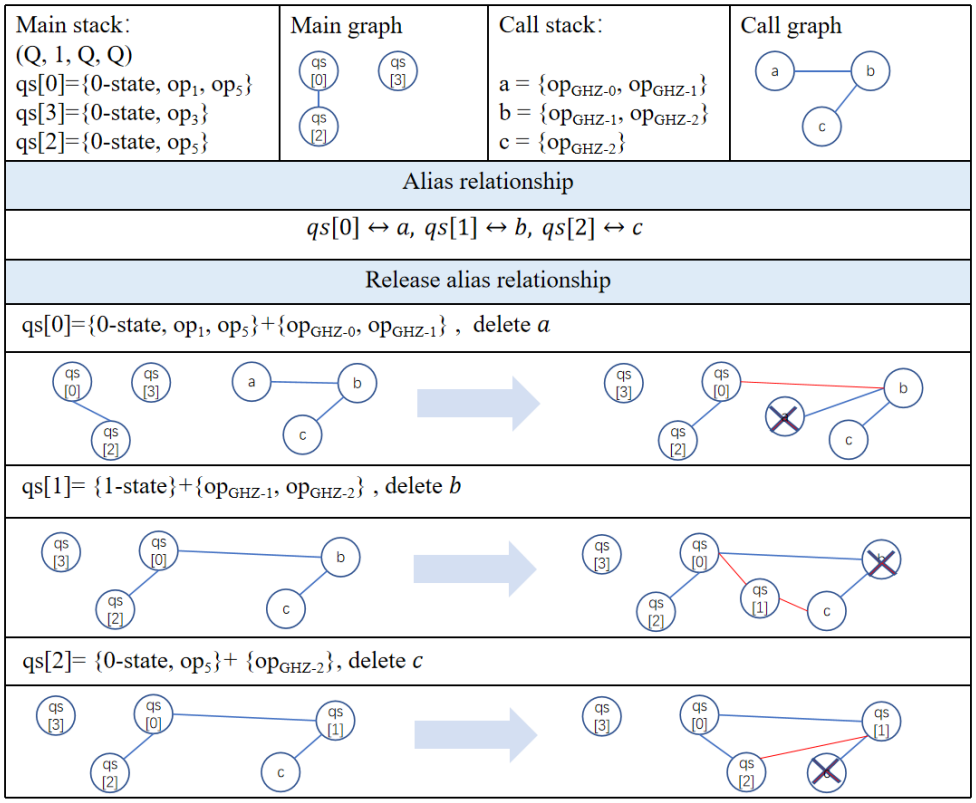


图 3。 基于用于调用 GHZ 操作的过程间控制流图进行的过程间分析。

对于GHZ 函数的多次调用，我们只需执行栈合并和图转换，而不必反复执行GHZ 函数。第7行和第 9行的操作同时满足相互逆操作和相同的控制关系，因此第 9行操作的执行将满足反计算机制，于是第7 行操作将被取消，相应的纠缠关系也将被消除。最后，当执行到第11 行操作时，同样满足反计算机制，因此 qs[3] 中的第3行操作也会被取消。此时，通过转换规则 $\mathcal{Q}$ 对 qs[3]的处理将变为初始 0态。此时，我们将删除相应的栈和节点。我们的算法返回的最终纠缠关系如图 4 所示。

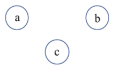
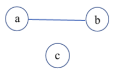
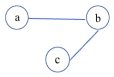
4 分析算法

接下来，我们将介绍针对 Q# 程序的纠缠分析算法。对于每个程序点，该算法都会为该点当前分析范围计算一张纠缠图。过程内分析的范围是截至该点正在分析的操作模块。过程间分析则将范围扩展至包含（传递调用的）操作模块。

4.1 程序表示

我们介 绍了几种程序表示形式，算法将在分析过程中使用这些形式 .

