λ_Q : A Simple Quantum Programming Language

Wenhao Tang 1800013088
Xinzhao Wang 1800013102

Department of EECS
Peking University

June 30, 2021

Contents

1 Introduction

The idea of quantum computation was first proposed by Richard Feynman in 1981 to simulate quantum systems that are too hard to simulate using conventional classical digital computers.

In 1994, Peter Shor found an efficient quantum algorithm for factoring large numbers, which caused great intrest in quantum computation for its implications for cryptanalysis.

Then, it was again Peter Shor who found quantum error-correcting codes and fault-tolerant methods for executing a quantum computation reliably using noisy hardware. It makes it possible for quantum computing to be scaled up to large devices that solve very hard problems in principal.

We often use NISQ to describe the current state of quantum computation. It stands for *Noisy Intermediate-Scale Quantum*. "Intermediate-Scale" conveys that today's device with more than 50 well-controlled qubits cannot be simulated by brute force using the most powerful currently existing classical computer. "noisy" means it is still not error corrected, which limits its computational power.

Although physical realization of universal quantum computation is still in a primitive stage, there has been a lot of work in software field like IBM's QisKit, Google's Cirq and Microsoft's Q♯. They provide software tools to describe and simulate quantum algorithms and give access of cloud quantum processors or simulators to wide community.

There are three main models of quantum computing: *Quantum Turing Machine*, *Quantum λ-Calculus* and *Quantum Circuit*, among which the third one is the most practical. Most quantum programming languages are *quantum-circuit description languages*, which means they are used to describe the architecture of quantum circuits. The current quantum programming languages can be categorized into two categories according to their styles: functional quantum programming languages and imperative quantum programming languages.

Functional:	Imperative:
• Qwire	• QASM
• QML	• QCL
• Quipper	• ScaffCC
• QuaFL	• Qiskit
• Silq	• Quil

Advanced quantum programming languages can use more powerful abstract constructs and type systems to make it easier for programmers to write correct quantum programs. In particular, we design a simple quantum programming language named λ_Q, which means **λ-calculus with quantum circuit**. Its syntax consists of a traditional part, which is a simple λ-calculus, and a quantum part, whose syntax is based on Qwire, a functional quantum programming language with linear type system. What's more, we implement a compiler from λ_Q to QASM (Quantum Assembly Language),

Figure 1: Structure of λ_Q compiler.

an imperative quantum programming language with low-level instruction sets. The output QASM program can run on the IBM cloud quantum machine.

The main feature of λ_Q is that the syntax for traditional computation and quantum computation are separated. They communicate with each other via some specific operations : quantum circuit can be *abstracted* or *lifted (measured)* into traditional term, and traditional term for quantum circuit can be applied to quantum bits. Thus, the syntax of quantum circuit can use *linear type system* to guarantee that the Quantum Non-cloning Theorem is not violated, meanwhile the λ-calculus of the traditional part makes it easier to write quantum programs. We will explain it in detail in Section 2.

The overall structure of the compiler can be visualized in Figure 1. The frontend is implemented using `Haskell`, and the backend is implemented using `C`. The code can be found in `https://github.com/thwfhk/lambdaQ`. We will discuss the implementation of λ_Q compiler in detail in Section 3 and 4.

2 Specification of λ_Q

In this section we give the language specification of λ_Q, including the syntax, typing rules and syntactic sugars. The syntax and typing rules of the quantum circuit are similar to Qwire [0]. The operational semantics are not given in this section. Instead, we give a transformation from λ_Q to $QASM$ in the section of frontend, which gives semantics to each syntax of λ_Q. Interested readers can refer to the operational semantics of simply-typed λ-calculus and Qwire.

