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

- The idea of quantum control flow was first used by [Altenkirch and Grattage, 2005].
- Later, it was defined by [Ying et al., 2012].
- Nevertheless, most quantum languages focus on classical control flow with quantum data.
- Only recently, the Quantum Control Machine was proposed by [Yuan et al., 2024].
- It is an instruction set architecture allowing for quantum control flow with an assembly-like syntax.
- However, the low-level design of the architecture and syntax results in complex programs.
- In turn, we want to build upon the ideas and define a high-level quantum language with quantum control flow.
- Furthermore, we also want to implement a compiler for it.

Quantum Control Flow

- The idea of quantum control flow was first used by [Altenkirch and Grattage, 2005] to define a function quantum programming language.
- For example, it was used to define the Hadamard gate as the function *had*:

$$
\begin{aligned}
\mathit{had} &: Q \to Q \\
\mathit{had} &: x \mapsto \texttt{if}^{\circ} x \\
&\qquad\quad \texttt{then} \ \{\mathit{false} \mid -\mathit{true}\} \\
&\qquad\quad \texttt{else} \ \{\mathit{false} \mid \mathit{true}\}
\end{aligned}
$$

- Later, the concept was formally defined by [Ying et al., 2012].
- Quantum branching allows for the execution of functions based on values in superposition.
- The result is the superposition of the results of individual executions.

Limitations — Reversibity

- Quantum control flow is mainly limited by two principles: *reversibility* and *synchronization*.
- Any sequence of instructions on gate-based quantum computers, excluding measurements, is required to be reversible by definition, as they are all unitary transformations.
- As a result, control flow, as implemented in classical computers, is not possible.
- For example, any classical jump instruction is inherently irreversible.
- Landauer Embedding [Landauer, 1961] seems to offer a solution.
- The embedding can turn any non-reversible function into a reversible one by not only returning the output but also the input of the function.
- For example, any non-reversible function $f : D \to D'$ can be given as a reversible function $g : D \to D' \times D$ with $g(x) = (f(x), x)$.
- However, because the output is the result together with the program history and the result depends on the history, they become entangled.
- This leads to disruptive entanglement [Yuan et al., 2024].

Limitations — Synchronization

- The program counter can become entangled with the data and result in disruptive entanglement leading to an invalid result.
- The principle of synchronization states that control flow must become independent from the data.
- In turn, loops cannot depend solely on value in superposition.
- Instead, a loop must be bounded by a classical value [Yuan et al., 2024].
- An example of this is the *Tortoise and hare* problem.
- When a program is run with a loop based on a superposition of two different values, the loop is executed a different number of times and one will finish first.
- The control flow must become independent of the data, but the program cannot wait.
- To prevent this issue, the program must include padding operations that are executed instead of the main loop.

Quantum Control Machine

- Quantum Control Machine (QCM), proposed by [Yuan et al., 2024], is an instruction set architecture focused on quantum control flow.
- Both its syntax and logic are similar to classical assembly language, utilizing (conditional) jump instructions.
- The architecture employs a branch control register *bcr* to enable reversible jump instructions.
- Instead of increasing the IP by 1 after each statement, it is increased by the value in the *bcr*.
- The *bcr* can then be reversibly modified.
- To jump by 5, the *bcr* is increased by 5 and, at its destination, decreased by 5 again.

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[1]
No-op	`nop`	Only increases instruction pointer by the *bcr*.
Addition	`add` *ra rb*	Adds register *rb* to *ra*.
Multiplication	`mul` *ra rb*	Multiplies register *ra* by *rb*.
Jump	`jmp` *p*	Increases *bcr* by *p*.
Conditional Jumps	`jz` *p ra*	Increases *bcr* by *p* if *ra* is 0.
	`jne` *p ra rb*	Increases *bcr* by *p* if *ra* is not equal to *rb*.

[1] 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.

