



# **Compilation of Quantum Programs with Control Flow Primitives in Superposition**

**Master Thesis**

**05.02.25**

**Sascha Thiemann**

**Supervisors: apl. Prof. Dr. Thomas Noll**

**Prof. Dr. rer. nat. Dominique Unruh**

# Outline

---

## Introduction

## Background

- Quantum Control Flow
- Quantum Control Machine

## Language

- Overview
- Syntax
- Translation

## Implementation

- Lexical and Syntactic Analysis
- Semantic Analysis
- Code Generation
- Optimization

## Evaluation

## Conclusion

---

## Introduction

- Idea of Quantum control flow first used by [Altenkirch and Grattage, 2005].
  - Later, it was defined by [Ying et al., 2012].
  - Most languages focus on classical control flow.
  - Only recently, Quantum Control Machine, proposed by [Yuan et al., 2024].
    - An instruction set architecture.
    - Allows for quantum control flow with an assembly-like syntax.
    - Low-level design of the architecture and syntax → complex programs.
- ⇒ Build upon the ideas, define a high-level quantum language with quantum control flow.
- ⇒ Plan to implement a compiler for it.

## Quantum Control Flow

- First used by [Altenkirch and Grattage, 2005] to define functional programming language.
- For example, used to define Hadamard gate as function *had*:

$$\begin{aligned} \textit{had} &: Q \rightarrow Q \\ \textit{had} : x &\mapsto \text{if}^\circ x \\ &\quad \text{then } \{ \textit{false} \mid \neg \textit{true} \} \\ &\quad \text{else } \{ \textit{false} \mid \textit{true} \} \end{aligned}$$

- Later, formally defined by [Ying et al., 2012].
- Allows for the execution of functions based on values in superposition.
- Results in superposition of the results of individual executions.

## Limitations — Reversibility

- Mainly limited by two principles: *reversibility* and *synchronization*.
  - Instructions required to be reversible, as they are unitary transformations.
- ⇒ Control flow, as implemented in classical computers, not possible.
- Landauer Embedding [Landauer, 1961] seems to offer solution.
  - Can make non-reversible function  $f : D \rightarrow D'$  reversible by outputting input value.
  - Result is  $g : D \rightarrow D' \times D$  with  $g(x) = (f(x), x)$ .
  - However, output includes program history and the result depends on the history, they become entangled.
  - Leads to disruptive entanglement, invalid results [Yuan et al., 2024].

## Limitations — Synchronization

- Program counter may become entangled with data.
- ⇒ Disruptive entanglement.
- The principle of synchronization: Control flow must become independent from the data.
  - Program may include padding operations to ensure synchronization.
- ⇒ Loops cannot depend solely on value in superposition.
- ⇒ Instead, a loop must be bounded by classical value [Yuan et al., 2024].

## Quantum Control Machine

- Quantum Control Machine (QCM), proposed by [Yuan et al., 2024]
- Focused on quantum control flow.
- Syntax and logic similar to classical assembly language
- Employs a branch control register *bcr* to enable reversible jumps.
- Instead of increasing IP by 1 after statement, increased by value of *bcr*.
- The *bcr* can then be reversibly modified.
  - To jump by 5, the *bcr* increased by 5
  - At its destination, decreased by 5 again.

# Background

---

## Instructions

- Here, some instructions of the QCM are listed.
- For every instruction there exists a reversed instruction, e.g., `radd` is the subtraction operation.

| Operation         | Syntax                          | Semantics <sup>1</sup>  |
|-------------------|---------------------------------|---|
| No-op             | <code>nop</code>                | Only increases instruction pointer by the <i>bcr</i> .                    |
| Addition          | <code>add <i>ra rb</i></code>   | Adds register <i>rb</i> to <i>ra</i> .                                    |
| Multiplication    | <code>mul <i>ra rb</i></code>   | Multiplies register <i>ra</i> by <i>rb</i> .                              |
| Jump              | <code>jmp <i>p</i></code>       | Increases <i>bcr</i> by <i>p</i> .  |
| Conditional Jumps | <code>jz <i>p ra</i></code>     | Increases <i>bcr</i> by <i>p</i> if <i>ra</i> is 0.                       |
|                   | <code>jne <i>p ra rb</i></code> | Increases <i>bcr</i> by <i>p</i> if <i>ra</i> is not equal to <i>rb</i> . |

<sup>1</sup> After all operations, the instruction pointer is increased by the value of the *bcr*.

An excerpt of the QCM instruction set with instructions used in later examples.



# Background

## (Non-) Reversible Example

- Example of a classical program and the reversible equivalent.
- Both programs calculate  $x^y$ .
- Neither are synchronized because loop is not bound by classical value.

```
1      add    res $1
2      add    r1  y
3  l1:  jz     l2  r1
4      mul    res x
5      radd   r1  $1
6      jmp    l1
7  l2:  nop
```

A non-reversible exponentiation algorithm.

```
1      add    res $1
2      add    r1  y
3  l1:  rjne   r11 r1  y
4  r12:  jz     l2  r1
5      mul    res x
6      radd   r1  $1
7  r11:  jmp    l1
8  l2:  rjmp   l2
```

A reversible exponentiation algorithm.