2.1 Syntax

The syntax of the traditional part (terms, values, types) is an extended version of simply-typed λ-calcus with `run` C and $\kappa\ p : W.C$, which are used to interact with the quantum part. The syntax of the quantum part (circuits, wire types, wire patterns, gates, gate types) is similar to the syntax of Qwire.

t	::=	**terms**:

		x	variable
		`unit`	constant unit
		`true`	constant true
		`false`	constant false
		$\lambda\, x : T.t$	function abstraction
		$t\ t$	function application
		(t, t)	pair
		$t.1$	first projection
		$t.2$	second projection
		`if` t `then` t `else` t	conditional
		`run` C	static lifting
		$\kappa\, p : W.C$	circuit abstraction
v	::=		**values**:
		$\lambda\, x : T.t$	abstraction value
		(v, v)	pair value
		`unit`	unit value
		`true`	true value
		`false`	false value
		$\kappa\, p : W.C$	circuit value
T	::=		**types**:
		`Unit`	unit type
		`Bool`	boolean type
		$T \times T$	product type
		$T \to T$	function type
		$T \rightsquigarrow T$	circuit type
Γ	::=		**contexts**:
		$\varnothing$	empty context
		$\Gamma, x : T$	term variable binding
W	::=		**wire types**:
		`1`	wire unit type
		`Bit`	bit type
		`Qubit`	qubit type
		$W \otimes W$	wire product type
p	::=		**wire patterns**:
		$()$	empty
		w	wire variable
		(p, p)	wire pair
C	::=		**circuits**:
		`output` p	output a pattern
		$p_2 \leftarrow$ `gate` $g\ p_1; C$	gate application
		$p \leftarrow C; C$	circuit composition
		$x \hookleftarrow$ `lift` $p; C$	dynamic lifting

		$\texttt{capp}\ t\ \texttt{to}\ p$	circuit application
g	::=		**gates**:
		$\texttt{new}_0$	generate a bit 0
		$\texttt{new}_1$	generate a bit 1
		$\texttt{init}_0$	generate a qubit 0
		$\texttt{init}_1$	generate a qubit 1
		`meas`	measurement gate
		`discard`	disgard gate
		`H`	Hadamard gate
		`X`	Pauli-X gate
		`Z`	Pauli-Z gate
		`CNOT`	CNOT gate
G	::=		**gate types**:
		$\mathcal{G}(W, W)$	simple gate type
Ω	::=		**wire contexts**:
		$\varnothing$	empty context
		$\Omega, w : W$	wire variable binding

2.2 Typing Rules

The main feature of the language design of λ_Q is to use linear type system to guarantee no quantum bit is used twice or not used at all. To state the type inference rules of λ_Q, we first need to define what is a **well-formed wire context**, which is used to maintain variables used by quantum circuit in the calculus. This context is actually corresponding to the context of linear variables in the traditional linear type system.

Definition 1 (Well-formed Wire Contexts)**.** *A wire context Ω is well-formed, if there are no duplicate wire variables in it. For simplicity, we always assume the wire contexts are well-formed in the following contexts. And when we write Ω_1, Ω_2, we require Ω_1 and Ω_2 to be disjoint to preserve the well-formedness.*

Since there are some different kinds of terms: (λ-)terms, wire patterns, gates and circuits, we have defined several different typing relations for each of them.

- $\Omega \vdash p : W$ is the typing relation for patterns.
- $\Gamma; \Omega \vdash C : W$ is the typing relation for circuits;
- $\Gamma \vdash t : T$ is the typing relation for (λ-)terms;
- $g : G$ is the typing relation for gates.

2.2.1 Typing rules for gates

Note that since we only support built-in gates, the typing rules for gates are extremely simple, just assigning a type to each built-in gate.

Typing rules for gates: $\boxed{g : G}$

$$\frac{}{\texttt{new}_0 : \mathcal{G}(\texttt{1}, \texttt{Bit})}$$

$$\frac{}{\texttt{new}_1 : \mathcal{G}(\texttt{1}, \texttt{Bit})}$$

$$\frac{}{\texttt{init}_0 : \mathcal{G}(\texttt{1}, \texttt{Qubit})}$$

$$\frac{}{\texttt{init}_1 : \mathcal{G}(\texttt{1}, \texttt{Qubit})}$$

$$\frac{}{\texttt{meas} : \mathcal{G}(\texttt{Qubit}, \texttt{Bit})}$$

$$\frac{}{\texttt{discard} : \mathcal{G}(\texttt{Bit}, \texttt{1})}$$

2.2.2 Typing rules for patterns

The patterns are used to construct complex gates. Patterns are destructed when doing pattern matching in gate application, circuit composition and dynamic lifting.
Typing rules for patterns: $\boxed{\Omega \vdash p : W}$

$$\frac{}{\varnothing \vdash () : One}$$

$$\frac{}{w : W \vdash w : W}$$

$$\frac{\Omega_1 \vdash p_1 : W_1 \quad \Omega_2 \vdash p_2 : W_2}{\Omega_1, \Omega_2 \vdash (p_1, p_2) : W_1 \otimes W_2}$$