(Non-) Reversible Example

- An example of a classical program and the reversible equivalent are depicted below.
- Both programs calculate x^y.
- While the second one is reversible, it is not synchronized and depicts the tortoise and hare problem.

```
     add   res $1
     add   r1  y
l1:  jz    l2  r1
     mul   res x
     radd  r1  $1
     jmp   l1
l2:  nop
```

A non-reversible exponentiation algorithm.

```
     add   res $1
     add   r1  y
l1:  rjne  rl1 r1  y
rl2: jz    l2  r1
     mul   res x
     radd  r1  $1
rl1: jmp   l1
l2:  rjmp  l2
```

A reversible exponentiation algorithm.

Reversible Synchronized Example

- Shown below is a synchronized implementation that calculates $x^{\min y, max}$.
- In this case, *max* is a classical bound.

```
          add   res  $1
          add   r1   max
l1:       rjne  rl1  r1   max
rl2:      jz    l2   r1
rl3:      jg    l3   r1   y
          mul   res  x
rl4:      jmp   l4
l3:       rjmp  rl3
          nop                   ; padding
l4:       rjle  rl4  r1   y
          radd  r1   $1
rl1:      jmp   l1
l2:       rjmp  rl2
```

A synchronized, reversible exponentiation algorithm.

Language Overview

- The idea for our language is to provide a high-level language with the capabilities of the QCM.
- We want to remove low-level concepts and add high-level ones.
- Additionally, since jump instructions in superposition are removed, we need to add other control flow statements so that the language is as expressive as the QCM.
- For this, we introduce multiple high-level concepts and two basic control flow statements:
  - blocks and scopes,
  - different data types,
  - composite gates,
  - loop statements, unrolled at compile time, and
  - quantum if- and else-statements.

Syntax

- We define a CFG CFG_{Luie} for our language.
- The start symbol is the program, consisting of arbitrarily many gate declarations and a block.
- A block is a list of translatables, either statements or declarations.

$$\begin{aligned}
CFG_{Luie} &= (V_{Luie}, \Sigma_{Luie}, R_{Luie}, prg_{Luie}) \\
V_{Luie} &= \{exp, rExp, gate, qArg, stm, prg_{Luie}, \ldots\} \\
\Sigma_{Luie} &= \{\texttt{..}, \texttt{range}, \texttt{(}, \texttt{)}, \ldots\} \quad \texttt{where} \quad n \in \mathbb{N}_0, id \in Identifier
\end{aligned}$$

$$\begin{aligned}
Program : prg_{Luie} &::= gDcl_1 \ldots gDcl_n\ blk \mid blk \\
Block : blk &::= t_1 \ldots t_n \mid \epsilon \\
Translatable : t &::= stm \mid dcl \\
Declaration : dcl &::= \texttt{const}\ \ id\ \ \texttt{=}\ \ exp\texttt{;} \mid \\
&\qquad \texttt{qubit}\ \ id\texttt{;} \\
GateDeclaration : gDcl &::= \texttt{gate}\ \ id\ \ \texttt{(}id_1, \ldots, id_n\texttt{)}\ \ \texttt{do}\ \ blk\ \ \texttt{end}
\end{aligned}$$

Syntax

- There are three different statements: a quantum if-statement, a loop statement, and the application of a predefined or composite gate.
- The qubit argument differentiates between qubit or register access.
- For the register access or constant declarations, expressions can be used.
- Additionally, we defined a set of defined gates to differentiate the corresponding translations.

$$
\begin{aligned}
\mathit{Statement} : \mathit{stm} ::= {} & \texttt{qif} \ \ \mathit{qArg} \ \ \texttt{do} \ \ \mathit{blk} \ \ \texttt{end} \mid \\
& \texttt{for} \ \ \mathit{id} \ \ \texttt{in} \ \ \mathit{rExp} \ \ \texttt{do} \ \ \mathit{blk} \ \ \texttt{end} \mid \\
& \mathit{id} \ \mathit{qArg}_1, \ldots, \mathit{qArg}_n\texttt{;} \\
\mathit{QubitArgument} : \mathit{qArg} ::= {} & \mathit{id} \mid \mathit{id}[\mathit{exp}] \\
\mathit{Expression} : \mathit{exp} ::= {} & n \mid \mathit{id} \mid \mathit{exp}_1 \ + \ \mathit{exp}_2 \mid \mathit{exp}_1 \ - \ \mathit{exp}_2 \mid \ldots \\
\mathit{RangeExpression} : \mathit{rExp} ::= {} & n_1..n_2 \mid \texttt{range(}\mathit{exp}\texttt{)} \mid \texttt{range(}\mathit{exp}_1, \mathit{exp}_2\texttt{)}
\end{aligned}
$$

$$
\mathit{ConstGates} = \{\texttt{h}, \texttt{x}, \texttt{y}, \texttt{z}, \texttt{cx}, \texttt{ccx}\}
$$