•量子 O 操作符号。 为了更好地提取信息，我们对量子操作进行转换

GHZ (a, b, c)	State	Stack	Entangle	Entanglement Graph
0: (None, H, (None, None), a)	(Q, Q, Q)	a = {op_{GHZ-0}}		
1: (Controlled, X,(None, [a]), b)	(Q, Q, Q)	a = {op_{GHZ-0}, op_{GHZ-1}} b = {op_{GHZ-1}}	{op_{GHZ-1}}	
2: (Controlled, X, (None, [b]), c)	(Q, Q, Q)	a = {op_{GHZ-0}, op_{GHZ-1}} b = {op_{GHZ-1}, op_{GHZ-2}} c = {op_{GHZ-2}}	{op_{GHZ-1}, op_{GHZ-2}}	

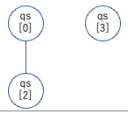
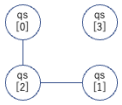
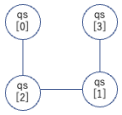
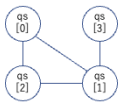
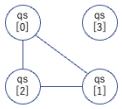
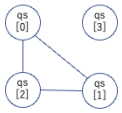
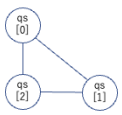
Entry	State	Stack	Entangle	Entanglement Graph
0: (use, qs, 4)	(0, 0, 0, 0)			
1: (None, H, (None, None), qs[0])	(Q, 0, 0, 0)	qs[0]={0-state, op_1}		
2: (None, X, (None, None), qs[1])	(Q, 1, 0, 0)	qs[0]={0-state, op_1}		
3: (None, H, (None, None), qs[3])	(Q, 1, 0, Q)	qs[0]={0-state, op_1} qs[3]={0-state, op_3}		
4: (Controlled, R1(pi/2), (None, [qs[0]]), qs[2])	(Q, 1, 0, Q)	qs[0]={0-state, op_1} qs[3]={0-state, op_3}		
5: (Controlled, X, (None, [qs[0]]), qs[2])	(Q, 1, Q, Q)	qs[0]={0-state, op_1, op_5} qs[3]={0-state, op_3} qs[2]={0-state, op_5}	{op_5}	
6: (Controlled, GHZ, (condition_1, None), [qs[0],qs[1],qs[2]])	(Q, Q, Q, Q)	qs[0]={0-state, op_1, op_5, op_{GHZ-0}, op_{GHZ-1}} qs[3]={0-state, op_3} qs[2]={0-state, op_5, op_{GHZ-2}} qs[1]={1-state, op_{GHZ-1}, op_{GHZ-2}}	{op_5, op_{GHZ-1}, op_{GHZ-2}}	
7: (Controlled, R1(pi/4), (condition_1, [qs[1]]), qs[3])	(Q, Q, Q, Q)	qs[0]={0-state, op_1, op_5, op_{GHZ-0}, op_{GHZ-1}} qs[3]={0-state, op_3, op_7} qs[2]={0-state, op_5, op_{GHZ-2}} qs[1]={1-state, op_{GHZ-1}, op_{GHZ-2}, op_7}	{op_5, op_{GHZ-1}, op_{GHZ-2}, op_7}	
8: (Controlled, X, (None, [qs[1]]), qs[0])	(Q, Q, Q, Q)	qs[0]={0-state, op_1, op_5, op_{GHZ-0}, op_{GHZ-1}, op_8} qs[3]={0-state, op_3, op_7} qs[2]={0-state, op_5, op_{GHZ-2}} qs[1]={1-state, op_{GHZ-1}, op_{GHZ-2}, op_7, op_8}	{op_5, op_{GHZ-1}, op_{GHZ-2}, op_7, op_8}	
9: (Controlled, R1(-pi/4), (condition_1, [qs[1]]), qs[3])	(Q, Q, Q, Q)	qs[0]={0-state, op_1, op_5, op_{GHZ-0}, op_{GHZ-1}, op_8} qs[3]={0-state, op_3, ~~op_7~~} qs[2]={0-state, op_5, op_{GHZ-2}} qs[1]={1-state, op_{GHZ-1}, op_{GHZ-2}, ~~op_7~~, op_8}	{op_5, op_{GHZ-1}, op_{GHZ-2}, ~~op_7~~, op_8}	
10: (Controlled, X, (None, [qs[1]]), qs[0])	(Q, Q, Q, Q)	qs[0]={0-state, op_1, op_5, op_{GHZ-0}, op_{GHZ-1}, op_8, op_{10}} qs[3]={0-state, op_3} qs[2]={0-state, op_5, op_{GHZ-2}} qs[1]={1-state, op_{GHZ-1}, op_{GHZ-2}, op_8, op_{10}}	{op_5, op_{GHZ-1}, op_{GHZ-2}, op_8, op_{10}}	
11: (None, H, (None, None), qs[3])	(Q, Q, Q, 0)	qs[0]={0-state, op_1, op_5, op_{GHZ-0}, op_{GHZ-1}, op_8, op_{10}} ~~qs[3]={0-state, op_3}~~ qs[2]={0-state, op_5, op_{GHZ-2}} qs[1]={1-state, op_{GHZ-1}, op_{GHZ-2}, op_8, op_{10}}	{op_5, op_{GHZ-1}, op_{GHZ-2}, op_8, op_{10}}	