# Background

---

## Reversible Synchronized Example

- Synchronized implementation, calculating  $x^{\min y, \max}$ .
- *max* is a classical bound for loop.
- Line 9 includes padding instruction.

```
1      add    res    $1
2      add    r1     max
3  l1:    rjne   r11   r1    max
4  r12:    jz     l2     r1
5  r13:    jg     l3     r1    y
6      mul    res    x
7  r14:    jmp    l4
8  l3:    rjmp   r13
9      nop                    ; padding
10 l4:    rjle   r14    r1    y
11      radd   r1     $1
12 r11:    jmp    l1
13 l2:    rjmp   r12
```

A synchronized, reversible exponentiation algorithm.

## Language Overview

- Idea for language: Provide high-level language with the capabilities of the QCM.
- Want to remove low-level concepts, add high-level ones.
- Jump instructions in superposition are removed → need to add other control flow primitives.
- Introduce multiple high-level concepts and control flow statements:
  - blocks and scopes,
  - different data types,
  - composite gates,
  - loop statements, unrolled at compile time, and
  - quantum if- and else-statements.

# Language

## Syntax

- Define  $CFG_{Luie}$  for our language.
- Start symbol is the program, consisting of gate declarations and a block.
- A block is a list of translatables, either statements or declarations.

$$\begin{aligned} CFG_{Luie} &= (V_{Luie}, \Sigma_{Luie}, R_{Luie}, prg_{Luie}) \\ V_{Luie} &= \{exp, rExp, gate, prg_{Luie}, \dots\} \\ \Sigma_{Luie} &= \{.., range, (, ), \dots\} \end{aligned}$$

$$\begin{aligned} Program : prg_{Luie} & ::= gDcl_1 \dots gDcl_n blk \mid \\ & blk \\ Block : blk & ::= t_1 \dots t_n \mid \\ & \epsilon \end{aligned}$$

```
1 /* Gate Declaration */
2 gate c_h_reg(control, reg) do
3     qif control do
4         for i in range(sizeof(reg)) do
5             h reg[i];
6         end
7     end
8 end
9
10 /* Block Start */
11 const regSize : int = 3; // Decl.
12 qubit c; // Decl.
13 qubit[regSize] a; // Decl.
14 c_h_reg c, a;
15 /* Block End */
```

An example Luie program.

# Language

## Syntax

- Three different statements: quantum if-statement, a loop statement, gate application.
- To combine qubit or register access: qubit argument.
- Expressions for numeric values or ranges.
- Fix set of defined gates  $\rightarrow$  differentiate translations.

*Statement* : *stm* ::= *qif* *qArg* *do blk end* |  
                  *for* *id* *in rExp* *do blk end* |  
                  *id* *qArg*<sub>1</sub>, ..., *qArg*<sub>*n*</sub>;  
*QubitArg* : *qArg* ::= *id* | *id*[*exp*]

```
1 gate c_h_reg(control, reg) do
2   /* If-Statement */
3   qif control do
4     /* Loop Stat. |  $\nabla$  Expression */
5     for i in range(sizeof(reg)) do
6       /*  $\uparrow$  Range Exp. */
7       h reg[i];
8     end
9   end
10 end
11
12 const regSize : int = 3;
13 qubit c;
14 qubit[regSize] a;
15 c_h_reg c, a; // Gate Application
16   /*  $\uparrow$   $\uparrow$  Qubit Argument */
```

An example Luise program.

*ConstGates* = {h, x, y, z, cx, ccx}

## Symbol Table

- Saves symbol information relevant for translation.
- Contains four types of symbols:
  1. named constants,
  2. quantum registers and qubits,
  3. qubit arguments, and
  4. composite gates.

$$\begin{aligned} \textit{SymbolTable} := \{st \mid st : \textit{Identifier} \dashrightarrow & (\{\textit{const}\} \times \mathbb{Q}) \\ & \cup (\{\textit{qubit}\} \times \mathbb{N} \times \textit{Identifier}) \\ & \cup (\{\textit{arg}\} \times \textit{QubitArgument}) \\ & \cup (\{\textit{gate}\} \times \textit{Block} \times \textit{Identifier}^+)\} \end{aligned}$$

## Translation Function and Block Translation

- *trans* function translates program.
- Symbol table updated with gate declaration by *up* function.
- Block translation *bt* translates all translatable.
- Updates symbol table used in next translation.