Example Program

- In the example program, a composite gate is defined that applies the *H* gate to a register with a control qubit.
- This composite gate is then applied to a register.

```
gate c_h_reg(control, reg) do
    qif control do
        for i in range(sizeof(reg)) do
            h reg[i];
        end
    end
end

const regSize : int = 3;
qubit c;
qubit[regSize] a;
c_h_reg c, a;
```

An example Luie program.

Symbol Table

- The symbol table saves the symbol information relevant for the translation.
- It contains four different types of symbols:
  1. named constants,
  2. quantum registers and qubits,
  3. qubit arguments, and
  4. composite gates.

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

Translation Function and Block Translation

- The *trans* function translates the Luie program to OpenQASM.
- The initial symbol table st_ϵ contains no mappings.
- Next, the block translation function *bt* translates all translatables, i.e., statements and declarations.

$$trans : Program \dashrightarrow QASMProgam$$
$$trans(gDcl_1 \dots gDcl_n\ blk) = bt(blk, update(update(update(st_\epsilon, gDcl_1), \dots), gDcl_n))$$

$$bt : Block \times SymbolTable \dashrightarrow QASM$$
$$\begin{aligned} bt(t_1 \dots t_n, st_1) = {} & tr_1 \quad \text{where } (tr_1, st_2) = tt(t_1, st_1) \\ & \dots \\ & tr_n \quad \text{where } (tr_n, \_) = tt(t_n, st_n) \end{aligned}$$

Translatable and Declaration Translation

- The translatable function *tt* translates each translatable.
- Since declarations update the symbol table, the function returns the updated symbol table.
- The language allows for different variable scopes and, in turn, an identifier can be used multiple times.
- Therefore, a unique identifier *uid* is generated for the translation of identifiers.

$$tt : \mathit{Translatable} \times \mathit{SymbolTable} \dashrightarrow \mathit{QASM} \times \mathit{SymbolTable}$$

$$tt(t, st) = \begin{cases} dt(t, st) & \text{if } t \in \mathit{Declarations} \\ (ct(t, st), st) & \text{otherwise} \end{cases}$$

$$dt : \mathit{Declaration} \times \mathit{SymbolTable} \dashrightarrow \mathit{QASM} \times \mathit{SymbolTable}$$

$$dt(\underbrace{\texttt{qubit}\ \ id;}_{decl}, st) = (\texttt{qubit}\ \ uid\,\boldsymbol{;}\,, st')$$

$$\text{where } st' = update(decl, st) \text{ and } st'[id] = (\texttt{qubit}, 1, uid)$$

Command Translation

- The commands are translated with the *ct* function.
- We take a look at an example translation of a quantum if-statement.
- The qubit argument translation *qt* is used to differentiate between qubits and register accesses and looks up the *uid*.
- The *control* function adds the translated *qArg* as a guard to all gate applications in the block translation.

$$ct : \mathit{Statement} \times \mathit{SymbolTable} \dashrightarrow \mathit{QASM}$$

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

$$\begin{aligned} control(qArg, \quad & id\ qArg_1, \ldots, qArg_{n'}\,\texttt{;}\,) = \\ \texttt{ctrl(1) @} \quad & id\ qArg, qArg_1, \ldots, qArg_{n'}\,\texttt{;} \\ control(qArg, \texttt{ctrl(}n\texttt{) @} \quad & id\ qArg_1, \ldots, qArg_{n'}\,\texttt{;}\,) = \\ \texttt{ctrl(}n+1\texttt{) @} \quad & id\ qArg, qArg_1, \ldots, qArg_{n'}\,\texttt{;} \end{aligned}$$