图 4. 基于 GHZ 操作和纠缠测试操作的控制流图进行的过程内分析。

转换为以下操作行形式：

（函子、操作、控制、目标）

其中，函子表示操作的属性。Q# 中有四种属性：*None*、伴随、受控和受控伴随，分别对应原始操作、逆操作、受控操作和受控逆操作。操作表示量子操作的名称。控制表示操作执行期间的控制关系，可以是经典条件，也可以是量子比特控制（*Q* 控制）。由于这两种控制互不影响，因此在控制中，我们将使用（条件，*Q* 控制）这一表达式。对于条件，我们使用 0、1 和未知。此外，为了提高分析的准确性，我们还将确定不同条件之间的等价关系。对于*Q* 控制，我们将在下一节中进行详细说明。目标表示量子操作执行所作用的对象。

• 过程间控制流图。该算法使用过程间控制流图（ICFG）表示整个 Q# 程序的计算过程。ICFG 由程序的调用图和控制流图（控制流图）组合而成。每个控制流图表示一个操作的程序计算过程。在控制流图中，每个节点表示一条语句，每条边表示从一条语句到另一条语句的控制流。

过程间控制流图可以通过以下三个步骤构建。首先，构建Q# 程序中每个操作的控制流图。我们可以使用一些传统的控制流图构建算法，例如 [4]。其次，构建程序的调用图。我们可以使用一些复杂的调用图构建算法，例如 [11]。最后，将调用图中的每个节点替换为其对应的控制流图，从而形成整个程序的过程间控制流图。

在 Q# 中，有 11 种语句，分别是调用语句、返回语句、失败语句、变量声明、变量重新赋值、迭代、*while*语句、*repeat*语句、*if*语句、共轭语句和量子比特分配。由于 Q# 语言面向经典-量子混合程序，其中一些语句只用于经典程序。该算法只分析会影响纠缠信息的语句。我们假设程序已经过预处理，使所有与分析相关的语句都符合以下形式之一：

- **调用语句**：形式为 $l = op(l_0, \ldots, l_k)$ 的操作调用点。每个操作调用点都对应一个操作调用点 $m \in M$。
- **量子比特分配**：Q# 中的量子操作。
- **迭代**： 针对可迭代对象中的变量{*op*}
- **if语句**： 如果 条件 则 $\{op_1\}$ 否则 $\{op_2\}$
- **共轭语句**: 在 $\{op_1\}$ 中应用 $\{op_2\}$
- **repeat语句**: *repeat* $\{op_1\}$ 直到条件 *(fixup*$\{op_2\}$*)*

我们可以使用我们的符号将上述语句转换为控制流图，如下所示：

- **调用语句**: *(*函子*,* 操作*,* 控制*,* 目标*)*
- **量子比特分配**： *(None,* 操作*, None,* 目标*)*
- **迭代**：由于一旦知道迭代值，循环长度就已确定。我们可以使用 对于可迭代对象中的变量{（函子、操作、控制、目标）}来生成控制流图。
- **if语句**：*(*受控，op_1*,* 条件*,* 目标*)* + *(*受控伴随，op_2*,* 条件*,* 目标*)* +*(None,* op_2*, None,* 目标*)*

• **共轭语句**：这是由 Q# 提供的自动生成反计算语句的功能。当op_2 执行 *apply* 块时，会自动生成 op_1 *within* 块的逆操作。我们将共轭语句转换为以下形式：*(None,* op_1*, None,* 目标*)* + *(None,* op_2*, None,* 目标*)* + *(*伴随*,* op_1*, None,* 目标*)*

• **repeat语句**：repeat语句会在判断条件之前执行 op_1 。如果条件为真，则退出循环；否则，在进入下一次循环迭代之前，会执行 ***fixup*** 代码块中定义的额外 op_2 （如果存在）。我们可以将 repeat语句转换为以下形式：*(None,* op_1*, None,* 目标*)* + *n* * *(*受控*,* op_2*, not-condition,* 目标*)* + *(*受控*,* op_1*, not-condition,* 目标*)*] (n 是最大迭代次数)

经过这样的程序预处理后，控制流图中只保留与量子操作相关的标记。每个操作 op 的控制流图都以进入语句 $enter_{op}$ 开始，并以退出语句 $exit_{op}$ 结束。