$$st_1 = up(st_e, \text{gate } c\_h\_reg \dots) = [c\_h\_reg \mapsto (\text{gate}, \text{qif control do } \dots \text{ end, control, reg})]$$
$$\begin{aligned} bt(\text{const } regSize = 3; \dots, st_1) &= tr_1 && \text{where } (tr_1, st_2) = tt(\text{const } regSize = 3; , st_1) \\ &tr_2 && \text{where } (tr_2, st_3) = tt(\text{qubit } c; , st_2) \\ &tr_3 && \text{where } (tr_3, st_4) = tt(\text{qubit}[regSize] a; , st_3) \\ &tr_4 && \text{where } (tr_4, -) = tt(c\_h\_reg c, a; , st_4) \end{aligned}$$

## Translatable and Declaration Translation

- *tt* translates each translatable, differentiates between declaration and command.
  - Declarations translated by *dt*.
  - Update symbol table  $\rightarrow$  function returns (updated) symbol table.
  - Language allows for different variable scopes  $\rightarrow$  same identifier can be used multiple times.
- $\Rightarrow$  Unique identifier *uid* is generated for translation.

$$tt(c\_h\_reg\ c,\ a;\ ,st) = (ct(c\_h\_reg\ c,\ a;\ ,st), st)$$

$$tt(qubit\ c;\ ,st) = dt(qubit\ c;\ ,st)$$

$$dt(qubit\ c;\ ,st) = (qubit\ uid_c;\ ,up(st, qubit\ c;\ ))$$

$$up(st, qubit\ c;\ ) = st[c \mapsto (qubit, 1, uid_c)]$$



## Command Translation

- Commands are translated with *ct*.
- Argument translation *qt* used to differentiate between qubits and register accesses, get *uid*.
- *control* adds the translated *qArg* as guard to all gate applications in translation.

$$st = [c \mapsto (\text{qubit}, 1, uid_c), \\ a \mapsto (\text{qubit}, 3, uid_a)]$$

$$ct(\text{if } c \text{ do } x \ a[0]; \text{ end}, st) = control(qt(c, st), bt(x \ a[0];, st)) \\ = control(uid_c, x \ uid_a[0];)$$

$$qt(c, st) = st[c]_3 = uid_c \\ bt(x \ a[0];, st) = x \ qt(a[0]); = x \ uid_a[0];$$

$$control(uid_c, x \ uid_a[0];) = ctrl(1) \ @ \ x \ uid_c, uid_a;$$

# Implementation

---

## Overview

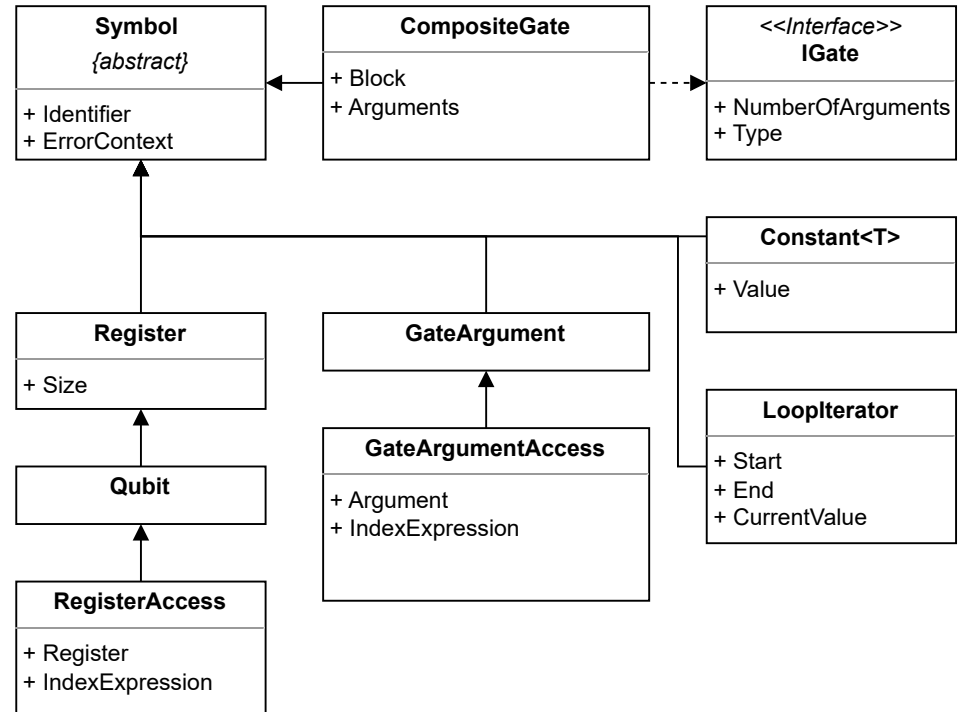
- The implementation of the compiler is differentiated into four different stages:
  1. the lexical and syntactic analysis,
  2. semantic analysis,
  3. code generation, and
  4. optimizations
- The process is managed by a static compiler class.
- It parses the command line parameters, handles the input and output of files, and calls the different stages.

```
1 ./LUIECompiler --input "./program.luie" \
2               --output "./build/program.qasm" \
3               --optimization nullgate+peepingcontrol
```

A command line interface example.

