面向代码安全的量子中间表示形式化

罗俊杰 赵建军九州大学 九州大学 luo.junjie.609@s.kyushu-u.ac.jp
zhao@ait.kyushu-u.ac.jp

arXiv:2303.14500v1 [quant-ph] 25 Mar 2023

量子中间表示（QIR）是微软开发的一种基于 LLVM 的量子程序编译器中间表示。QIR 旨在为量子程序编译器提供一种通用解决方案，不依赖前端语言和后端硬件，从而避免重复开发中间表示和编译器。由于 QIR 仍在开发中，目前仅以自然语言描述，且缺乏形式化定义，导致其解释存在歧义，量子函数的实现也不够严谨。本文为 QIR 的数据类型和指令集提供形式化定义，旨在为 QIR 中的操作和中间代码转换提供正确性与安全性保证。为验证我们的设计，我们展示了一些不安全的 QIR 代码示例，并说明通过我们的形式化方法可以检测出其中的错误。

1 引言

含噪中尺度量子计算机 (NISQ) [21] 的发展为量子软件的研究与开发带来了新的机遇。随着量子硬件不断发展，量子程序处理器（QPUs）有望进一步补充并加速现有的经典科学计算工作流，这种计算模式称为异构量子-经典计算。为实现这种计算模式，人们开发了 Qiskit [1], 、Cirq [27], 、Q# [26], 、Quipper [8],和 ProjectQ [28]等多种编程语言。与经典程序类似，Q# 等编译型量子编程语言要求将量子程序通过编译器编译为中间表示（IR），随后再对其进行优化和转换，以便在指定平台上高效执行。由于中间表示具有平台无关特性，其优化器和可执行文件生成器可以在多种源语言下重复使用，并根据目标执行平台生成相应的可执行代码。为了使中间表示适应量子特性，通常需要开发新的中间表示或扩展现有中间表示。量子编程语言中使用的中间表示包括 MLIR [11, 14], 、SQIR [9], 和 OpenQASM [5]。

量子中间表示 (QIR) [7] 是微软开发的一种新型量子程序中间表示，基于广受欢迎的开源 LLVM 中间语言 [10]。QIR 通过规定一套在 LLVM 中表示量子结构的规则（无需任何扩展或修改），使量子计算工作流和任务得以表示。其目标是为多种量子编程语言和量子计算平台提供统一、通用的接口，从而促进开发可在编译器机制中重复使用的通用量子编译工具。

基于 QIR 的编译器会将高级量子源语言转换为 QIR，并交由目标后端执行（见图 1）。由于 QIR 是基于 LLVM 设计和实现的，因此可以应用经典的 LLVM 优化方法来转换和

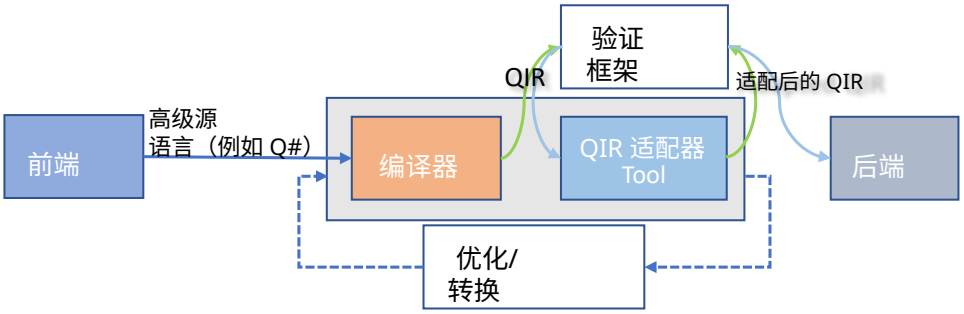

图 1：QIR 编译器架构。

优化 QIR 代码。然而，由于 QIR 仍处于发展初期，缺乏明确的形式语义。因此，很难严格证明其执行过程以及优化和转换的正确性，也就难以保证量子程序能够正确运行。在我们的工作中，我们希望为 QIR 设计一种形式化方法，并基于该方法开发一个验证框架，以便在 QIR 代码实际执行之前，通过该验证框架对其安全性进行测试。

在我们的工作中，我们聚焦于 QIR 的核心函数，这些函数对 QIR 程序的正确运行至关重要。因此，本文优先对这些核心函数的语法和语义进行形式化，以便检测 QIR 核心函数执行过程中可能出现的错误，并确保 QIR 的基本功能能够正常执行。我们的贡献可概括如下：

- 我们基于 Zhao 等人 [29]的工作对 QIR 的语法进行形式化。具体而言，我们调整了其抽象语法，移除了 QIR 中未使用的 LLVM 指令和类型，并补充了 QIR 专用数据类型。

- 我们为 QIR 中的重要指令设计了语义，例如量子比特的分配与释放、门操作和测量。这些操作构成了最基本量子程序函数的实现，基于这些操作进行形式化，能够帮助我们捕获代码中的不安全部分。

- 我们设计了一种量子比特管理模型，通过该模型结合我们的形式化方法，可以捕获 QIR 代码中的不安全部分（例如，克隆量子比特以及使用已释放的量子比特）。
- 我们将该形式化方法应用于真实的不安全 QIR 代码案例，以验证其有效性。

本文其余部分的安排如下。第 2 节介绍 LLVM 中间表示（LLVM IR）和 QIR 的背景信息。第 3 节阐述我们对 QIR 语法和语义的形式化。第 4 节通过真实案例验证我们的形式化方法。第 5 节讨论相关工作，第 6 节给出结论。

2 背景

本节简要介绍 LLVM 中间表示（QIR 的基础）以及 QIR 本身。

2.1 基本概念

与使用经典比特存储信息的经典程序不同，量子程序使用量子比特（“qubits”）作为数据存储介质。经典比特只能处于 0 或 1 状态，而量子比特可以同时处于 0 和 1 状态，这称为量子叠加态。量子比特的叠加态可表示为 $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$，其中 $|\rangle$ 称为狄拉克符号，且 $|\alpha|^2 + |\beta|^2 = 1$。我们无法直接观测处于叠加态的量子比特；相反，必须对其进行测量，并以概率 $|\alpha|^2$ 得到 0 状态，或以概率 $|\beta|^2$得到 1 状态。