分析将语句之间的控制流关系表示如下： $pred(st)$ 是可能紧邻 st 之前执行的语句集合，而 $succ(st)$ 是可能紧邻 st 之后执行的语句集合。每条语句 st 都有两个程序点：一个是 st 执行之前的程序点 $\bullet st$，另一个是 st 执行之后的程序点 $st\bullet$ 。

过程间分析使用调用图信息来计算在操作调用点可能被调用的操作集合。对于每个操作调用点， $m, callees(m)$是 m 可能调用的操作集合。给定一个操作， $op, caller(op)$是可能调用op的操作调用点集合。当前实现使用类层次结构分析的一个变体来获取此调用图信息。不过，算法可以使用程序运行时生成的实际调用图的任何保守近似。

• 纠缠图。 该算法使用纠缠图表示被分析程序在各个程序点处量子比特之间的纠缠关系。在纠缠图中，每个节点表示处于叠加态的量子比特，每条边表示两个量子比特之间的纠缠关系。若纠缠图中的两个节点相连，则它们处于纠缠状态。该算法根据程序的过程间控制流图，在各个程序点处修改纠缠图。

4.2 QuantumState 状态系统

在量子计算中，量子比特的状态只能是 $|0\rangle$、 $|1\rangle$ 或叠加态。*Q-control* 中的量子比特处于不同状态时，会影响受控操作的执行结果。当*Q-control* 中存在处于 $|0\rangle$的量子比特时，操作不会执行。对于*Q-control* 中处于$|1\rangle$的量子比特，即使执行操作，它也无法与目标 量子比特纠缠。只有处于叠加态的量子比特才能产生纠缠关系。因此，我们构建了一个量子态状态系统：0、1 和 $\mathcal{Q}$ 分别对应 $|0\rangle$、 $|1\rangle$或叠加态。只有当量子比特是某个操作的执行目标时，其状态才会发生转移。因此，我们将分析每个量子比特作为操作目标时的状态转移。以下将描述这些状态之间的转移规则：

• 状态转移规则 *0*：

转移	**条件**
$0 \to 0$:	执行操作是非幅值的 操作。
$0 \to 1$:	执行操作是非门(泡利 X 门) 操作- 满足非 Q 控制或全部的操作 Q 控制中的量子比特处于 1 状态。
$0 \to \mathcal{Q}$:	执行操作是 HAD 操作- 作，或满足于 NOT(X) 操作 Q 控制在 $\mathcal{Q}$状态中存在一个量子比特。

• 状态转移规则 *1*：

转移	条件
$1 \rightarrow 0$:	执行操作是满足非 Q 控制条件的非门（泡利 X 门）操作，或所有 Q 控制中的量子比特均处于 1 状态条件=l。
$1 \rightarrow 1$:	执行操作是非幅度的操作或 NOT(X) 操作，其满足 非 Q 控制, 或全部 量子比特 in Q 控制 are in 1 状态 在条件下-tion=未知。
$1 \rightarrow \mathcal{Q}$:	执行操作是 HAD 操作-操作，或满足以下条件的 NOT(X) 操作红黑大战 Q 控制存在一个处于 $\mathcal{Q}$状态的量子比特。

处于叠加态的量子比特将存在无限多个状态。如果精确记录每个状态，程序分析的计算成本将呈指数级增长。因此，我们使用栈结构记录每个 $\mathcal{Q}$ 态量子比特的操作过程。当 0态和 1 状态量子比特转变为 $\mathcal{Q}$ 态时，我们会创建一个栈来记录量子操作。$\mathcal{Q}$ 态的转换规则也将取决于栈。

• 转换规则 $\mathcal{Q}$:

转移	条件
$\mathcal{Q} \rightarrow 0/1$:	由于反计算机制，量子比特的状态会转变为原始 0态或 1 状态，当操作的执行使目标的栈量子比特空。
$\mathcal{Q} \rightarrow \mathcal{Q}$:	在其他情况下，量子比特将保持着原始 $\mathcal{Q}$-状态。

4.3 过程内分析

对 Q# 程序进行预处理后，我们使用处理后的列表（其中存储了操作行）作为遍历对象，并将其输入过程内分析。在程序分析中，我们将引入三个部分来记录分析结果：(状态, 栈, 纠缠)。状态用于记录每个量子比特的状态系统。栈 用于记录处于 $\mathcal{Q}$-状态的量子比特的操作内容。由于反计算机制对操作顺序有严格要求，我们使用栈来记录操作。纠缠用于记录量子比特之间的纠缠关系。