2.2.3 Typing rules for circuits

The typing rules for circuit uses linear type. The context Γ is for normal (traditional) variables and the context Ω is for linear (quantum) variables.
Typing rules for circuits: $\boxed{\Gamma; \Omega \vdash C : W}$

$$\frac{\Omega \vdash p : W}{\Gamma; \Omega \vdash \texttt{output}\ p : W} \quad \text{(C-OUTPUT)}$$

$$\frac{g : \mathcal{G}(W_1, W_2) \quad \Omega_1 \vdash p_1 : W_1 \quad \Omega_2 \vdash p_2 : W_2 \quad \Gamma; \Omega_2, \Omega \vdash C : W}{\Gamma; \Omega_1, \Omega \vdash p_2 \leftarrow \texttt{gate}\ g\ p_1; C : W} \quad \text{(C-GATE)}$$

$$\frac{\Gamma; \Omega_1 \vdash C : W \quad \Omega \vdash p : W \quad \Gamma; \Omega, \Omega_2 \vdash C' : W'}{\Gamma; \Omega_1, \Omega_2 \vdash p \leftarrow C; C' : W'} \quad \text{(C-COMPOSE)}$$

$$\frac{\Omega \vdash p : W \quad \Gamma, x : |W|; \Omega' \vdash C : W'}{\Gamma; \Omega, \Omega' \vdash x \leftarrow \texttt{lift}\ p; C : W'} \quad \text{(C-LIFT)}$$

$$\frac{\Gamma \vdash t : W_1 \rightsquigarrow W_2 \quad \Omega \vdash p : W_1}{\Gamma; \Omega \vdash \texttt{capp}\ t\ \texttt{to}\ p : W_2} \qquad \text{(C-CAPP)}$$

2.2.4 Typing rules for terms

Typing rules for (λ-)terms: $\boxed{\Gamma \vdash t : T}$

$$\frac{\Gamma; \varnothing \vdash C : W}{\Gamma \vdash \texttt{run}\ C : |W|} \qquad \text{(T-RUN)}$$

$$\frac{\Omega \vdash p : W_1 \quad \Gamma; \Omega \vdash C : W_2}{\Gamma \vdash \kappa\ p : W_1.C : W_1 \rightsquigarrow W_2} \qquad \text{(T-CABS)}$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \text{(T-VAR)}$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda\ x : T_1.t_2 : T_1 \to T_2} \qquad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \qquad \text{(T-APP)}$$

$$\frac{}{\Gamma \vdash \texttt{true} : Bool} \qquad \text{(T-TRUE)}$$

$$\frac{}{\Gamma \vdash \texttt{false} : Bool} \qquad \text{(T-FALSE)}$$

$$\frac{\Gamma \vdash t_1 : \texttt{Bool} \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T}{\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 : T} \qquad \text{(T-IF)}$$

$$\frac{}{\Gamma \vdash \texttt{unit} : Unit} \qquad \text{(T-UNIT)}$$

$$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash (t_1, t_2) : T_1 \times T_2} \qquad \text{(T-PAIR)}$$

$$\frac{\Gamma \vdash t : T_1 \times T_2}{\Gamma \vdash t.1 : T_1} \qquad \text{(T-FST)}$$

$$\frac{\Gamma \vdash t : T_1 \times T_2}{\Gamma \vdash t.2 : T_2} \qquad \text{(T-SEC)}$$

2.3 Syntactic Sugar

In programming languages, it is useful to design some syntactic sugar to make it easier for programmer to programming. We introduce a syntactic sugar to λ_Q to abstract one common form of gate application.

$$p \leftarrow \texttt{gate}\ g\ p; \texttt{output}\ p\ =\ \texttt{gate}'\ g\ p$$

3 Frontend

In this section we illustrate the design and implementation of the frontend of compiler of λ_Q. The frontend consists of about 1300 lines of Haskell. [1] The code and file structure can be found in our repository.