在量子计算中，我们通常使用量子逻辑门来控制量子比特的状态。一些基本门操作包括：

- **X门（非门）**：当一个量子系统执行X门操作时，其状态会从$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$变为 $|\psi\rangle = \beta|0\rangle + \alpha|1\rangle$。我们可以用矩阵表示这一操作：

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

- **Y门和Z门**：类似地，Y门和Z门具有如下表达式：

$$Y = \begin{bmatrix} 0 & -1i \\ 1i & 0 \end{bmatrix} \quad \& \quad Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \tag{2}$$

- **哈达玛门（H门）**：哈达玛门可以将 $|0\rangle$态或 $|1\rangle$态转换为$|0\rangle$和 $|1\rangle$ 的等概率叠加态：

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{3}$$

- **受控门**：与上述只能接收一个量子比特作为输入的单量子比特门不同，受控门可以接收多个量子比特作为输入。具体来说，输入受控门的量子比特分为控制量子比特和目标量子比特；只有当两个控制量子比特的状态均为 $|1\rangle$时，目标量子比特的状态才会改变。例如，对于CNOT门（受控非门），其逻辑表达式为：

$$|A, B\rangle \rightarrow |A, B \oplus A\rangle \tag{4}$$

类似地，托福利门（也称为 CCNOT 门）具有两个控制量子比特，它们共同控制目标量子比特的翻转，其表达式为：

$$|A, B, C\rangle \rightarrow |A, B, C \oplus AB\rangle \tag{5}$$

以上所有量子逻辑门都是可逆的，因此量子计算过程中不会发生信息丢失。通过对量子比特执行门操作，可以组合成量子电路，进而实现量子算法。这是当今量子程序广泛采用的模型。另一项重要操作是测量。对单个量子比特执行测量操作，可以得到处于 0 或 1 状态的经典比特。有关量子计算的更多内容，可参阅 Nielsen 和 Chuang 的著作 [20]。

2.2 LLVM 中间表示

LLVM 中间表示 [10] 是一种高级静态单赋值形式（SSA）语言，LLVM 编译器基础设施将其用作源代码的中间表示。它是一种独立于源语言和目标平台的源代码表示。目前，它已得到广泛应用，并有许多与之相关的转换和优化方法。编译器可以据此对程序代码执行各种优化，包括公共子表达式消除、死代码消除和循环展开。这些优化能够提升生成的机器码的性能，使其运行得更快、更高效。

LLVM 中间表示是一种强类型语言，这意味着每个值都有特定类型，例如整数或浮点值。这种设计有助于防止类型错误，并使编译器能够生成更高效的代码。与 RISC 指令类似，LLVM 中间表示采用三地址形式。结合其强类型特性，它既支持基于单个编译单元进行编译和优化，也支持在链接时利用类型信息进行精确的全局优化。LLVM 中间表示还支持控制流、循环和其他常见的语言结构，因此能够表示各种各样的程序。

2.3 量子中间表示 (QIR)

微软开发了一种新的量子中间表示，称为 Quantum Intermediate Representation（QIR）。它基于广泛使用的开源 LLVM 中间表示。为了推广 QIR 并为高效使用不同的量子处理器提供解决方案，微软成立了 QIR 联盟 [2]。QIR 利用 LLVM 中间表示现有的经典基础设施，避免了为不同量子编程语言创建新编译器的需要，因此无需额外扩展或修改，就能支持量子程序所需的功能。具体而言，QIR 保留了 LLVM 原语，例如 **调用**、**位转换**、**获取元素指针**，以及整数等其他经典数据类型 **i***sz* 和双精度浮点数 **双精度浮点数**。为了实现量子操作，QIR 在 LLVM 中间表示中新增了两种数据类型：**%Qubit** 和 **%Result**，分别表示量子寄存器和测量结果。此外，还有两种数据结构：**%Array** 和 **%Tuple**，分别表示由相同类型数据构成的数组，以及由多种类型数组组成的用户定义结构。上述四种数据类型对于 QIR 都是不透明的。为了更便于表示量子比特的操作，QIR 还新增了诸如**%Pauli**这样的数据类型，它指的是量子力学中的四个泡利矩阵。

除了数据类型，QIR 还声明了与量子相关的运行时函数，以及对量子比特执行操作的函数（例如量子门操作和测量操作）。这些函数并未在 QIR 中实现，需要在外部定义并实现。这种设计便于将 QIR 适配到不同的硬件后端，从而使 QIR 独立于硬件后端。QIR 的官方文档 [15] 包含其指令的定义，我们将简要说明本研究中使用的 QIR 指令。由于 QIR 中声明的量子比特指令都具有相同的前缀@ *quantum rt* 或 @ *quantum qis* ，为节省空间，我们在本研究中采用省略记法（例如，将 @ *quantum rt qubit allocate* 省略并记为*qubit allocate*，将 @ *quantum qis gateop body* 省略并记为*gateop body*）。

图 2 展示了一段 QIR 代码，简要说明了 QIR 的设计。代码第 1 至 3 行标记了不透明数据类型。第 5 至 12 行实现了分配量子比特寄存器并在执行 H 门后对其进行测量的功能。第 7 行调用了 量子比特分配 函数，用于分配量子比特寄存器。对量子比特 执行操作的函数均声明为 门操作主体，其中 门操作 表示要执行的具体操作，例如第 8 行的 *h* 指的是 H 门。