## Symbols and Symbol Table

- Through out the whole compilation process, the symbol table is used to save and propagate symbol information.
- Additionally, it handles higher-level concepts such as variable contexts.
- It contains a dictionary that maps identifiers to the symbol objects.
- All symbol objects are derived from an abstract symbol class.



A diagram showing the hierarchy of symbol classes.

## Lexical and Syntactic Analysis

- The first compilation stage is the lexical and syntactic analysis.
- Both the lexer and parser are created with the ANTLR4 tool.
- It generates the source code based on a given grammar.
- The implementation of the grammar is a more elaborate version of the syntax given previously.

```
1 parse      : mainblock EOF;  
2  
3 mainblock  : gateDeclaration* (declaration | statement)*;  
4  
5 block      : (declaration | statement)*;
```

The basic structure of parsing rules for Luie.

## Semantic Analysis

- The next step is the semantic analysis.
- An analysis without any context is not sufficient for non-syntactic constraints of the program.
- For example, the syntactic analysis may ensure gate declarations are always at the beginning of a program, but it cannot ensure that all identifiers in a gate application were previously defined.
- Our semantic analysis is divided into two parts:
  1. the declaration analysis and
  2. the type checking.
- Both parts are implemented as ANTLR listener classes.
- These traverse the parse tree and call both an enter and exit function for each grammar rule, e.g., `EnterBlock` and `ExitBlock`.

## Semantic Analysis

- The declaration analysis ensures that all identifiers used were previously declared and all identifiers used in declarations are not already declared.
- This includes throwing warnings for declared but unused identifiers.
- Additionally, it prevents the use of a qubit in a code block that is guarded by the same qubit because this would lead to irreversible gates.
- The type checking ensures that symbols are used in the correct context.
- For example, while a qubit symbol can be used as the argument for a gate application, it does not represent a classical numerical value and, therefore, cannot be used in the context of a factor.
- Since we do not give the type of composite gate argument, its body cannot be type checked, and any invalid types are thrown at generation time.

# Implementation

---

## Code Generation

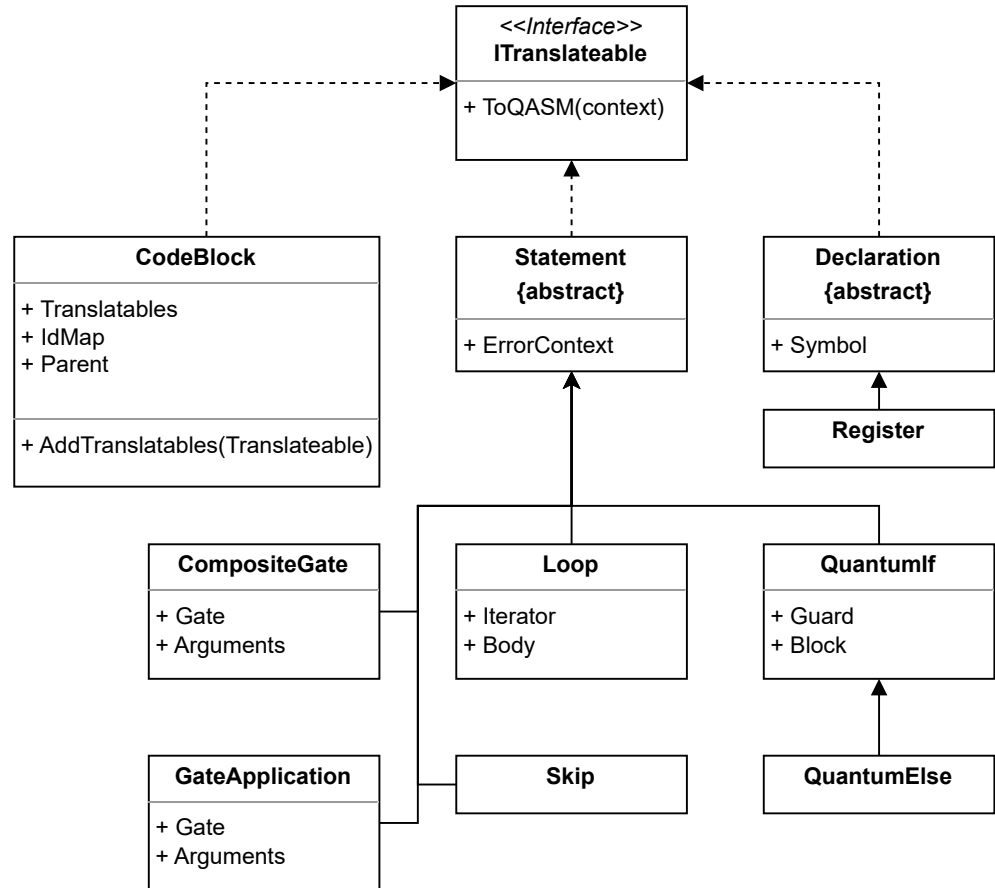
- First, the parse tree is traversed and the source code is translated to an in-memory representation.
- Next, this source code representation (SCR) is translated to the target code representation (TCR).
- Then, the TCR can be translated directly to the textual OpenQASM code.
- We want to go through the process with the example program from before.
- For simplicity, the named constant was replaced with a constant value.

```
1 gate c_h_reg(control, reg) do
2     qif control do
3         for i in range(sizeof(reg)) do
4             h reg[i];
5         end
6     end
7 end
8
9 qubit c;
10 qubit[3] a;
11 c_h_reg c, a;
```

An example Luise program to show the code generation process.