Implementation

Overview

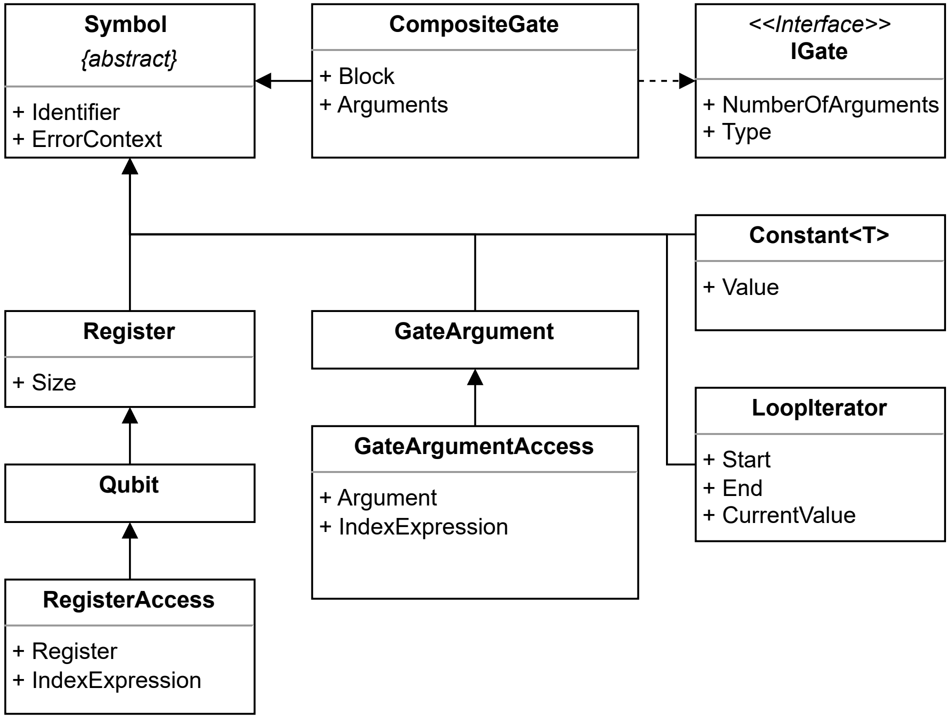
- The implementation of the compiler is differentiate into four different stages:
  1. the lexical and syntactic analysis,
  2. semantic analysis,
  3. code generation, and
  4. optimizations
- The process is managed by a static compiler class.
- It parses the command line parameters, handles the input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

```
./LUIECompiler --input "./program.luie"                 \
               --output "./build/program.qasm"          \
               --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 identifier 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.

```
parse       : mainblock EOF;

mainblock : gateDeclaration* (declaration | statement)*;

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 ensure 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.

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.