执行第 8 行的 H 门操作后，第 9 行中的测量操作在第 14 至 32 行得到了明确的定义。完成测量后，第 10 行使用量子比特释放 释放量子比特，并在第 11 行返回测量结果。 -

```
1 %Result =type opaque 2 %Qubit =type opaque 3 %Array =type opaque 4 5 define internal %Result*
@Sample__SampleQ__body() {6 entry: 7 %q = 调用 %Qubit*@ __quantum__rt__qubit_分配() 8 调用
void @__quantum__qis__哈达玛门__body(%Qubit* %q) 9 %0 = 调用 %Result*@Microsoft__Quantum__
Intrinsic__测量__body(%Qubit* %q) 10 调用 void @__quantum__rt__qubit_释放(%Qubit* %q) 11 ret %
Result*%0 12 }13 14 define internal %Result* @Microsoft__Quantum__Intrinsic__测量__body(%Qubit* %
qubit) {15 entry: ... 31 ret %Result*%4 32 }33 34 declare %Qubit*@ __quantum__rt__qubit_分配() 35
declare void @__quantum__rt__qubit_释放(%Qubit*) 36 declare void @__quantum__qis__哈达玛门__
body(%Qubit*) 37 declare void @__quantum__rt__array_update_alias_count(%Array*, i32) 38 declare %
Array*@ __quantum__rt__array_create_1d(i32, i64) 39 declare i8*@ __quantum__rt__array_get_
element_ptr_1d(%Array*, i64) 40 declare %Result*@ __quantum__qis__测量__body(%Array*, %
Array*) 41 declare void @__quantum__rt__array_update_reference_count(%Array*, i32)
```

图 2：QIR 示例。

3 方法论

本节中，我们将对 QIR 的语法和语义进行形式化，以支持对不安全代码的验证。由于 QIR 是基于 LLVM 中间表示设计的，本文不考虑经典LLVM中间表示部分的形式化，这可以使用方法 [12, 29] 完成。我们对量子指令程序的形式化基于 QIR 的 0.1版。

3.1 QIR 的语法

我们首先在图 3 中给出 QIR 语法的形式化定义。我们的抽象语法扩展了 Zhou等人 [29], 的工作，他们对 LLVM 中间表示进行了较为完整的形式化。我们主要为其添加了 QIR 特有的数据类型，例如**%Qubit**、**%Result**、**%Array**等。由于 QIR 指令通过 LLVM 的**调用**指令调用，其抽象语义可归属于*id*，因此，关于量子门操作、测量操作等的指令无法体现在抽象语法中。

类型 *typ ::=isz* |双精度浮点数 |*typ** | [*sz* × *typ*]┼*typ typj*┼{*typj*} |*id* |*%Pauli* |*%Range* |*%Result*|*%Qubit* | *%Array* |*%Tuple* 乘积 *prod ::= define typ id (arg)* {*b*} | *declare typ id (arg)* 值 *val ::= id* |*cnst* 常量 *cnst ::= isz Int* | 双精度浮点数 *Double* | *void* | *typ*id* | *typ* [*cnstj j*] | { *cnstj j*} | *extractvalue cnst cnstj j* 获取元素指针 *cnst cnstj j* | 位转换 *cnst to typ* | *%Range Int Int Int* | *%Pauli Int* 块 *b ::= lϕctmn* 终结符 *tmn ::= ret typ val* | *ret void* | *br val l*1 *l*2 | *br l* 命令 *c ::= id* = 位转换 *typ*1*val to typ*2 | *store typ val*1 *val*2 *align* | *option id* = 调用 *typ*0 *val*0 *para m*| *id* = 获取元素指针 *(typ*) val valj* | *id* = *load (typ*) val*1 *align*| *id* = *icmp cond typ val*1 *val*2 | *id* = 分配 *typ val align*

图 3：QIR 的抽象语法。红色字体表示与 LLVM 中间表示相比，QIR 抽象语义中新增的内容。

3.2 QIR 的语义

接下来，我们将介绍对 QIR 语义的形式化。由于 QIR 将指令设计为量子运行时函数的声明，因此会明确指定每条指令输入和输出的类型。这种设计避免了经典LLVM中间表示中的非确定性语义，从而免去了繁琐的类型检查工作，使我们能够专注于形式化检查量子操作的正确性。由于 QIR 中量子操作的实现由后端硬件负责，因此无需在 QIR 中对这一过程进行形式化。量子操作正确性的保证应由后端硬件承担。因此，我们可以假设量子操作是正确的，并进一步提升 QIR 代码的安全性。

3.2.1 分配和释放量子比特

QIR 通过将**%Qubit*** 或由量子比特组成的**%Array***传递给相应函数，来操作后端寄存器中对应的量子比特；它无法直接访问**%Qubit** 或**%Array**的信息（因为这些信息对 QIR 和 LLVM 来说都是不透明的）。这使 QIR 难以管理量子比特的状态，编译器也无法判断量子比特是否已释放，从而带来潜在的安全风险。本节将对单个量子比特及量子比特数组的创建过程进行语义形式化，并提出一套量子比特及量子比特数组的管理方案，以便有效捕捉 QIR 中可能出现的量子比特克隆和使用已释放量子比特等问题。

静态创建。在量子中间表示中，量子比特的分配包括静态和动态两种方式。对于标识符在编译时已可确定的量子比特，可以使用 LLVM 的 **inttoptr**指令生成静态量子比特值。以下示例摘自量子中间表示的规范文档:

%qubit3 =**inttoptr i**32 3 **to %量子比特***

这段代码将 **i**32 数据类型的 3 转换为 **%Qubit*** 类型，从而引用真实设备（或模拟器）中的 qubit3。由于静态创建量子比特的指令只使用经典LLVM中间表示，因此本文不再详述其形式语义。

动态管理。动态量子比特由量子运行时方法分配和释放。为了在 QIR 代码中存储已分配的量子比特和量子比特数组，我们在

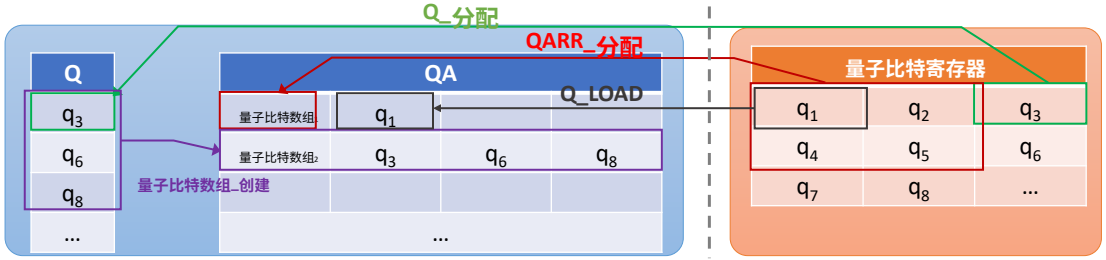

图4：QIR 中量子比特和量子比特数组的管理模型。

框架中引入了两种数据结构，分别是一维数组 **Q** 和二维数组 **问答**。我们的管理模型如图4所示，其中 **Q** 存储每个已分配量子比特的 **%Qubit*** 指针。**问答** 的第一列存储量子比特数组的 **%Array*** 指针，而对应的行则存储数组中每个量子比特的 **%Qubit*** 指针。表1列出了一些用于管理这两种数据结构的方法。