## Source Code Representation

- All SCR objects implement the translatable interface, which requires a translation function.
- There are three main translatables: the code block, statement, and declaration classes.
- The block contains a list of translatables and is used for both the main block and the body of some statements.
- The declaration consists only of register declarations; the compile-time-only declarations are only saved in the symbol table.



A diagram showing the hierarchy of translatable classes.



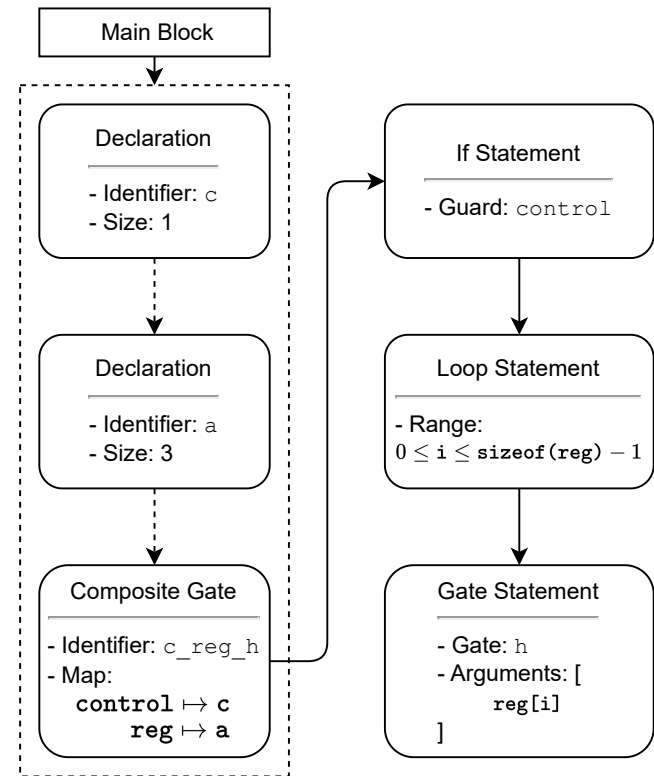
# Implementation

## Source Code Representation — Example

- Any translation always consists of the main block.
- It contains three translatables.
- The first two are the declarations, and the last is the composite gate statement.
- The gate's body contains only an if-statement, which, in turn, contains a loop statement.
- The loop statement consists of a gate application.

```
1 gate c_h_reg(control, reg) do
2   qif control do
3     for i in range(sizeof(reg)) do
4       h reg[i];
5   end end end
6
7 qubit c; qubit[3] a;
8 c_h_reg c, a;
```

An example Luie program to show the code generation process.



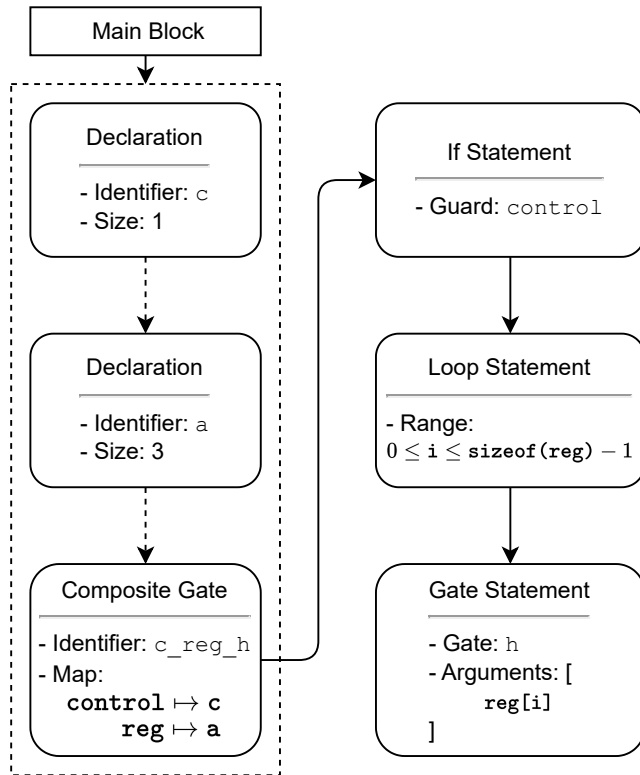
The SCR of the example program.

