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

- 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 $\longrightarrow$ complex programs.

$\Rightarrow$ Build upon the ideas, define a high-level quantum language with quantum control flow.

$\Rightarrow$ Plan to implement a compiler for it.

Background

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*:

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

- 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.

Background

Limitations — Reversibility

- Mainly limited by two principles: reversibility and synchronization.
- Instructions required to be reversible, as they are unitary transformations.

$\Rightarrow$ Control flow, as implemented in classical computers, not possible.

- Landauer Embedding [Landauer, 1961] seems to offer solution.
- Can make non-reversible function $f : D \to D'$ reversible by outputting input value.
- Result is $g : D \to D' \times D$ 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.

$\Rightarrow$ Disruptive entanglement.

- The principle of synchronization: Control flow must become independent from the data.
- Program may include padding operations to ensure synchronization.

$\Rightarrow$ Loops cannot depend solely on value in superposition.

$\Rightarrow$ Instead, a loop must be bounded by classical value [Yuan et al., 2024].

Background

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[1]
No-op	`nop`	Only increases instruction pointer by the *bcr*.
Addition Multiplication	`add` *ra rb* `mul` *ra rb*	Adds register *rb* to *ra*. Multiplies register *ra* by *rb*.
Jump	`jmp` *p*	Increases *bcr* by *p*.
Conditional Jumps	`jz` *p ra* `jne` *p ra rb*	Increases *bcr* by *p* if *ra* is 0. 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.

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.

```
      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.

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  rl1  r1   max
 4 rl2:     jz    l2   r1
 5 rl3:     jg    l3   r1   y
 6          mul   res  x
 7 rl4:     jmp   l4
 8 l3:      rjmp  rl3
 9          nop                  ; padding
10 l4:      rjle  rl4  r1   y
11          radd  r1   $1
12 rl1:     jmp   l1
13 l2:      rjmp  rl2
```

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 $\rightarrow$ 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.

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

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

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

/* Block Start */
const regSize : int = 3; // Decl.
qubit c;                  // Decl.
qubit[regSize] a;         // Decl.
c_h_reg c, a;
/* Block End */
```

Luie program with structure highlighted.

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.

$$\begin{aligned} \textit{Statement} : \textit{stm} ::= {} & \texttt{qif}\ \ \textit{qArg}\ \ \texttt{do}\ \ \textit{blk}\ \ \texttt{end} \mid \\ & \texttt{for}\ \ \textit{id}\ \ \texttt{in}\ \ \textit{rExp}\ \ \texttt{do}\ \ \textit{blk}\ \ \texttt{end} \mid \\ & \textit{id}\ \textit{qArg}_1, \ldots, \textit{qArg}_n\texttt{;} \end{aligned}$$

$$\textit{QubitArg} : \textit{qArg} ::= \textit{id} \mid \textit{id}[\textit{exp}]$$

$$\textit{ConstGates} = \{\mathrm{h}, \mathrm{x}, \mathrm{y}, \mathrm{z}, \mathrm{cx}, \mathrm{ccx}\}$$

```
gate c_h_reg(control, reg) do
    /* If-Statement */
    qif control do
        /* Loop Stat. | ⬐ Expression */
        for i in range(sizeof(reg)) do
                 /* ⬑ Range Exp. */
            h reg[i];
        end
    end
end