表1：我　们的验证框架中用于管理量子比特和量子比特数组的一些方法　　ork

方法	描述
appqlist(队列, q)	将 q 追加到**队列**。
追加队列数组列表(问答, 量子比特数组)	将 量子比特数组 追加到**问答**。
检查量子比特(队列, q)	检查q 是否在**队列**中。如果是，返回 1；否则，返回 0。
检查量子比特数组列表(问答, 量子比特数组)	检查量子比特数组 是否在**问答**中。如果是，返回 1；否则，返回 0。
删除队列元素(队列，q)	从**队列**中移除q 。
删除队列数组(问答，量子比特数组)	从**问答**中移除一行量子比特数组 。
追加队列数组(问答，量子比特数组，q)	将q 添加到QA 中的**量子比特数组**。如果q 已存在于量子比特数组中，则跳过。
检查队列数组(问答，量子比特数组，q)	检查q 是否在量子比特数组中。如果是，返回 1；否则，返回 0。
查找量子比特数组(问答，q)	返回q 所在的数组；如果该数组不存在，则返回 0。

我们的形式化方法关注 QIR 代码中的两个潜在安全问题：**量子比特的克隆操作**以及**已释放量子比特的使用**。我们提出的管理模型能够有效捕获这两类不安全代码。公式 6 至 9 分别展示了量子比特和量子比特数组分配与释放的操作语义，其中量子比特分配、量子比特数组分配、量子比特释放和量子比特数组释放是 QIR 提供的指令。具体而言，在分配量子比特和量子比特数组后，其返回值（**%Qubit*** 和 **%Array***）会存储在**队列**和**问答** 中（见图4）。释放量子比特和量子比特数组后，相应的值会从**队列** 和**问答**中移除。**为避免使用已释放的量子比特和量子比特数组**，除 Q ALLOC 和 QARR ALLOC 外，我们还在所有规则中加入检查量子比特和量子比特数组是否位于**队列** 和**问答** 中的步骤（使用**检查量子比特** 和**检查量子比特数组列表**）。

尤其是，释放单个量子比特（规则「量子比特释放」）时，我们新增了检查该量子比特是否属于量子比特数组的步骤（使用 **查找量子比特数组**），因为直接释放量子比特数组中的单个量子比特，可能会导致之后释放整个量子比特数组时出错。

$$\frac{q = qubit_allocate() \ \textbf{appqlist}(\mathtt{Q}, q) = \mathtt{Q}'}{\mathtt{Q} \vdash \mathtt{Q}, id = \textbf{call } \%\textbf{Qubit}^* qubit_allocate() \Rightarrow \mathtt{Q}', id \leftarrow q} \ \text{Q_ALLOC} \tag{6}$$

$$\frac{qarray = qubit_allocate_array(n) \ \textbf{appqarrlist}(\mathtt{QA}, qarray) = \mathtt{QA}'}{\mathtt{QA}, n \vdash \mathtt{QA}, id = \textbf{call } \%\textbf{Array}^* qubit_allocate_array(n) \Rightarrow \mathtt{QA}', n, id \leftarrow qarray} \ \text{QARR_ALLOC} \tag{7}$$

$$\frac{\text{if not } qarray = \textbf{findqarr}(\mathtt{QA}, q) \text{ and } \textbf{checkq}(\mathtt{Q}, q) \text{ then } qubit_release(q)\ \textbf{delq}(\mathtt{Q}, q) = \mathtt{Q}' \text{ else } abort}{\mathtt{Q}, \mathtt{QA}, q \vdash \mathtt{Q}, \mathtt{QA}, \textbf{call void } qubit_release(q) \Rightarrow \mathtt{Q}', \mathtt{QA}, q}\ \text{Q_DEALLOC} \quad (8)$$

如果 **检查量子比特数组列表(问答,**量子比特数组) 那么 量子比特数组释放(量子比特数组) **删除量子比特数组(问答,**量子比特数组) = **问答** 否则 **中止 问答,**量子比特数组 ⊢ **问答,调用 void** 量子比特数组释放(量子比特数组) ⇒ **问答',**量子比特数组 QARR DEALLOC

(9)

3.2.2 Mani在量子比特数组中操作量子比特，以及使用已分配的量子比特构建量子比特数组 its

除了使用 QIR 的方法直接分配量子比特和量子比特数组外，还可以从量子比特数组中加载单个量子比特，或将多个已分配的量子比特组合成新的量子比特数组。我们分别用以下函数概述这两种操作：加载量子比特 和 创建量子比特数组：

$$(id = \text{加载量子比特}(\text{量子比特数组}, n)) = \{id_1 = \textbf{调用 i}sz^* \textit{ array get element ptr 1}d(\text{量子比特数组}, n);$$
$$id_2 = \textbf{位转换 i}sz^*\ id_1 \textbf{ to \%Qubit}^{**};$$
$$\text{标识符} = (\text{标识符}_3 = \textbf{加载 \%Qubit}^*, \textbf{\%Qubit}^{**}\ \text{标识符}_2, \text{对齐})\}$$

$$(id = \text{创建量子比特数组}(q_1, q_2, ..., q_n)) = \{$$
$$id = (id_1 = \textbf{调用 \%Array}^* \text{数组创建 1维}(sz, n));$$
$$\overline{id_2}^{i} = (\textbf{调用 i}sz^* \text{数组获取元素指针 1维}(qarray, i)); id_{3i} = \textbf{位转换 i}sz^* id_{2i} \textbf{转换为 \%Qubit}^{**}, {}^{i=1...n}$$
$$\textbf{存储 \%Qubit}^* q_i, \textbf{\%Qubit}^{**} id_{3i}, \text{对齐}; \}$$