对于控制流图中的每一行操作，我们将系统中的变化表示为（状态、栈、纠缠）。首先，使用我们定义的转移规则修改状态 。然后，对于栈，我们定义消除规则 $Kill_S$ 和生成规则 Gen_S 如下:

$$Kill_S(x, op_i) = \begin{cases} Stack_{IN}(x)[-1] & if\ x\ is\ op_i.Target and\ Stack_{IN}(x)[-1] \\ & is\ inverse\ with\ op_i. \\ Stack_{IN}(op_i.Target)[-1] & if\ x\ is\ in\ op_i.Control\ and \\ & Kill_S(op_i.Target, op_i) \neq \emptyset \\ \emptyset & otherwise \end{cases}$$

$$Gen_S(x, op_i) = \begin{cases} op_i & if\ (x\ is\ op_i.Target\ or\ x\ is\ in\ op_i.Control)\ and \\ & Kill_S(x, op_i) = \emptyset \\ \emptyset & otherwise \end{cases}$$

在 $Kill_S$ 中，我们判断新操作是否满足反计算机制。如果满足，则删除所有相应的操作记录 op_i。否则，将该操作添加到栈中。栈 的传递函数如下:

$$Stack_{OUT}(x, op_i) = (Stack_{IN} - Kill_S(x, op_i)) \cup Gen_S(x, op_i)$$

对于纠缠，我们还定义了相应的消除规则 $Kill_E$ 和生成规则 Gen_E，如下所示：

$$Kill_E(op_i) = \begin{cases} Stack_{IN}(op_i.Target)[-1] & if\ Kill_S(op_i.Target, op_i) \neq \emptyset \\ & and\ Stack_{IN}(op_i.Target)[-1] \\ & is\ in\ Entangle_{IN} \\ \emptyset & otherwise \end{cases}$$

$$Gen_E(x, op_i) = \begin{cases} op_i & if\ Kill_E = \emptyset\ and\ \mathcal{Q} - state\ is\ in\ State[op_i.Control] \\ \emptyset & otherwise \end{cases}$$

纠缠的传递函数如下：

$$Entangle_{OUT}(op_i) = (Entangle_{IN} - Kill_E(op_i)) \cup Gen_E(op_i)$$

对于整个算法流程，首先需要验证每个操作在当前量子比特状态系统中是否能够有效执行，因此我们引入一个检查是否执行函数。该函数会检查每个操作的控制。当控制中的条件为 False，或者Q 控制中存在处于 0态的量子比特时，说明该操作不会执行，因此可以忽略。

确保每个操作都能有效执行后，我们将引入检查基本操作函数，以检查操作的调用关系。当检查到某个操作存在调用关系时，我们将使用过程间分析进行处理。对于基本操作，我们将继续执行过程内分析，并创建一个可扩展的基本操作库，以便快速检查。

确保操作能够正确执行且属于过程内分析的处理范围后，我们将根据目标量子比特所处的状态采用不同的处理方式：对于处于 0态或 1 状态的量子比特，我们将分别使用状态转移规则 0 和状态转移规则 1，并修改状态中相应的量子比特状态。当存在一个被转换为 $\mathcal{Q}$-状态的量子比特时，我们将在栈中创建相应的操作栈，并将该量子比特的原始状态及相应的操作行压入栈中。对于 $\mathcal{Q}$-状态的量子比特，我们检查当前操作与相应栈中的最后一个操作是否满足反计算机制。我们引入检查逆操作和检查受控函数，分别用于检查这两个操作是否互为逆操作，以及它们之间的控制关系是否相同。如果两个函数的检查均通过，则当前操作满足反计算机制，因此我们将相应栈中的操作弹出，并执行状态转移规则 $\mathcal{Q}$，以检测目标量子比特的状态转移。由于当前操作会使原操作失效，因此原操作产生的纠缠关系将被消除。我们将从纠缠中移除相应的操作，并从纠缠图中移除相应的边。如果两个函数的检查未能同时通过，则当前操作无法消除原操作，因此我们将当前操作压入栈中。如果当前操作的Q 控制中存在处于 $\mathcal{Q}$-状态的量子比特，则当前操作将产生新的纠缠关系。因此，我们将当前操作保存在纠缠中，并创建一条边，将Q 控制中所有处于 $\mathcal{Q}$-状态的量子比特与目标量子比特相连。整体流程如图 5 所示。