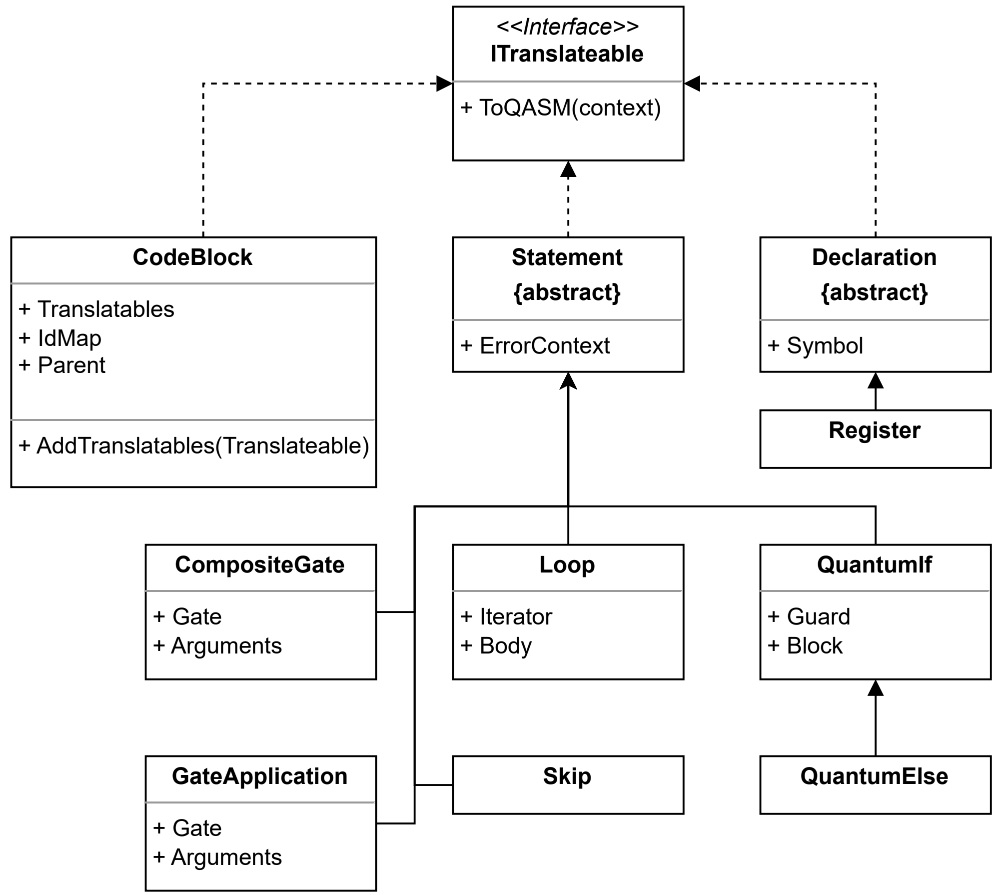
```
gate c_h_reg(control, reg) do
    qif control do
        for i in range(sizeof(reg)) do
            h reg[i];
        end
    end
end

qubit c;
qubit[3] a;
c_h_reg c, a;
```

An example Luie program to show the code generation process.

Implementation

Source Code Representation

- All SCR objects implement the translatable interface, which requires a translation function.
- The 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.

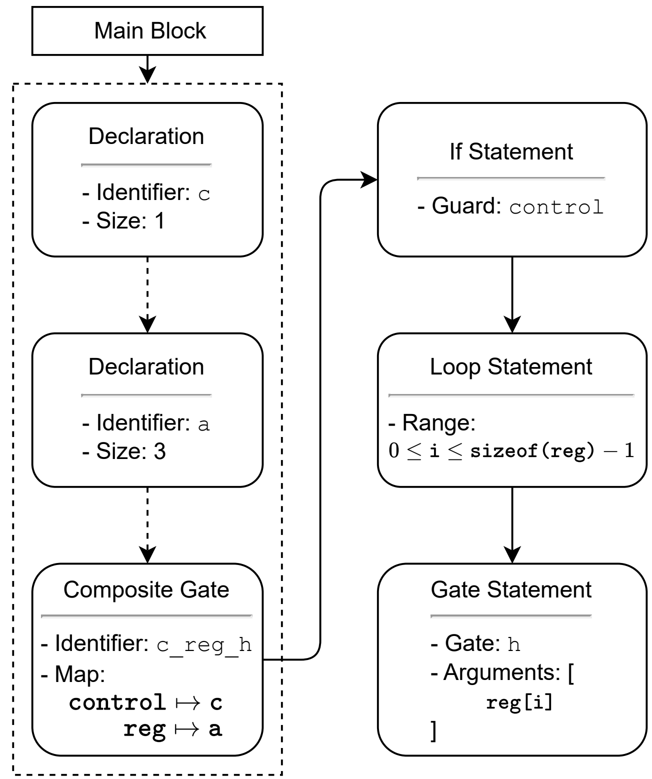
Implementation

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.

```
gate c_h_reg(control, reg) do
    qif control do
        for i in range(sizeof(reg)) do
            h reg[i];
end end end

qubit c; qubit[3] a;
c_h_reg c, a;
```

An example Luie program to show the code generation process.