在上述两个函数中，数组创建1维 和_数组获取元素指针 1维 是 QIR 中的指令。数组创建 1维 会返回一个长度为 n 、大小为 sz 的数组指针 **%Array*** ，其输入参数为数组元素的字节大小 sz和数组长度 n。数组获取元素指针 1维 可通过输入量子比特数组 *qarray* 和索引 n ，获取相应元素的 **i**sz^* 指针。然后，通过 LLVM 的**位转换**指令，可将其转换为 **%Qubit**** 指针，并可使用**加载** 或 **存储** 指令加载或存储量子比特指针 **%Qubit*** 。

操作量子比特数组中的量子比特。规则"量子比特加载"定义了从量子比特数组加载量子比特的语义。具体而言，除了在执行加载操作之前需要检查该量子比特数组是否已释放外，还需要在执行结束时，将该量子比特记录到列表**问答**中对应的量子比特数组内（见图4）。

$$\frac{\begin{array}{l}\text{若检查量子比特数组列表}(QA, qarray),\ \text{则 } ptr = (id_1 = \text{调用 } isz^* \text{一维数组元素指针获取}\\ (qarray, n))\ qptr = (id_2 = \text{位转换 } isz^*\ ptr \text{ to \%Qubit}^{**})\ q = (id_3 = \text{加载 \%Qubit}^*, \text{\%Qubit}^{**}\\ qptr, align) \text{ 追加队列数组}(QA, qarray, q) = QA' \text{ 否则中止 } QA, qarray, n \vdash QA, id = \text{加载量子比特}\end{array}}{(qarray, n) \Rightarrow QA', qarray, n, id \leftarrow q \text{ 量子比特加载}} \quad (10)$$

用已分配的量子比特构建量子比特数组。 规则 QARR CREATE 概述了由已分配的量子比特构建量子比特数组的函数语义。我们在存储指令之前，加入检查待存储的量子比特是否已存在于数组中的过程（**检查队列数组**）；如果该量子比特已存在于数组中，则中止操作。这是因为同一个量子比特数组中不应有两个相同的量子比特，否则就会**存在量子比特克隆的风险**。如果将此量子比特数组用作控制量子比特，两个相同的量子比特就会同时充当控制量子比特。虽然这不会影响操作的

结果，但违反了量子计算的不可克隆原理。图4展示了对**Q** 和**QA** 的管理。具体而言，规则 QARR CREATE 会将新的数组指针添加到**QA** 中，并将构成该数组的量子比特添加到相应的行。

如果 $checkq(Q,q_i)$ 或 $qarray_{temp} = findqarr(QA,q_i)_{i=1...n}$，则 $qarray = (id_1 =$ 调用 $\%Array^*$一维数组创建$(sz,n))$ $appqarrlist(QA,qarray)$ $ptr_i = (id_{2i} =$(调用 isz^* 一维数组元素指针获取$(qarray,i)))$ $qptr_i = (id_{3i} =$ 位转换 isz^* ptr_i 为 $\%Qubit^{**})_{i=1...n}$如果 $not\ checkqarr(QA,qarray,q_i)$，则存储 $\%Qubit^*\ q_i$,$\%Qubit^{**}\ qptr_i$,$align$；否则中止$_{i=1...n}$ $appqarr(QA,qarray,q_i)^{1=1...n} = QA'$；否则中止 $Q,QA,(q_1,q2,...,q_n) \vdash Q,QA,id =$ 创建量子比特数组$(q_1,q2,...,q_n) \Rightarrow Q,QA',(q_1,q2,...,q_n),id \leftarrow$ 量子比特数组 $QARR\ CREATE$

(11)

3.2.3 门操作

在 QIR 中，门操作可分为两大类：单量子比特门操作，以及通过为单量子比特门操作添加控制量子比特而形成的受控门操作。我们将分别形式化这两类门操作的语义。在 QIR 中，量子比特的门操作不会改变我们针对 **Q** 和**QA** 的管理模型，因为它们不涉及数据的加载与存储，也不涉及量子比特的分配与释放。规则 SG OP 和 CG OP 分别展示了我们对单量子比特门和受控门语义的形式化，其中*gateop body* 和*gateop ctl* 是 QIR 中的门操作指令，而*gateop* 是单量子比特门，例如x、Rx、h。

$$\frac{\text{if } \mathbf{checkq}(\mathbf{Q},q) \text{ or } qarray_{temp} = \mathbf{findqarr}(\mathbf{QA},q) \text{ then } gateop_body(option\ d,q) \text{ else } abort}{\mathbf{Q},\mathbf{QA},option\ d,q \vdash \mathbf{Q},\mathbf{QA},\mathbf{call\ void}\ gateop_body(option\ d,q)\ \Rightarrow \mathbf{Q},\mathbf{QA},option\ d,q}\ \text{SG_OP} \tag{12}$$

$$\frac{\text{如果 (}\mathbf{检查量子比特}(\mathbf{Q},q) \text{ 或 量子比特数组}_{temp} = \mathbf{查找量子比特数组}(\mathbf{QA},q)) \text{ 且 } \mathbf{检查队列数组}(\mathbf{QA},qarray) \text{且非 } \mathbf{检查队列数组}(\mathbf{QA},qarray,q) \text{ 则 } gateop\ ctl(qarray,(\text{选项 } d,q)) \text{ 否则 中止 } \mathbf{Q},\mathbf{QA},\text{选项 } d,q \vdash \mathbf{Q},\mathbf{QA},\text{选项 } d,q\ \mathbf{调用\ void}\ gateop\ ctl(qarray,(\text{选项 } d,q)) \Rightarrow \mathbf{Q},\mathbf{QA},\text{选项 } d,q}{\text{CG OP}} \tag{13}$$

在量子中间表示中，控制量子比特以量子比特数组的形式输入指令。因此，类似于 QARR CREATE 规则，对于 CG OP，我们也会使用**检查队列数组**检查目标量子比特是否存在于控制量子比特数组中，以避免量子比特克隆问题。