const regSize : int = 3;
qubit c;
qubit[regSize] a;
c_h_reg c, a;  // Gate Application
        /* ⬑  ⬑ Qubit Argument */
```

Luie program with statements and arguments highlighted.

Language

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.

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

Language

Translation Function and Block Translation

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

$$st_1 = up(st_\epsilon, \texttt{gate c\_h\_reg...}) = [\texttt{c\_h\_reg} \mapsto (\texttt{gate}, \texttt{qif control do ... end}, \texttt{control}, \texttt{reg})]$$

$$\begin{aligned} bt(\texttt{const regSize = 3;}\ \ldots, st_1) = \ & tr_1 && \text{where } (tr_1, st_2) = tt(\texttt{const regSize = 3;}, st_1) \\ & tr_2 && \text{where } (tr_2, st_3) = tt(\texttt{qubit c;}, st_2) \\ & tr_3 && \text{where } (tr_3, st_4) = tt(\texttt{qubit[regSize] a;}, st_3) \\ & tr_4 && \text{where } (tr_4, \_) = tt(\texttt{c\_h\_reg c, a;}, st_4) \end{aligned}$$

Language

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.

$$\begin{aligned} tt(\texttt{c\_h\_reg c, a;}, st) &= (ct(\texttt{c\_h\_reg c, a;}, st), st) \\ tt(\texttt{qubit c;}, st) &= dt(\texttt{qubit c;}, st) \end{aligned}$$

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

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

Language

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.

$$\begin{aligned} st = [&\texttt{c} \mapsto (\texttt{qubit}, 1, uid_c), \\ &\texttt{a} \mapsto (\texttt{qubit}, 3, uid_a)] \end{aligned}$$

$$\begin{aligned} ct(\texttt{if c do x a[0]; end}, st) &= control(qt(c, st), bt(\texttt{x a[0];}, st)) \\ &= control(uid_c, \texttt{x}\ \ uid_a[0]\texttt{;}) \\ qt(c, st) &= st[c]_3 = uid_c \\ bt(\texttt{x a[0];}, st) &= \texttt{x}\ \ qt(a[0])\texttt{;} = \texttt{x}\ \ uid_a[0]\texttt{;} \end{aligned}$$

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

Implementation

Overview

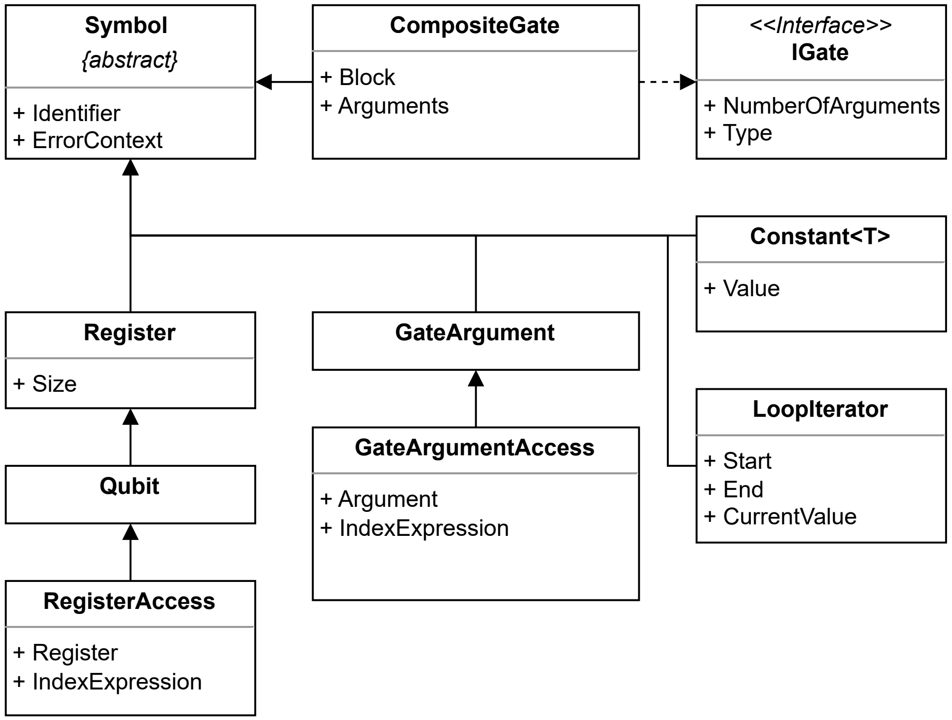
- Four compilation stages:
  1. the lexical and syntactic analysis,
  2. semantic analysis,
  3. code generation, and
  4. optimizations
- Process managed by a static compiler class.
  - Parses command line parameters.
  - Handles input and output of files.
  - Calls the different stages.
  - Handles logging and error messages.

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

A command line interface example.

Symbols and Symbol Table

- Symbol table used to save and propagate symbol information.
- Handles higher-level concepts such as variable contexts.
- Contains dictionary that maps identifiers to symbol objects.
- All symbol objects derived from abstract symbol class.



A diagram showing the hierarchy of symbol classes.

Implementation

Lexical and Syntactic Analysis

- First compilation stage is lexical and syntactic analysis.
- Both lexer and parser created with the ANTLR4 tool.
- Generates the source code based on a given grammar.
- Implementation of the grammar is more elaborate version of previously disused one.