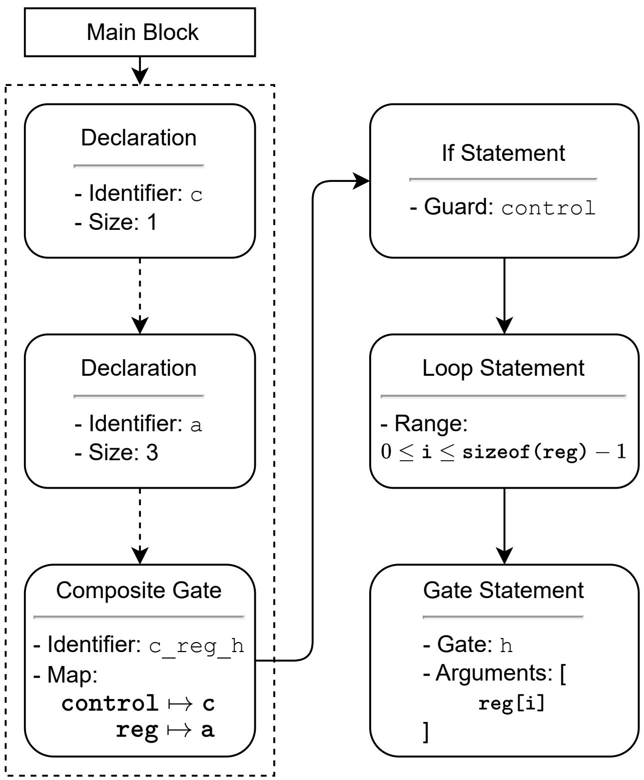
The SCR of the example program.

Target Code Representation

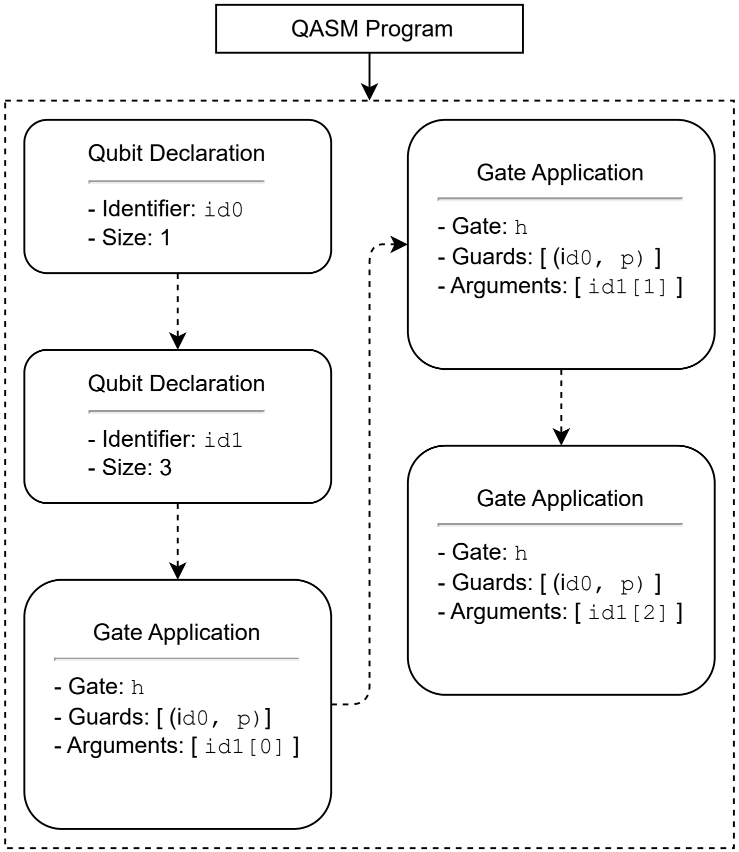
- The basis of the TCR is the `QASMProgram` object.
- It contains a list of `Code` objects, which are either declarations of gate applications.
- All SCR objects can be translated to a list of code objects and appended to the program object.
- The translations are as described in translation section.
- For example, a 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.

