



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

- Semantic Analysis

- Code Generation

- Optimization

Evaluation

Conclusion

Introduction

test [Aaby, 2003]

Quantum Control Flow

- The idea of Quantum Control Flow was first used by [Altenkirch and Grattage, 2005] to define function quantum programming language.
- For example, it was used to define the Hadamard gate as the 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, the concept was formally defined by [Ying et al., 2012].
- Quantum branching allows for the execution of function based on values in superposition.
- The result is the superposition of the results of individual executions.

Limitations — Reversibility

- 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 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 $f : D \rightarrow D'$ can be given as a reversible function $g : D \rightarrow D' \times D$ 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.
- For example, loops cannot depend solely on value in superposition.
- *Tortoise and hare* problem
- Instead, a loop must be bounded by a classical value [Yuan et al., 2024].

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.

Background

Instructions

Operation	Syntax	Semantics ¹
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> .

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

```
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

```
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

- 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
 - 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.

$$CFG_{Luie} = (V_{Luie}, \Sigma_{Luie}, R_{Luie}, prg_{Luie})$$

$$V_{Luie} = \{exp, rExp, gate, qArg, stm, prg_{Luie}, \dots\}$$

$$\Sigma_{Luie} = \{., range, (,), \dots\} \quad \text{where } n \in \mathbb{N}_0, id \in Identifier$$

$$Program : prg_{Luie} ::= gDcl_1 \dots gDcl_n blk \mid blk$$

$$Block : blk ::= t_1 \dots t_n \mid \epsilon$$

$$Translatable : t ::= stm \mid dcl$$

$$Declaration : dcl ::= \text{const } id = exp; \mid \\ \text{qubit } id;$$

$$GateDeclaration : gDcl ::= \text{gate } id (id_1, \dots, id_n) \text{ do } blk \text{ end}$$

Syntax

- There are three different statements: quantum if-statement, loop statement, and application of 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.

$$\begin{aligned} \textit{Statement} : \textit{stm} ::= & \text{qif } \textit{qArg} \text{ do } \textit{blk} \text{ end} \mid \\ & \text{for } \textit{id} \text{ in } \textit{rExp} \text{ do } \textit{blk} \text{ end} \mid \\ & \textit{id } \textit{qArg}_1, \dots, \textit{qArg}_n; \end{aligned}$$
$$\textit{QubitArgument} : \textit{qArg} ::= \textit{id} \mid \textit{id}[\textit{exp}]$$
$$\textit{Expression} : \textit{exp} ::= n \mid \textit{id} \mid \textit{exp}_1 + \textit{exp}_2 \mid \textit{exp}_1 - \textit{exp}_2 \mid \dots$$
$$\textit{RangeExpression} : \textit{rExp} ::= n_1..n_2 \mid \text{range}(\textit{exp}) \mid \text{range}(\textit{exp}_1, \textit{exp}_2)$$
$$\textit{ConstGates} = \{h, x, y, z, cx, ccx\}$$

Example Program

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

An example Luie program.

Symbol Table

- The symbol tables 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
 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

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

$$trans : Program \dashrightarrow QASMProgam$$

$$trans(gDcl_1 \dots gDcl_n blk) = bt(blk, update(update(update(st_\epsilon, gDcl_1), \dots), gDcl_n))$$

$$bt : Block \times SymbolTable \dashrightarrow QASM$$

$$bt(t_1 \dots t_n, st_1) = tr_1 \quad \text{where } (tr_1, st_2) = tt(t_1, st_1)$$

...

$$tr_n \quad \text{where } (tr_n, -) = tt(t_n, st_n)$$

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.

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

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

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

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

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

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.