```
parse     : mainblock EOF;

mainblock : gateDeclaration* (declaration | statement)*;

block     : (declaration | statement)*;
```

The basic structure of parsing rules for Luie.

Implementation

Semantic Analysis

- Analyses non-syntactic constraints of program, mainly:
  1. declaration analysis and
  2. type checking.
- Declaration analysis ensures all identifiers used were previously declared and all identifiers used in declarations are not already declared.
- Type checking ensures that symbols are used in the correct context.

```
qubit c;                /* ⬐ Type error (cannot use qubits in arith. expression) */
const regSize : int = c + 2;
           /* ⬐ Already declared */
qubit[regSize] c;
             /* ⬐ Type error (expects a register or qubit) */
c_h_reg c, a, regSize;
       /* ⬑ Undeclared identifier */
```

Luie program with semantic errors highlighted.

Implementation

Code Generation

- Parse tree is traversed and source code is translated to in-memory representation.
- Source code representation (SCR) is translated to target code representation (TCR).
- TCR can be translated directly to the textual OpenQASM code.
- Example code generation with program:

```
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

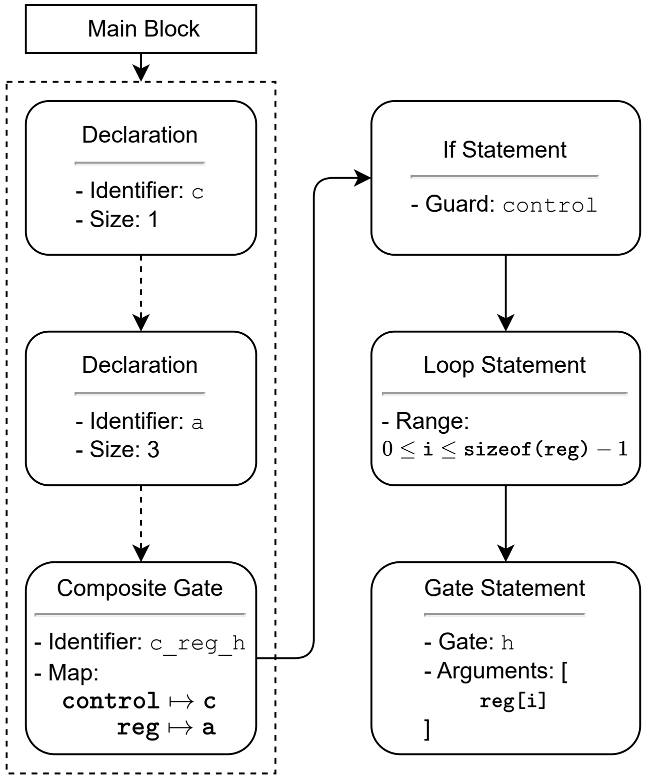
Source Code Representation

- Three main classes:
  1. Code block: contains list of translatables.
  2. Declaration: only register declaration, constants compile-time only.
  3. Statements: variety of different classes.
- Example contains three translatables.