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

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 is appended 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.

```
OPENQASM 3.0;
include "stdgates.inc";
qubit id0;
qubit[3] id1;
ctrl(1) @ h id0, id1[0];
ctrl(1) @ h id0, id1[1];
ctrl(1) @ h id0, id1[2];
bit id0_measurement;
measure id0 -> id0_measurement;
bit[3] id1_measurement;
measure id1 -> id1_measurement;
```

The OpenQASM translation of the example Luie program.

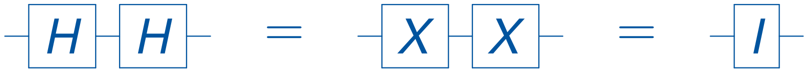
Optimization

- 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*.

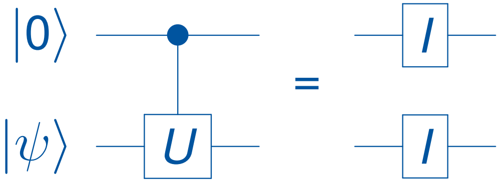
Implementation

Null Gate and Peeping Control Rules

- Null gates are combinations or 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 know 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, its separated as its own rule.



Null gates of self-inverse gates.

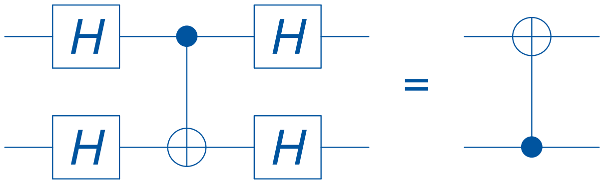


Null gates for gates in specific conditions.

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 then 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.

Cricuit 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 a 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.

$$C = (V, E, Q, Q_E, Q_V)$$

$$V = \underbrace{I}_{\text{Input Nodes}} \cup \underbrace{O}_{\text{Output Nodes}} \cup \underbrace{G}_{\text{Gate Nodes}}$$

$$E \subseteq \{(x, y) \mid x, y \in V \land x \neq y\}$$

$$Q_V : I \cup O \to Q$$

$$Q_E : E \to Q$$

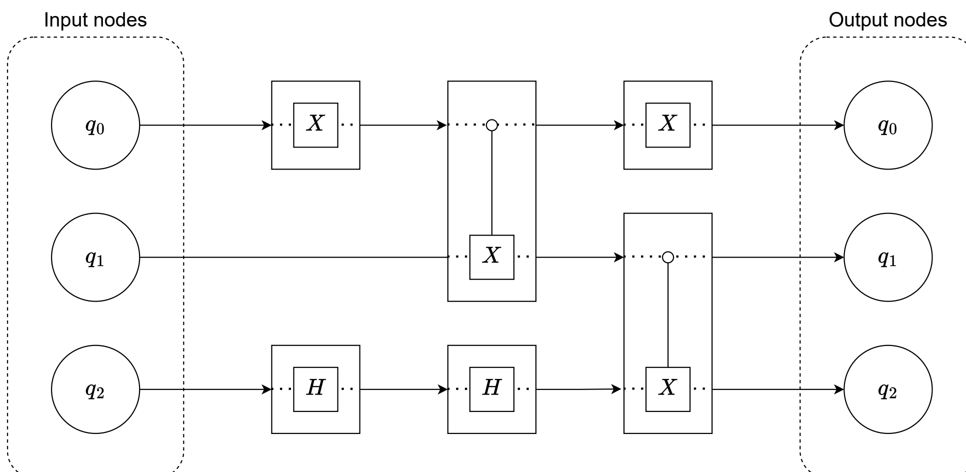
Implementation

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.

```
qubit[3] q;
x q[0];
cx q[0], q[1];
x q[0];
h q[2];
h q[2];
cx q[1], q[2];
```