$ct : Statement \times SymbolTable \dashrightarrow QASM$

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

$control(qArg,$	$id \ qArg_1, \dots, qArg_{n'} ;) =$
$\quad ctrl(1) \ @$	$id \ qArg, qArg_1, \dots, qArg_{n'} ;$
$control(qArg, ctrl(n) \ @$	$id \ qArg_1, \dots, qArg_{n'} ;) =$
$\quad ctrl(n+1) \ @$	$id \ qArg, qArg_1, \dots, qArg_{n'} ;$

Implementation

Overview

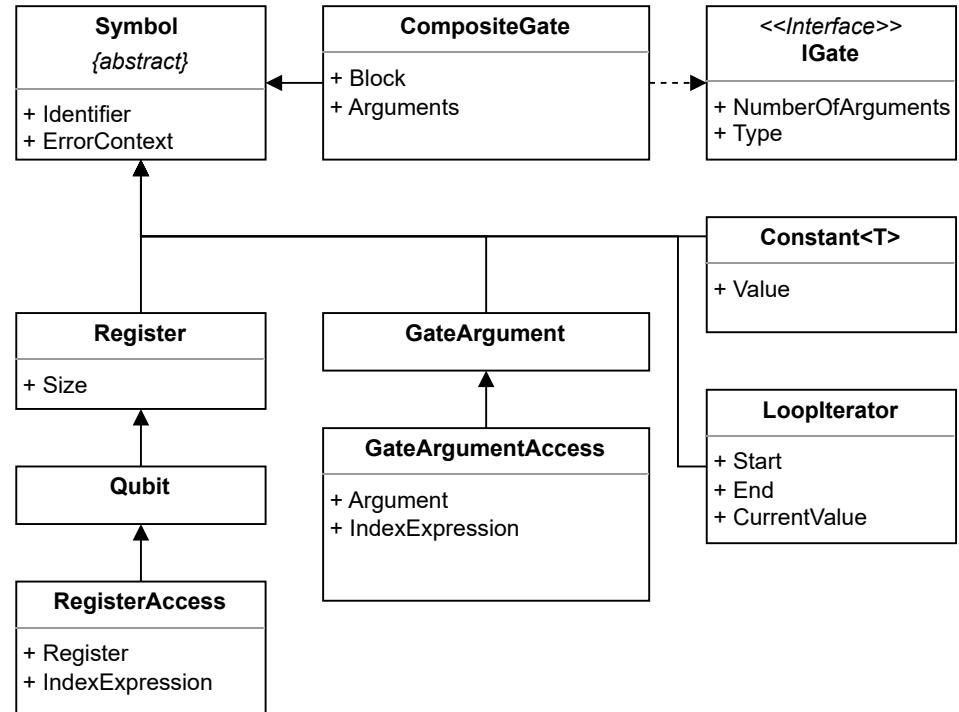
- The implementation of the compiler is differentiated into four different stages:
 1. the lexical and syntactic analysis,
 2. semantic analysis,
 3. code generation, and
 4. optimizations
- The process is managed by a static compiler class.
- It parses the command line parameters, handles the input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

```
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 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.

Implementation

Code Generation

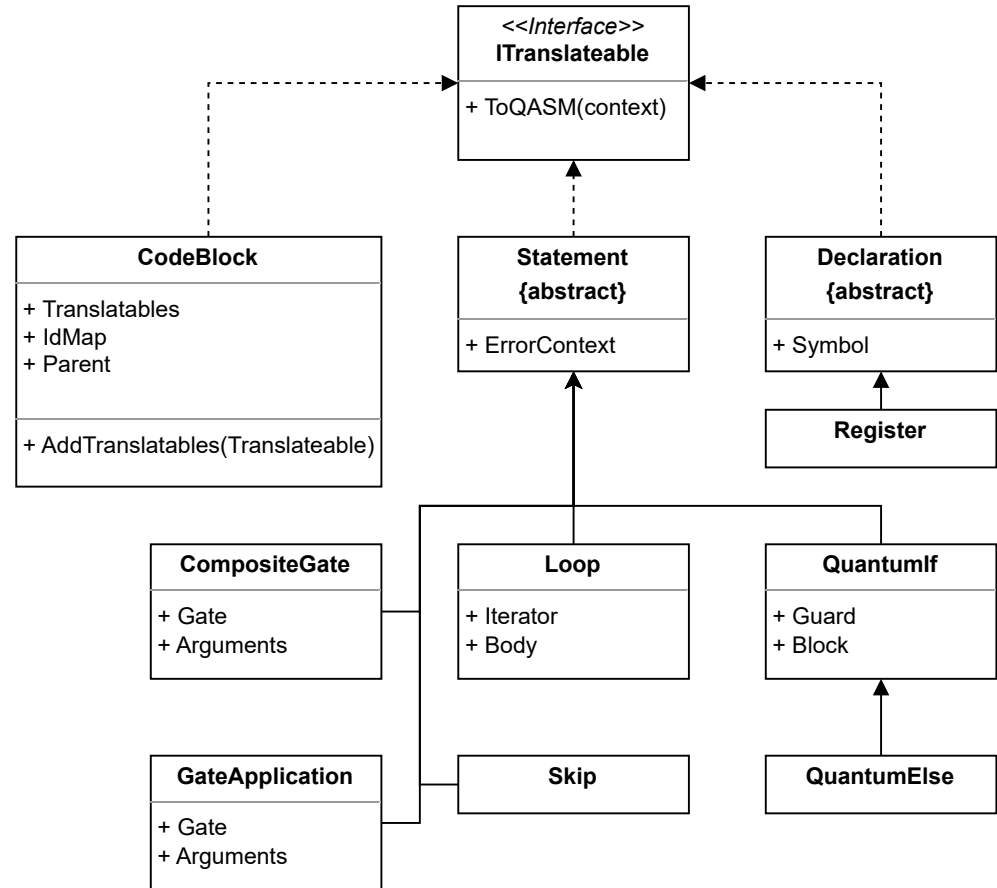
- First, the parse tree is traversed and the source code is translated to an in-memory representation.