  - First two declarations, last is gate statement.
  - Gate's body $\supset$ if-statement $\supset$ loop statement $\supset$ 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;
```

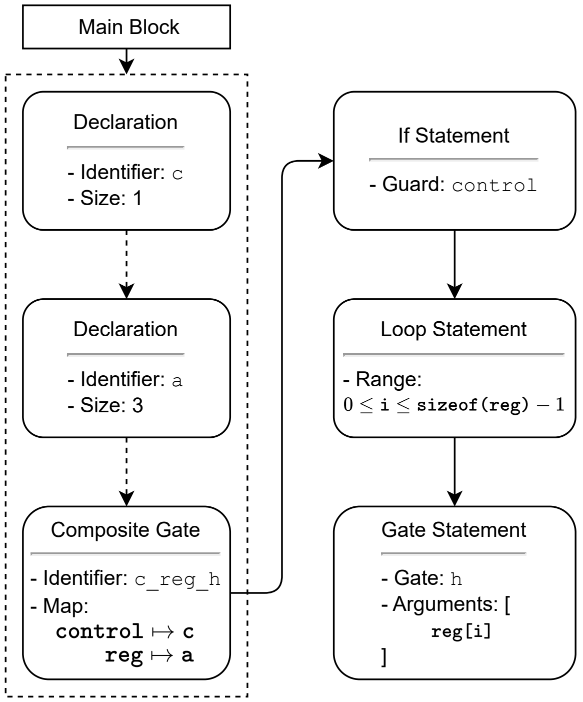
An example Luie program to show the code generation process.



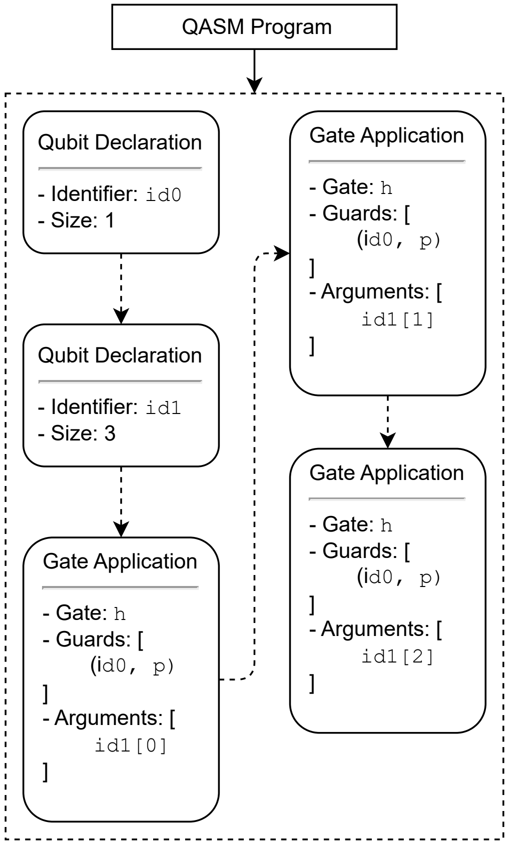
The SCR of the example program.

Target Code Representation

- TCR based on `QASMProgram` object.
- Contains list of `Code` objects, either gate or declaration.
- All SCR objects translated to list of code objects and appended to program object.
- Translate program object: code iterated, converted to text, written file.



The SCR of the example program.



The TCR of the example program.

Implementation

Translated Example Program

- TCR converted to OpenQASM program.
- Version string and include header prepended to the code.
- For each quantum register, classical one is declared, the registers are measured and saved to registers.
- Additions are performed right before result is written to output and after 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.

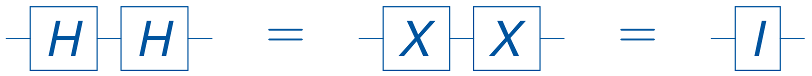
Implementation

Optimization

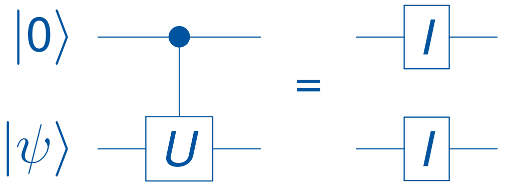
- Compiler can perform optimizations to reduce number of gates and qubits.
- Optimizations are performed to TCR $\rightarrow$ allow for more optimizations at cost of performance.
- Compiler performs general peephole optimizations based on rules presented by [Garcia-Escartin and Chamorro-Posada, 2011].