3.2.4 测量

对于测量操作，可以通过输入一个泡利值数组和一个量子比特数组，在量子中间表示中返回测量结果。测量操作的语义为：

$$\frac{\text{如果 } \mathbf{检查队列数组}(\mathbf{问答},\text{量子比特数组}) \text{ 则 结果 } = \text{测量主体}(\text{泡利数组},\text{量子比特数组}) \text{ 否则 中止}}{\mathbf{问答},\text{泡利数组},\text{量子比特数组} \vdash \text{标识 } = \mathbf{调用\ \%Result}^* \text{ 测量主体}(\text{泡利数组},\text{量子比特数组}) \Rightarrow \mathbf{问答},\text{泡利数组},\text{量子比特数组},\text{标识} \leftarrow \text{结果}}\ \text{测量} \tag{14}$$

4 量子中间表示不安全代码验证

本节将我们的方法应用于验证一些不安全的 QIR 代码。由于 QIR 尚未正式投入使用，目前生成 QIR 代码最主要的方式是使用微软提供的 Q# 编译器。这启发我们收集验证样例。如果一个不安全的 Q#

程序通过了编译器检查，但在运行时失败，我们就可以从中生成相应的 QIR 代码以供验证。图5展示了两个引用自 [23]，的不安全 Q# 代码样例：在图 5a 中，函数 *NewQubit* 返回的量子比特会在执行结束时被释放；在图 5b 中，*CCNOT* 调用的三个量子比特实际上是同一个量子比特，这违反了不可克隆定理。在接下来的小节中，我们将展示相应的 QIR 代码，并应用我们设计的语义来检查其中的错误。

```
EntryPoint()
操作 Deadqubit():                    测量结果 {
    令 q = NewQubit();
    哈达玛门(q);
    返回 测量(q);
}

操作 NewQubit():                     量子比特 {
    使用 q = 量子比特();
    返回 q;
}
```

(a) 返回一个已释放的量子比特。

```
EntryPoint()
操作 克隆():                                测量结果 {
    使用 q1 = 量子比特();
    令 q2 = q1;
    let q3 = q2;
    CCNOT(q1, q2, q3);
    return M(q3);
}
```

（b）控制量子比特与目标量子比特相同one.

图5：不安全 Q# 代码示例。

4.1 使用已释放量子比特

图 7 展示了由图 5a 中的 Q# 代码转换得到的 QIR 代码。在此示例中，对于第 3 行函数 *@NewQubit body* 的返回值，我们可以使用规则量子比特分配和量子比特释放进行分析，-从而得知返回的 %*q* 已被释放（见图 6a）。执行第 4 行的 *h body* 时，由于 %*q* 不存在于**队列**中，因此应用规则 SG ŌP 中断程序，从而避免使用已释放量子比特所导致的错误。

4.2 量子比特克隆

图 8 展示了图5b 中克隆量子态经过代码转换后得到的 QIR 代码。在示例中，对于第 4 行的函数，*CCNOT body* 将两个相同的 **量子比特*** %*q*1 作为控制量子比特输入，并将 %*q*1 本身作为目标量子比特。因此，对于代码第 12 至 18 行，可以应用 QARR CREATE 规则，从而在第 18 行中断程序，因为转移到数组 % 控制量子比特数组的内容存储了相同的 **量子比特*** %*q*1，避免了量子比特克隆问题（见图 6b）。除此之外，对于_ --

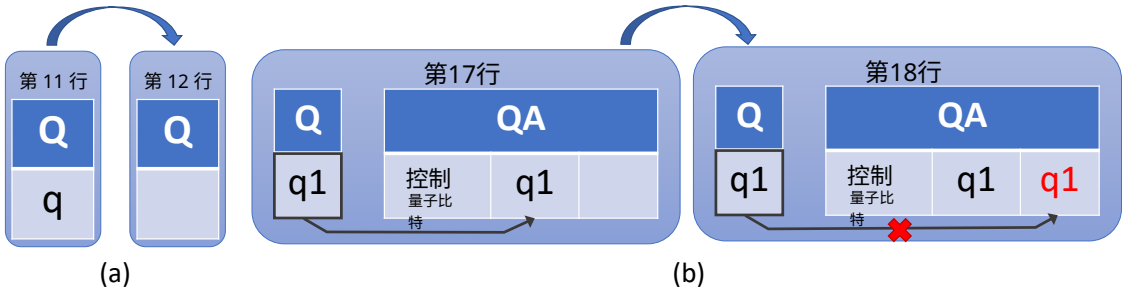

图6：应用形式化方法验证不安全的 QIR 代码示例。

```
1 定义内部 %测量结果* @Deadqubit__body() {2 入口: 3 %q = 调用 %量子比特*@NewQubit__
body() 4 调用 void @__量子__量子指令集__哈达玛门__body(%量子比特* %q) 5 %0 = 调用 %测
量结果*@微软__量子__内置__M__body(%量子比特* %q) 6 返回 %测量结果*%07 }8 9 定义内部
% 量子比特 *@NewQubit__body() {10 入口: 11 %q = 调用 % 量子比特 *@ __量子 __ 运行
时__量子比特_分配() 12 调用 void @__量子__运行时__量子比特_释放(%量子比特* %q) 13 返回 %
量子比特*%q14 }...
```

图 7：由图 5a 转换而来的 QIR 代码。为节省空间，我们仅截取了最关键的最小代码片段。

在代码第 20 行，可以应用 CG_OP 规则；由于作为目标量子比特的 $\%q1$ 与控制量子比特数组中的元素相同，因此此步骤也可以避免量子比特克隆。

5 相关工作

本节将讨论经典和量子编程语言中间表示领域的一些相关工作。

5.1 LLVM 中间表示语义

作为 QIR 的基础，LLVM 中间表示的形式化方法是我们工作的重要参考。Zhao等人 [29] 提出了 Vellvm（经过验证的 LLVM）框架，该框架对 LLVM 中间表示的静态语义、内存模型以及若干操作语义进行了形式化。该框架使用 Coq 实现，由此可以直接提取经过验证且可信度高的可执行代码；此外，SoftBound [17] 案例验证了 Vellvm 的有效性。Li 和 Gunter [12]设计了 K-LLVM，即完整的 LLVM 中间表示语义，其中包括所有 LLVM 中间表示指令、LLVM 文档中的内在函数等。与着重将 LLVM 语义形式化为数学对象的 Vellvm 相比，K-LLVM 以更直接的方式展示了在虚拟计算机中实现这些语义的可能方法。K-LLVM 由 $\mathbb{K}$ [16] 实现，其有效性通过针对单元测试程序以及实际 LLVM 中间表示程序进行测试得到了验证。然而，上述工作旨在将 LLVM 中间表示形式化，并未涉及量子编程语言或中间表示。