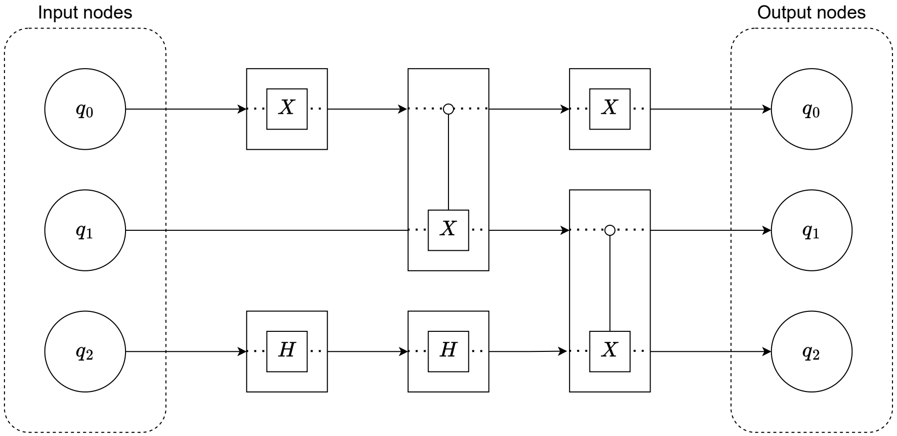
A simple, unoptimized program.



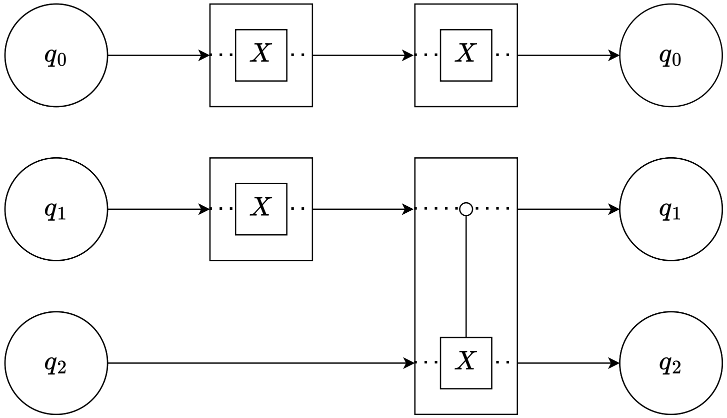
An example of a simple, unoptimized circuit graph.

Example Optimization Process I

- To optimize the graph, each qubit wire is iterated.
- All subpaths up to a maximum length are checked for optimized alternatives.
- In the example, the peeping control rule can be applied to the first *CX* gate.
- Next, the *HH* null gate can be removed.



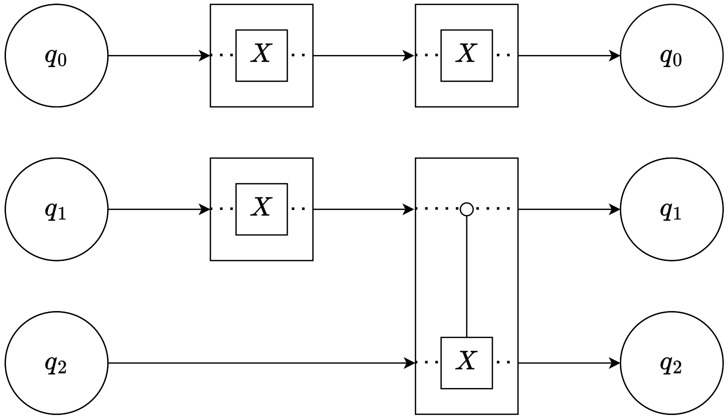
An example of a simple, unoptimized circuit graph.



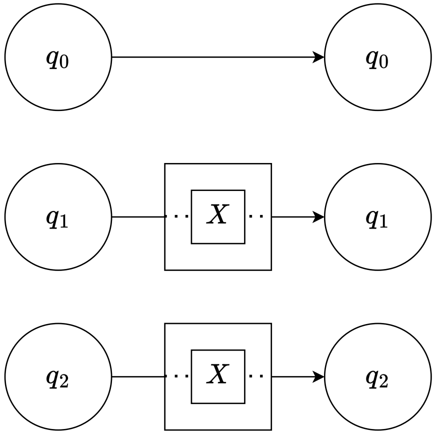
The circuit graph after the first optimization step.

Example Optimization Process II

- While all qubit wire 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 consist 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 be 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 π 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.