我们将重复此过程，直到遍历完操作列表。由于现有的量子程序没有死循环，因此我们的方法会在某处终止。此时，执行量子程序后，可以从状态中获得所有量子比特的状态，并从纠缠图中获得程序中的纠缠关系。

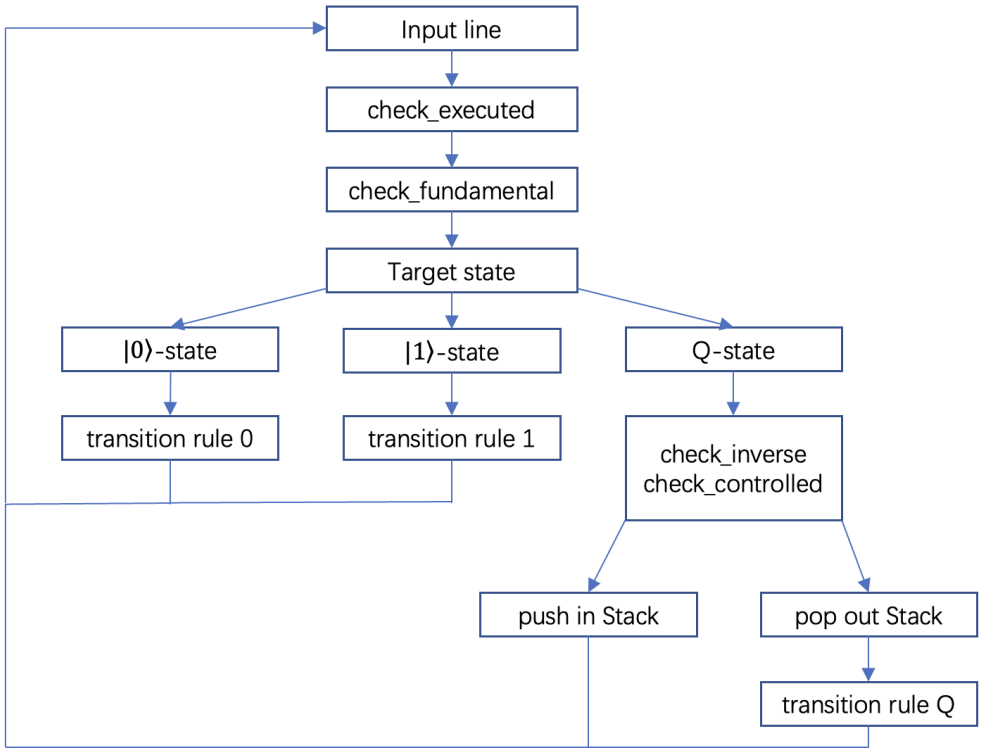


图 5. 过程内分析的算法流程

4.4 过程间分析

该算法使用数据流分析，在每个操作点生成纠缠关系。对每个操作的分析，都从为该操作中的第一条语句构建纠缠图 (a, b, c) 开始。

我们执行过程间分析，以处理操作模- 块之间调用的问题。现有量子程序中存在许多特定于函数的模块化函数，例如 GHZ、QFT和振幅放大（AA）。我们预先为高频使用的模块化函数生成相应的栈和纠缠图。不同于常规处理， Q-状态对所有量子操作都很敏感，因为调用函数需要输入。面对量子操作时，Q-状态的行为不同于0态和1状态。因此，我们预设在处理调用操作模块时，所有输入量子比特都处于 Q-状态。当主程序执行调用操作模块时，我们在主模块中作为输入的量子比特与预处理调用操作模块中的量子比特之间创建别名关系。然后合并两个栈，并按照从栈底到栈顶的顺序，将调用模块中的栈依次压入主模块中的栈。如果满足逆操作，则执行偏移；否则，将其压入栈中。对于纠缠图，主操作中的量子比特继承调用操作中的连接关系，然后移除调用操作图中的节点。

如果两个栈合并完成后，主操作的量子比特无法保持 Q-状态，则连接关系将被传递，并删除节点。

由于 Q# 在调用操作时可以生成操作的各种版本（例如 受控），因此我们还需要根据所使用的版本调整相应的操作行。因此，我们将提出一个 行包装器 ，根据调用操作的属性调整被调用操作内部的操作行，从而修改相应的函子 和控制。