5.2 形式化 Quantum 中间表示与编程语言

作为与我们最相关的工作，Hietala等人 [9] 提出了 VOQC，这是一种经过验证的量子电路优化器。作为 VOQC 的输入，研究人员开发了一种小型量子中间表示（SQIR），用于表示量子电路并支持对量子电路优化进行验证。SQIR 经过了良好的形式化，其量子电路语法和语义能够保证 VOQC 优化的正确性。我们的工作与 SQIR 主要有两点不同：首先，形式化对象是微软开发的 QIR；其次，其应用场景是为量子提供一种通用解决方案

```
1 定义内部 %Result* @克隆__主体() {2 入口：3 %q1 = 调用 %Qubit* @ __量子__rt__量子比特_分配() 4 调用 void @微软__Quantum__Intrinsic__CCNOT__主体(%Qubit* %q1, %Qubit* %q1, %Qubit* %q1) ...8 }9 10 定义内部 void @微软__Quantum__Intrinsic__CCNOT__主体(%Qubit* %control1, %Qubit* %control2, %Qubit* %target) {11 入口：12 %__控制量子比特数组__ = 调用 %Array* @ __量子__rt__数组_创建_1d(i32 8, i64 2) 13 %0 = 调用 i8* @ __量子__rt__数组_获取_元素_ptr_1d(%Array* % __控制量子比特数组__, i64 0) 14 %1 = 位转换 i8* %0 to %Qubit** 15 %2 = 调用 i8* @ __量子__rt__数组_获取_元素_ptr_1d(%Array* % __控制量子比特数组__, i64 1) 16 %3 = 位转换 i8* %2 to %Qubit** 17 存储 %Qubit* %control1, %Qubit**%1, 对齐 8 18 存储 %Qubit* %control2, %Qubit**%3, 对齐 8 19 调用 void @__量子__rt__数组_更新_别名_计数(%Array* % __控制量子比特数组__, i32 1) 20 调用 void @__量子__qis__x__ctl(%Array* % __控制量子比特数组__, %Qubit* %target) ...24 }
```

图 8：由图 5a 转换而来的 QIR 代码。为节省空间，我们仅截取了最关键的最小代码片段。

编程语言和后端硬件。另一方面，SQIR 是一种独立开发的量子中间表示，主要用于 VOQC 的验证过程。其次，两者的目的不同。在我们的工作中，形式化 QIR 的主要目标是为量子程序的正确性以及 QIR 的行为提供保证，而 SQIR 则侧重量子电路优化正确性的保证。

关于量子编程语言的形式化，Singhal 等人 [23] 提出了 $\lambda_{Q\#}$，即 Q# 的理想化版本。基于 Staton 的工作 [24], ，他们为 $\lambda_{Q\#}$ 提供了句法和语义形式化，使其能够确保量子不可克隆定理，并提供类似栈的量子比特管理。通过将 Q# 转换为 $\lambda_{Q\#}$，可以为 Q# 代码提供额外的安全保证。对于 Quipper [8] 编程语言，Mahmoud 和 Felty [13]提出了 Proto-Quipper，它包含 Quipper 的核心功能，并基于线性规范逻辑（SL）扩展了 Quipper 的系统。他们还通过 Hybird [6], 实现了 Proto-Quipper 的形式化，对完整的 Proto-Quipper 规范进行了编码，并证明了其类型系统的正确性。尽管这些工作可以为我们的研究提供有价值的参考，但必须认识到我们的工作与它们存在显著差异；我们的研究聚焦于量子中间表示，而非量子编程语言。

6 结语

本文对 QIR 的核心功能进行了形式化，尤其是其抽象语法以及若干运行时函数的语义。我们的形式化确保 QIR 程序遵循量子不可克隆定理，并避免调用已释放的量子比特和数组。基于对真实 QIR 代码的分析，我们还展示了这种形式化在捕获程序中的不安全代码方面的有效性。

我们目前的工作只是对 QIR 规范进行形式化的初步尝试，还需要进一步完善。目前已形式化的量子中间表示函数只占很小一部分。还需形式化更多

函数，例如数据类型中的 元组 和字符串 、完整测量过程的语义，以及对量子比特数组执行批量门操作。从实践角度来看，若要将我们的形式化方法直接应用于 QIR 程序，以实现自动验证（而非像本工作中那样通过人工分析），我们需要在后续工作中借助 Coq [4] 等交互式形式化定理证明器来实现这一形式化方法。