## Target Code Representation

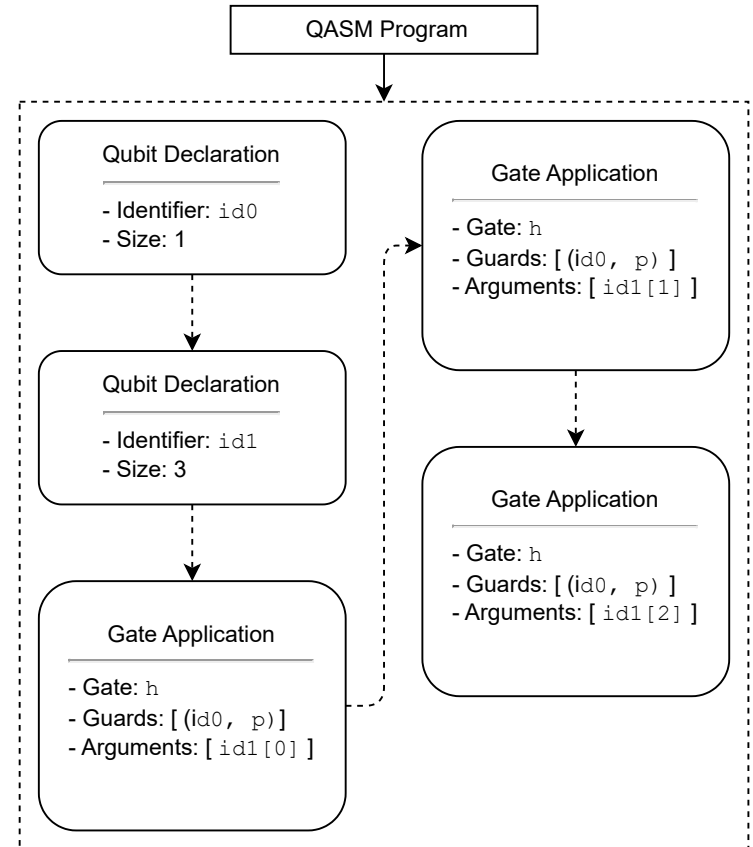
- The basis of the TCR is the `QASMPprogram` object.
- It contains a list of `Code` objects, which are either declarations of gate applications.
- All SCR objects can be translated to a list of code objects and appended to the program object.
- The translations are as described in the translation section.
- For example, an if-statement adds a control qubit to all gate applications in the block translation.
- All code objects implement a `ToCode` function that returns the textual representation of the statement.
- To translate the program object, the code objects are simply iterated, converted to text, and written to a file.

$$ct(\text{qif } qArg \text{ do } blk \text{ end}, st) = control(qt(qArg, st), bt(blk, st))$$

## Target Code Representation — Example



The SCR of the example program.



The TCR of the example program.

## Translated Example Program

- In the end, the TCR is converted to the OpenQASM program.
- For this, the version string and include header are prepended to the code.
- Additionally, for each quantum register, a classical one is declared, and the registers are measured and saved to the corresponding classical registers.
- These additions are performed right before the result is written to the output and after the optimization.

```
1 OPENQASM 3.0;
2 include "stdgates.inc";
3 qubit id0;
4 qubit[3] id1;
5 ctrl(1) @ h id0, id1[0];
6 ctrl(1) @ h id0, id1[1];
7 ctrl(1) @ h id0, id1[2];
8 bit id0_measurement;
9 measure id0 -> id0_measurement;
10 bit[3] id1_measurement;
11 measure id1 -> id1_measurement;
```

The OpenQASM translation of the example Luie program.

## Optimization

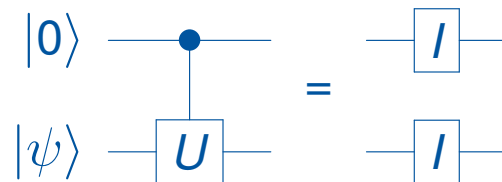
- The compiler can perform optimizations to reduce both the number of gates and qubits.
- The optimizations are performed to the TCR to allow for more optimizations at the cost of performance.
- The compiler performs general peephole optimizations based on rules presented by [Garcia-Escartin and Chamorro-Posada, 2011].
- They can be divided into four rules:
  1. *null gate*,
  2. *peeping control*,
  3. *Hadamard reduction*, and
  4. *control reversal*.

## Null Gate and Peeping Control Rules

- Null gates are combinations of gates under specific conditions that are equivalent to the identity gate.
- The simplest null gate version is the twofold application of a self-inverse gate.
- They can be removed entirely from the circuit.
- Our peeping control rules are a special case of null gates.
- In this case, the value of a control is known and the gate can be removed if it is not applied.
- Since the implementation of the rules differs greatly from the other null gate rules, it is separated as its own rule.



Null gates of self-inverse gates.



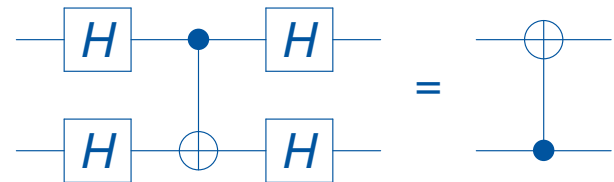
Null gates for gates in specific conditions.