4.5 可靠性

在现有方法中，所有可能生成纠缠的语句都被视为纠缠关系。尽管这种方法可以保证纠缠关系的可靠性，但仍会有。

大量虚假的纠缠关系。在我们的方法中，由于引入了反计算机制，程序中的纠缠关系并非单调递增。因此，就我们方法的可靠性而言，只需证明引入反计算机制后被消除的纠缠关系都是误报即可。

反计算机制取决于操作的两个方面：两个操作是否互为逆操作，以及两个操作之间的控制关系是否一致。对于第一种情况，即逆操作，我们的方法在此阶段不会引入等价操作，因此我们使用量子计算的基本知识和 Q# 中的 伴随函子来保守地定义逆操作。对于第二种情况，即控制关系，我们将其分为两类：量子控制和经典控制。对于量子控制，我们将判断两个操作是否由相同的量子比特控制，并判断控制量子比特是否处于相同的幅值状态。对于经典控制，我们将比较两个操作的经典约束是否完全相同。

我们的方法对反计算机制的判定，使我们只能消除绝对互为逆操作的纠缠关系。因此，我们消除的纠缠关系属于无论如何都应当进行反计算的假阳性关系。对于只有在某些实际情况下才能消除的纠缠关系，我们不会将其视为满足反计算机制，因此我们的方法会保留其中一些假阳性关系。

此外，由于纠缠关系只会出现在叠加态量子比特之间以及某些操作执行无效时，我们定义的状态系统也消除了某些误报关系。

5 提议的实现

我们计划开发一个名为 *JiuChan* 的工具，以支持我们的方法。图 6 展示了 *Ji-uChan*的框架：输入包括Q#程序、程序的调用图（该调用图使用 [11], 中的算法生成）以及每个量子操作的控制流图（使用算法 [4] 生成）；输出则是所有程序点处的纠缠图（EG）。对于一个 Q# 程序，我们首先生成相应的过程间控制流图，并将每个量子操作转换为结构化形式。然后，对于所有调用操作，我们首先使用过程内分析生成相应的栈、纠缠和纠缠图。接着，我们对主操作执行过程内分析；在分析过程中遇到调用操作时，则使用已经分析得到的结果执行过程间分析。每个程序节点之间的关系会被生成并表示为一个无向图。

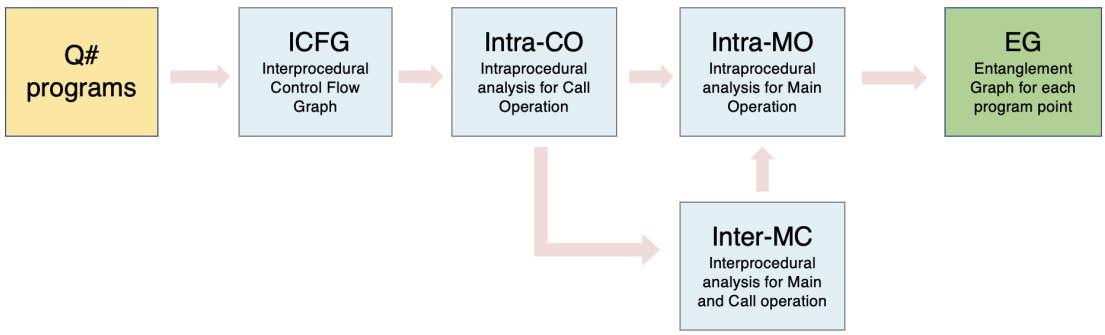


图 6. 九禅工具框架

6 相关工作

本节讨论量子程序纠缠分析方面的一些相关工作。

作为处理纠缠分析的第一步，Perdrix [8] 提出了一种类型系统，用于反映量子比特之间的纠缠与可分离性，并能够近似表示量子比特数组的纠缠关系。Prost 和 Zerrari [10] 提出了一种逻辑纠缠分析方法

用于处理具有高阶函数的函数式编程语言。他们遵循Berger等 [1] 提出的经典别名分析思想，并将其应用于量子纠缠分析。他们的逻辑框架可以分析更复杂的量子程序，但无法分析没有注释的量子程序，并且只考虑纯量子态。在 [9], Perdrix 进一步提出了一种基于抽象解释的纠- 缠分析方法 [2]。该方法在具体量子语义与一种基于超算子的简单量子编程语言之间建立了关联。Scaf- fCC [6] 通过识别程序中可能彼此纠缠的每一对量子比特，支持保守的纠缠分析。由此获得的纠缠信息可以帮助程序员设计算法并进行调试。ScaffCC 使用数据流分析技术来获取程序中的纠缠信息。Yuan等 [15] 将纯性形式化为自动推理量子程序中纠缠问题的核心工具。纯表达式是指其求值不受对其不拥有的量子比特进行测量影响的表达式，这意味着在计算过程中它不与任何其他表达式发生纠缠。他们还设计了 *Twist*，这是第一种具有用于推理纯性的类型系统的语言。然而，这些方法只分析特定量子比特之间的细粒度纠缠，其分析规模也受到限制。此外，它们只考虑简单的量子编程语言，无法分析实际的量子编程语言。相比之下，据我们所知，*JiuChan* 是首个旨在分析实际量子编程语言 Q# 中纠缠关系的静态纠缠分析方法，用于处理大规模量子程序。