- 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 equivalent to I.
  - Simplest null gate version is twofold application of self-inverse gate.
  - Can be removed entirely from circuit.
- Our peeping control rules are special case of null gates.
  - Control is $|1\rangle \rightarrow$ remove control.
  - Control is $|0\rangle \rightarrow$ remove gate.



Null gates of self-inverse gates.

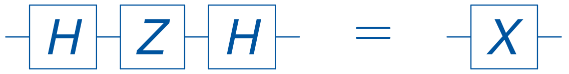


Null gates for gates in specific conditions.

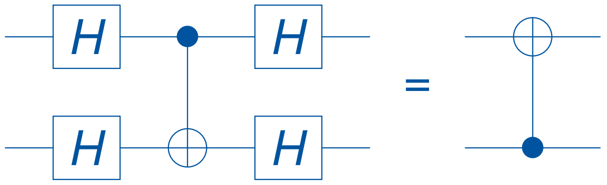
Implementation

Hadamard Reduction and Control Reversal Rules

- Hadamard reduction can reduce either X or Z surrounded by H to the other without surrounding H.
- Rule is basis for control reversal.
- Emerges when Hadamard reduction is combined with control reversal of CZ.
- A CX surrounded by H is equivalent to target and control qubits exchanged without H.



A Hadamard reduction rule.



Null gates for gates in specific conditions.

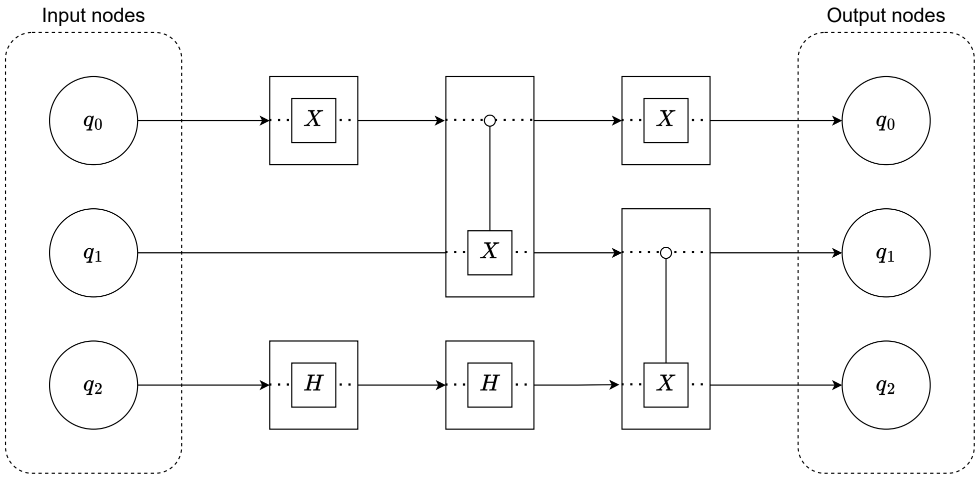
Implementation

Cricuit Graph

- Rules not directly applied to program, but to circuit graph.
- Program $\xrightarrow{\text{Translation}}$ Graph $\xrightarrow{\text{Optimization}}$ Graph $\xrightarrow{\text{Translation}}$ Program
- Circuit graph is acyclic and directed.
  - Nodes split into input, output, gate nodes.
  - Input, output nodes have only one outgoing, incoming edge.
  - Each input, output, edge corresponds belongs to qubit in program.
  - Qubit wire represented by path from input to output node over corresponding edges.