参考文献

[1] Gadi Aleksandrowicz、Thomas Alexander、Panagiotis Barkoutsos、Luciano Bello、Yael Ben-Haim、David Bucher、F Jose Cabrera-Hern´andez、Jorge Carballo-Franquis、Adrian Chen、Chun-Fu Chen 等（2019）：Qiskit：一个用于量子计算的开源框架。访问日期：3月16日。[2] QIR 联盟（2022）：QIR 联盟。网址：https://www.qir-alliance.org/。[3] Benjamin Bichsel、Maximilian Baader、Timon Gehr 和 Martin Vechev：Silq：一种支持安全反计算且语义直观的高级量子语言。载于：第 41 届 ACM SIGPLAN 编程语言设计与实现会议论文集。[4] Coq 开发团队（2017）：《Coq 证明助手参考手册》，8.7版。网址：http://coq.inria.fr。[5] Andrew W. Cross、Lev S. Bishop、John A. Smolin 和 Jay M. Gambetta（2017）：开放量子汇编语言，doi:10.48550/ARXIV.1707.03429。网址：https://arxiv.org/abs/1707.03429。[6] Amy P. Felty 和 Alberto Momigliano（2008）：Hybrid：一种用于高阶抽象语法推理的定义式双层方法。CoRR abs/0811.4367。arXiv:0811.4367。[7]Alan Geller（2020）：介绍量子中间表示（QIR）。网址：https://devblogs.microsoft.com/qsharp/ introducing-quantum-intermediate-representation-qir/。[8] Alexander S Green、Peter LeFanu Lumsdaine、Neil J Ross、Peter Selinger 和 Benoˆıt Valiron（2013）：Quipper：一种可扩展的量子编程语言。载于：第 34 届 ACM SIGPLAN 编程语言设计与实现会议论文集，第 333–342 页。[9] Kesha Hietala、Robert Rand、Shih-Han Hung、Xiaodi Wu 和 Michael Hicks（2021）：量子电路的经验证优化器。《ACM 程序设计语言会议论文集》5(POPL)，doi:10.1145/3434318。网址：https: //doi.org/10.1145/3434318。[10] Chris Lattner 和 Vikram Adve（2004）：LLVM：用于程序全生命周期分析与变换的编译框架。美国加利福尼亚州圣何塞，第 75–88 页。[11] Chris Lattner、Mehdi Amini、Uday Bondhugula、Albert Cohen、Andy Davis、Jacques Pienaar、River Riddle、Tatiana Shpeisman、Nicolas Vasilache 和 Oleksandr Zinenko（2021）：MLIR：扩展领域专用计算的编译器基础设施。载于：2021 年 IEEE/ACM 代码生成与优化国际研讨会（CGO），第 2–14 页，doi:10.1109/CGO51591.2021.9370308。[12] Liyi Li 和 Elsa L. Gunter（2020）：K-LLVM：LLVM 中间表示的一种相对完备语义。载于 Robert Hirschfeld 和 Tobias Pape 编：第 34 届欧洲面向对象编程会议（ECOOP 2020），莱布尼茨国际信息学会议录（LIPIcs）166，德国达格施图尔，Schloss Dagstuhl–Leibniz-Zentrum f¨ur Informatik，第 7:1–7:29 页，doi:10.4230/LIPIcs.ECOOP.2020.7。网址：https: //drops.dagstuhl.de/opus/volltexte/2020/13164。[13] Mohamed Yousri Mahmoud 和 Amy P. Felty（2018）：用线性逻辑形式化 Quipper 量子编程语言的元理论。CoRR abs/1812.03624。arXiv:1812.03624。[14] Alexander McCaskey 和 Thien Nguyen（2021）：用于量子汇编语言的 MLIR 方言，doi:10.48550/ARXIV.2101.11365。网址：https://arxiv.org/abs/2101.11365。[15] 微软（2022）：量子中间表示（QIR）。网址：https://github.com/qir-alliance/qir-spec/tree/main/specification。

[16] D. Monderer 和 M. Tennenholtz（2004）：K-实现。《人工智能研究期刊》21，第 37–62 页，doi:10.1613/jair.1231。网址：https://doi.org/10.1613%2Fjair.1231。[17] Santosh Nagarakatte、Jianzhou Zhao、Milo M.K. Martin 和 Steve Zdancewic（2009）：SoftBound：针对 c 的高度兼容且完备的空间内存安全。载于：第 30 届 ACM SIGPLAN 编程语言设计与实现会议论文集，PLDI '09，计算机协会，美国纽约州纽约，第 245–258 页，doi:10.1145/1542476.1542504。网址：https: // doi.org/10.1145/1542476.1542504。[18]Santosh Nagarakatte、Jianzhou Zhao、Milo M.K. Martin 和 Steve Zdancewic（2009）：SoftBound：针对 c 的高度兼容且完备的空间内存安全。SIGPLAN 通讯 44(6)，第 245–258 页，doi:10.1145/1543135.1542504。网址：https://doi.org/10.1145/1543135.1542504。[19]Thien Nguyen、Anthony Santana、Tyler Kharazi、Daniel Claudino、Hal Finkel 和 Alexander Mc- Caskey（2020）：扩展 C++ 以支持异构量子-经典计算。arXiv 电子预印本：arXiv:2010.03935。arXiv:2010.03935。[20] Michael A Nielsen 和 Isaac Chuang（2002）：量子计算与量子信息。[21] John Preskill（2018）：NISQ 时代及之后的量子计算。Quantum 2，第 79 页。[22] IBM 研究院：Qiskit。访问日期：2021年6月。网址：https://qiskit.org。[23] Kartik Singhal、Kesha Hietala、Sarah Marshall 和 Robert Rand（2022）：Q# 作为量子算法语言，doi:10.48550/ARXIV.2206.03532。网址：https://arxiv.org/abs/2206.03532。[24] Sam Staton（2015）：代数效应、线性性与量子编程语言。SIGPLAN 通讯 50(1)，第 395–406 页，doi:10.1145/2775051.2676999。网址：https://doi.org/10.1145/ 2775051.2676999。[25] Sam Staton（2015）：代数效应、线性性与量子编程语言。载于：第 42 届 ACM SIGPLAN-SIGACT 编程语言原理研讨会年会论文集，POPL '15，计算机协会，美国纽约州纽约，第 395–406 页，doi:10.1145/2676726.2676999。网址：https://doi.org/10.1145/2676726.2676999。[26] Krysta Svore、Alan Geller、Matthias Troyer、John Azariah、Christopher Granade、Bettina Heim、Vadym Kli- uchnikov、Mariia Mykhailova、Andres Paz 和 Martin Roetteler（2018）：Q#：通过高级领域特定语言实现可扩展量子计算与开发。载于：2018 年现实世界领域特定语言研讨会论文集，第 1–10 页。[27] Google AI Quantum 团队（ 2018 ）：Cirq。网址：https://github.com/quantumlib/Cirq。[28] ProjectQ 团队（2017）：ProjectQ。访问日期：2020年4月。网址：https://projectq.ch/。[29] Jianzhou Zhao、Santosh Nagarakatte、Milo M.K. Martin 和 Steve Zdancewic （ 2012 ）：形式化 LLVM 中间表示以实现经验证的程序变换。载于：第 39 届 ACM SIGPLAN-SIGACT 编程语言原理研讨会年会论文集，POPL '12，计算机协会，美国纽约州纽约，第 427–440 页，doi:10.1145/2103656.2103709。网址：https://doi.org/10.1145/2103656.2103709。