7 结论

本文提出了一种针对实用量子编程语言 Q# 开发的量子程序静态纠缠分析方法。该方法首先为 Q# 程序构建过程间控制流图（ICFG），以便开展分析。随后，计算各模块内部以及程序模块之间的纠缠信息。通过揭示程序中由纠缠引发的错误，我们的分析方法有助于提高量子程序的可靠性和安全性。

作为未来工作，我们希望处理 Q# 中更多的语言特性，例如经典-量子混合程序，并将我们的分析方法应用于其他量子编程语言。我们正在实现一个面向 Q# 的纠缠分析工具，以展示该方法的有效性。下一步，我们将开展一些实验，评估纠缠分析在实际量子软件开发中的实用价值。

参考文献

[1] Martin Berger、Kohei Honda 和 Nobuko Yoshida。《命令式高阶函数中别名的逻辑分析》。载于第十届 *ACM SIGPLAN* 函数式编程国际会议论文集，第 280–293 页，2005 年。[2] Patrick Cousot 和 Radhia Cousot。《抽象解释：通过构造或近似不动点，为程序静态分析建立统一的格模型》。载于第四届 *ACM SIGACT-SIGPLAN* 编程语言原理研讨会论文集，第 238–252 页，1977 年。[3] Albert Einstein、Boris Podolsky 和 Nathan Rosen。《对物理实在的量子力学描述能否被认为是完备的？》《物理评论》，47(10):777，1935 年。[4] Jeanne Ferrante、Karl J Ottenstein 和 Joe D Warren。《程序依赖图及其在优化中的应用》。《*ACM* 编程语言与系统汇刊 *(TOPLAS)*》，9(3):319–349，1987 年。[5] Kentaro Honda。《使用稳定子形式分析量子程序中的量子纠缠》。*arXiv* 预印本 *arXiv:1511.01572*，2015 年。[6] Ali JavadiAbhari、Shruti Patil、Daniel Kudrow、Jeff Heckey、Alexey Lvov、Frederic T Chong 和 Margaret Martonosi。《ScaffCC：量子程序的可扩展编译与分析》。《并行计算》，45:2–17，2015 年。[7] Michael A Nielsen 和 Isaac Chuang。《量子计算与量子信息》，2002 年。

[8] Simon Perdrix。《用于纠缠与可分离性的量子模式和类型》。《理论计算机科学电子札记》，170:125–138，2007 年。[9] Simon Perdrix。《基于抽象解释的量子纠缠分析》。载于国际静态分析研讨会，第 270–282 页。Springer，2008 年。[10] Fr´ed´eric Prost 和 Chaouki Zerrari。《量子高阶函数中纠缠与可分离性的推理》。载于国际非常规计算会议，第 219–235 页。Springer，2009 年。[11] Barbara G Ryder。《构建程序的调用图》。《*IEEE* 软件工程汇刊》，(3):216–226，1979 年。[12] Erwin Schr¨odinger。《分离系统间概率关系的讨论》。载于《剑桥哲学学会数学论文集》，第 31 卷，第 555–563 页。剑桥大学出版社，1935 年。[13] Krysta Svore、Alan Geller、Matthias Troyer、John Azariah、Christopher Granade、Bettina Heim、Vadym Kliuch- nikov、Mariia Mykhailova、Andres Paz 和 Martin Roetteler。《Q#：通过高级领域专用语言实现可扩展的量子计算与开发》。载于*2018* 年真实世界领域专用语言研讨会论文集，第 1–10 页，2018 年。[14] Google AI 量子团队。《Cirq》，2018 年。[15] Charles Yuan、Christopher McNally 和 Michael Carbin。《Twist：对量子程序中的纯性与纠缠进行可靠推理》。《*ACM* 编程语言论文集》，6(POPL):1–32，2022 年。