- Next, this source code representation (SCR) is translated to the target code representation (TCR).
- Then, the TCR can be translated directly to the textual OpenQASM code.
- We want to go through 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 Luie 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.

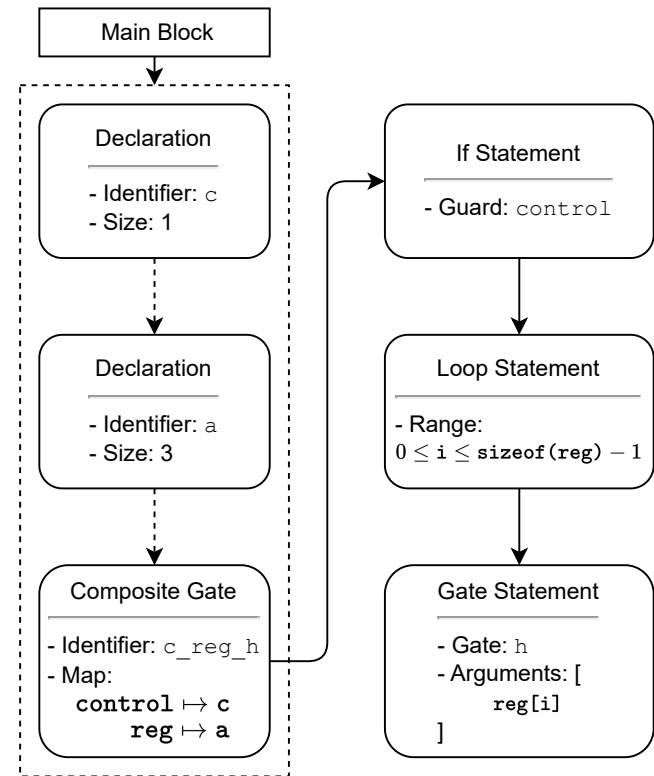
Implementation

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 source code representation of the example program.

Target Code Representation — Example

-

Implementation

Optimization

...






Evaluation

Evaluation

...

Conclusion

References

-  Aaby, A. A. (2003).
Compiler construction using flex and bison.
-  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.
-  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.
-  Yuan, C., Villanyi, A., and Carbin, M. (2024).
Quantum control machine: The limits of control flow in quantum programming.

References

Proceedings of the ACM on Programming Languages, 8(OOPSLA1):1–28.