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

test [Aaby, 2003]

Background

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*:

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

- Later, the concept was formally defined by [Ying et al., 2012].
- Quantum branching allows fot the execution of function 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 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 by reversibly modified.
- To jump by 5, the *bcr* is increased by 5 and, at its destination, decreased by 5 again.

Intructions

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.

Background

(Non-) Reversible Example

```
      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

```
            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
  - Quantum if- and else-statements

Language

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 of declarations.

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

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

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.

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

Language

Example Program

```
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 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

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

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 translatables, i.e., statements and declarations.

$$
\begin{aligned}
\mathit{trans} &: \mathit{Program} \dashrightarrow \mathit{QASMProgam} \\
\mathit{trans}(\mathit{gDcl}_1 \ldots \mathit{gDcl}_n\ \mathit{blk}) &= \mathit{bt}(\mathit{blk}, \mathit{update}(\mathit{update}(\mathit{update}(\mathit{st}_\epsilon, \mathit{gDcl}_1), \ldots), \mathit{gDcl}_n))
\end{aligned}
$$

$$
\begin{aligned}
\mathit{bt} &: \mathit{Block} \times \mathit{SymbolTable} \dashrightarrow \mathit{QASM} \\
\mathit{bt}(t_1 \ldots t_n, \mathit{st}_1) &= \mathit{tr}_1 \quad \text{where } (\mathit{tr}_1, \mathit{st}_2) = \mathit{tt}(t_1, \mathit{st}_1) \\
&\ \ \ldots \\
&\ \ \mathit{tr}_n \quad \text{where } (\mathit{tr}_n, \_) = \mathit{tt}(t_n, \mathit{st}_n)
\end{aligned}
$$

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.

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

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

$$\begin{aligned}
\mathit{control}(\mathit{qArg}, \quad & \mathit{id}\ \mathit{qArg}_1, \ldots, \mathit{qArg}_{n'}\,\texttt{;}\,) = \\
\texttt{ctrl(1) @} \quad & \mathit{id}\ \mathit{qArg}, \mathit{qArg}_1, \ldots, \mathit{qArg}_{n'}\,\texttt{;} \\
\mathit{control}(\mathit{qArg}, \texttt{ctrl(}n\texttt{) @} \quad & \mathit{id}\ \mathit{qArg}_1, \ldots, \mathit{qArg}_{n'}\,\texttt{;}\,) = \\
\texttt{ctrl(}n+1\texttt{) @} \quad & \mathit{id}\ \mathit{qArg}, \mathit{qArg}_1, \ldots, \mathit{qArg}_{n'}\,\texttt{;}
\end{aligned}$$

Implementation

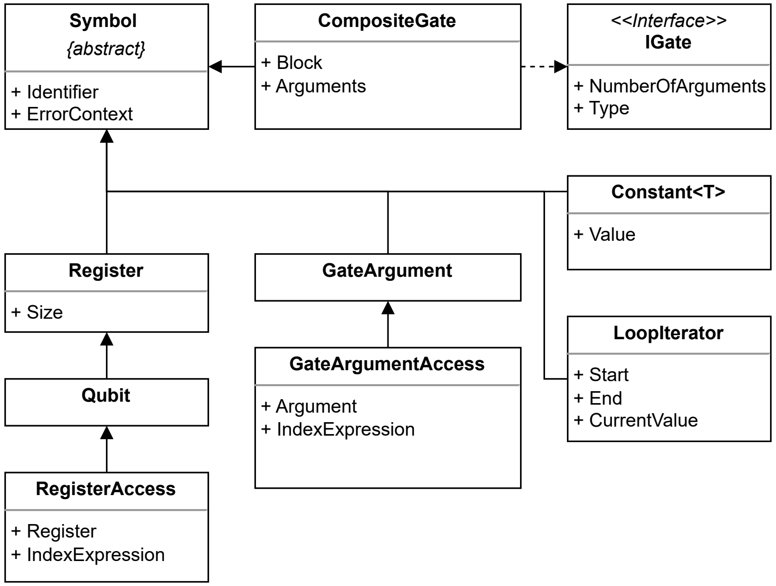
Overview

- The implementation of the compiler is differentiate into four different stages:
  1. Lexical and Syntactic Analysis
  2. Semantic Analysis
  3. Code Generation
  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



A diagram showing the hierarchy of symbol classes.

Code Generation

-

Code Generation

-

Implementation

Code Generation

-

Optimization

...

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.