```
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];
```

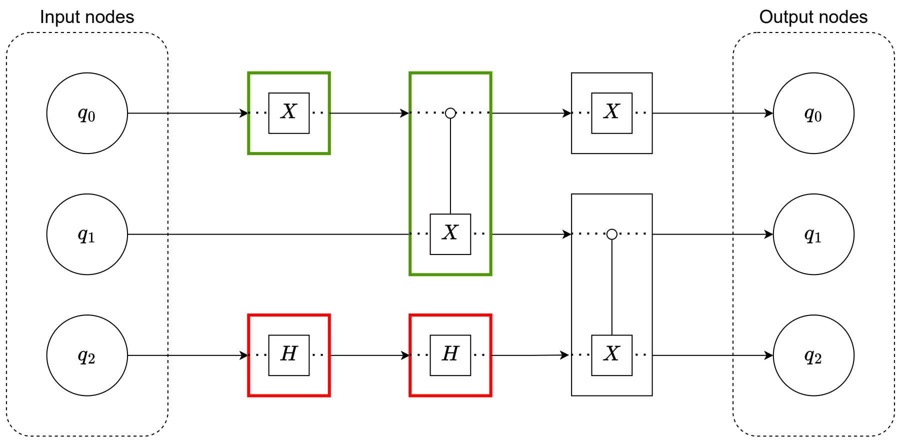
A simple, unoptimized program.



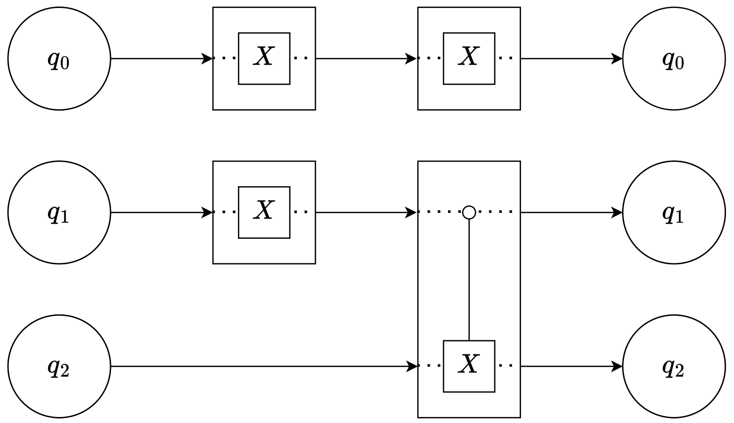
An example of a simple, unoptimized circuit graph.

Example Optimization Process I

- To optimize the graph, each qubit wire is iterated.
- All subpaths (up to max. length) checked for alternatives.
- Example:
  - First wire: Peeping control rule can be applied to first *CX* gate.
  - Third wire: *HH* null gate can be removed.



An example of a simple, unoptimized circuit graph.



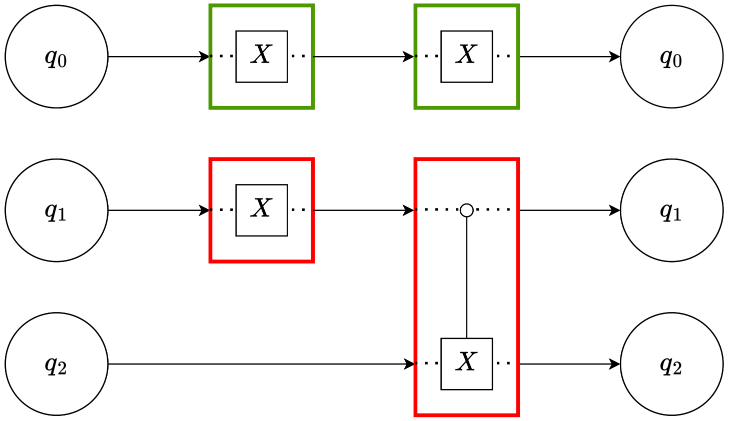
The circuit graph after the first optimization step.

Example Optimization Process II

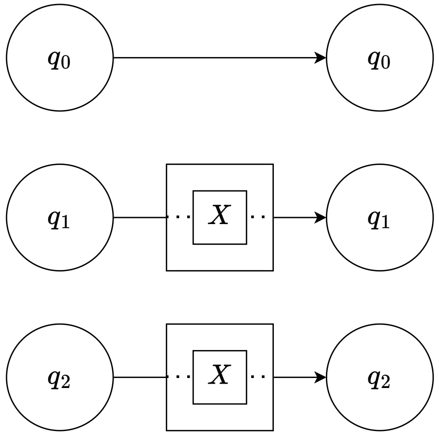
- While all qubit wires iterated, still possible optimizations.
- Applying optimizations may enable others!

$\Rightarrow$ Optimization repeated as long as previous iteration applied optimizations.

- Example:
  - First wire: *XX* null gate can be removed.
  - Second wire: Peeping control rule can be applied to *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.

Evaluation

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.

Evaluation

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

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.