## Hadamard Reduction and Control Reversal Rules

- The Hadamard reduction can reduce either an  $X$  or a  $Z$  gate surrounded by  $H$  gates to the other without the surrounding gates.
- The rule is also the basis for the control reversal.
- It emerges when the Hadamard reduction is combined with the control reversal of the controlled- $Z$  gate.
- A controlled-not gate surrounded by Hadamard gates is equivalent to the target and control qubits exchanged without Hadamard gates.



A Hadamard reduction rule.



Null gates for gates in specific conditions.

## Circuit Graph

- Instead of applying the rules directly to the program, it is translated to a circuit graph.
- Then, the graph is optimized and the result is translated back to an OpenQASM program.
- The circuit graph  $C$  is acyclic and directed.
- It consists of a set of nodes  $V$  — split into input, output, and gate nodes — and edges  $E$ .
- Additionally, it contains a set of qubits  $Q$  and two functions that map nodes  $Q_V$  and edges  $Q_E$  to qubits.

$$\begin{aligned} C &= (V, E, Q, Q_E, Q_V) \\ V &= \underbrace{I}_{\text{Input Nodes}} \cup \underbrace{O}_{\text{Output Nodes}} \cup \underbrace{G}_{\text{Gate Nodes}} \\ E &\subseteq \{(x, y) \mid x, y \in V \wedge x \neq y\} \\ Q_V &: I \cup O \rightarrow Q \\ Q_E &: E \rightarrow Q \end{aligned}$$



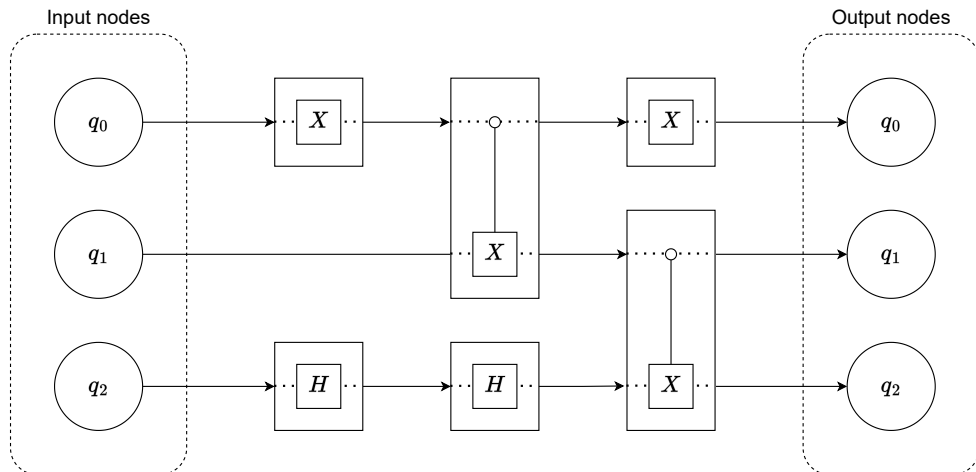
# Implementation

## Example Circuit Graph

- Input and output nodes have only one outgoing or incoming edge, respectively.
- For each qubit,  $C$  contains an input-output node pair.
- Each edge coming into a gate node has a corresponding outgoing one with the same qubit.
- These represent an argument of the gate application.
- The wire for qubit  $q$  is represented by the path from the input to the output node where all edges map to  $q$ .

```
1 qubit[3] q;  
2 x q[0];  
3 cx q[0], q[1];  
4 x q[0];  
5 h q[2];  
6 h q[2];  
7 cx q[1], q[2];
```

A simple, unoptimized program.

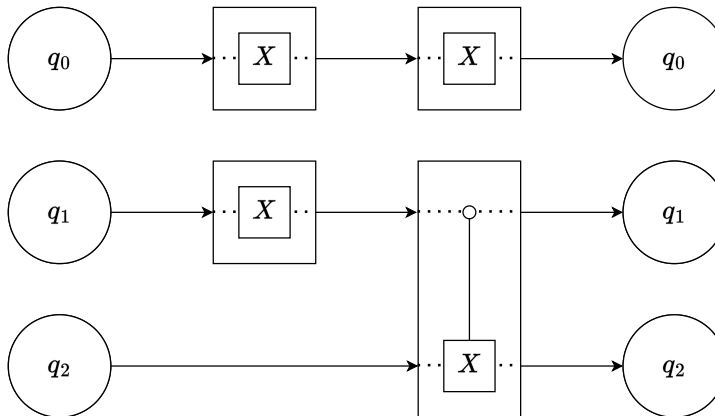


An example of a simple, unoptimized circuit graph.

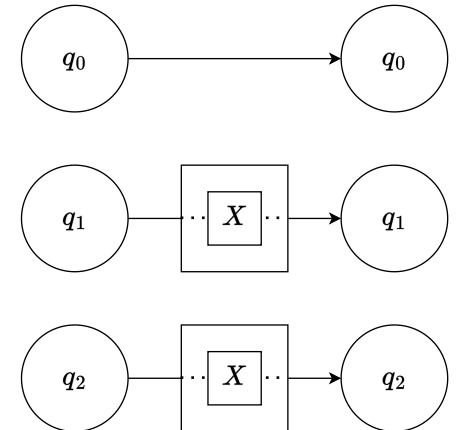


## Example Optimization Process II

- While all qubit wires were iterated, there are still possible optimizations.
- This is because applying optimizations may enable others.
- Therefore, the optimization is repeated as long as the previous iteration applied optimizations.
- In the next iteration, the  $XX$  null gate can be removed.
- Additionally, a peeping control rule can be applied to the remaining  $CX$  gate.



The circuit graph after the first optimization step.



The completely optimized graph.

## Evaluation

- The evaluation consists of two aspects:
  1. the optimizations performed by the compiler and
  2. the execution time of the compilation stages.
- As an example program, we use the quantum ripple-carry adder proposed by [Cuccaro et al., 2004].
- It takes two registers *a* and *b* as well as two qubits *cin* and *cout*.
- The adder adds the *a* register to the *b* register.
- The *cin* and *cout* qubits are used as input and output carry bits.
- In our implementation, it consists of *CX* and *CCX* gates.

## Optimization Evaluation

- For the optimization evaluation, we used both inputs with classical values and values in superposition.
- The first inputs were  $a = |1\rangle$  and  $b = |15\rangle$ .
  - Since the inputs are classical values, peeping control rules and null gate rules can be applied.
  - The circuit can be optimized such that the resulting one only contains gates that initialize the result.
  - Only two X gates remain.
  - While the first gate flips the first qubit of the a register, initializing it to  $|1\rangle$ , the second flips the carry output qubit, indicating a result of  $|16\rangle$ .
- The second inputs were  $a = \frac{1}{\sqrt{2}}(|0\rangle + |3\rangle)$  and  $b = |4\rangle$ .
  - Since the inputs are now values in superposition, peeping control rules and null gate rules can only partially be applied.
  - In this case, only twelve of 25 gates can be optimized.
  - For other inputs in superposition, even fewer gates are optimized.

## Performance Evaluation

- To evaluate the performance, we compiled the adder with an input of  $a = \frac{1}{\sqrt{2}}(|0\rangle + |3\rangle)$  and  $b = 15$  for different register sizes  $n$ .
- Since the program size does not change, the execution times of the semantic analysis remain constant.
- The code generation stage shows a linear increase, as the compiled program increases linearly with the register size.
- The optimization has the worst performance with an approximate quadratic increase.

| Register Size $n$ | Execution Time of Stages in ms |                 |              |
|-------------------|--------------------------------|-----------------|--------------|
|                   | Semantic Analysis              | Code Generation | Optimization |
| 64                | 27.3                           | 47.8            | 711.6        |
| 128               | 26.3                           | 50.4            | 2292.4       |
| 256               | 26.2                           | 59.7            | 10755.7      |
| 512               | 25.8                           | 74.9            | 60204.7      |
| 1024              | 26.1                           | 109.1           | 405376.6     |






The execution times compiling a quantum ripple-carry adder with different register sizes.

## Conclusion

- The evaluation revealed aspects that can be improved.
- The optimizations are focused on high-level optimizations and work best in tandem with other tools.
- A possible improvement is the addition of hardware-focused features and transpilation to concrete devices.
- Additionally, the performance of the optimization stage could be improved by parallelization of the wire traversals.
- However, this would increase the complexity of the program as, e.g., race conditions need to be considered.
- Other general improvements could be the addition of type casting for the named constants.
- Furthermore, the predefined function could be expanded, and constants such as  $\pi$  or  $e$  could be added.
- Lastly, explicit measurements could be added.

## References


---

-  Altenkirch, T. and Grattage, J. (2005).  
A functional quantum programming language.  
*In 20th Annual IEEE Symposium on Logic in Computer Science (LICS' 05)*, pages 249–258. IEEE.
-  Cuccaro, S. A., Draper, T. G., Kutin, S. A., and Moulton, D. P. (2004).  
A new quantum ripple-carry addition circuit.
-  Garcia-Escartin, J. C. and Chamorro-Posada, P. (2011).  
Equivalent quantum circuits.
-  Landauer, R. (1961).  
Irreversibility and heat generation in the computing process.  
*IBM Journal of Research and Development*, 5(3):183–191.
-  Ying, M., Yu, N., and Feng, Y. (2012).  
Defining quantum control flow.



## References

---

-  Yuan, C., Villanyi, A., and Carbin, M. (2024).  
Quantum control machine: The limits of control flow in quantum programming.  
*Proceedings of the ACM on Programming Languages*, 8(OOPSLA1):1–28.