3.1 Lexer & Parser

We use the `Parsec` package of Haskell to implement the lexer and parser. It is a parser-combinator library which is used to write a parser in a top-down style. This part together with the syntax definition consists of about 500 lines of Haskell code. There is nothing much special in this part. Interested readers can refer to our code.

3.2 Desugar

The process of desugaring is simply a top-down recursion on the abstract syntax tree.

3.3 Type Inference

The implementation of the type checker (or "inferencer") is the most tricky part of the frontend. Although it is only about 350 lines together with context management, there are a lot of details to deal with. The goal of type inference is to infer the type of a term based on the typing rules, meanwhile checking that the terms doesn't violate these rules. The type inference for linear types also requires to check every context for linear variables are *well-formed*, which is defined in Section 2.2.

The type inference for λ-calculus is standard. The novel part is that we develop an algorithm to check the linear type system used by λ_Q. In particular, we specify the order in which each component is checked for each syntax of quantum circuits to infer the wire type of quantum circuits meanwhile ensuring the context for linear (quantum) variables is well-formed. What's more, we use the mechanism of *monad transformers* to deal with the state and error information in the process of type inference. Readers can refer to our code and the typing rules in Section 2.2 for more details.

3.4 Intermediate Code Generation

There are two main problems for the code generation from λ_Q to the extended QASM. We briefly describe our solutions here.

The first is to maintain the register and variable bindings. Because QASM can use traditional bit as conditions for if-statements, we make use of the De Bruijn index to maintain the mapping

[1] The code of functional programming languages are usually much more dense than imperative ones.

from λ_Q variables to traditional bit variables in QASM. And we use a hash table and a counter to implement a register pool in a functional language, which is used to allocate new registers.

The second is to deal with the restricted expressiveness of QASM. QASM only supports condition statements, so we restrict the term in `capp` t `to` p to be the form of circuit abstraction or if-statements whose branches are all circuit abstractions. This restriction can be loosed further, but we do not implement it yet because it is already enough to use.

4 Backend

In this section we illustrate the design and implementation of the backend of compiler of λ_Q. The backend consists of four parts: lexer, parser, qubit allocation and optimization.

4.1 Lexer & Parser

We use Lex/Yacc to implement the lexer and parser of a modified version of OpenQasm2.0 in backend.

Subcircuit definition is not available in our OpenQasm2.0. Besides $\{U3, CX\}$ that are available in the original OpenQasm2.0, we add $\{H, X, Y, Z\}$ to the built-in gate set.

This part consists of 600 lines of C code formatted in Lex/Yacc file's syntax. Readers can refer to *lex.l* and *parser.y* to get more detailed information.

4.2 Qubit Allocation

In recent years, some companies like IBM have made quantum computers available to wide community. Users can build their experiments based on a circuit representation on the cloud platform. However, today's quantum computer prototypes have tight resources constraints. For instance, you can only apply two qubits operations to a subset of qubit pairs which is specificed by a partial network.

In *qubit_allocation.cpp*, we implement a heuristic algorithm to allocate physical qubits to logic qubits.

4.2.1 Problem Definition

The most basic form of Qubit Allocation Problem is: given a quantum circuit and an architecture, we want to know if it is possible to map logic qubits in the former to physical qubits in the latter. Notice that even this basic problem is NP-hard.

In some cases, the constraints of the architecture is impossible to satisfy, and we need some circuit transformation to relax these constraints.

There are three kinds of transformations we can apply:

Reversal Apply CX to (p, q) when (q, p) is available in the architecture.

Bridge Apply CX to (p, q) when (p, s) and (s, q) is available.

Swap Swap the state of p and q when (p, q) or (q, p) is available.

All these transformation need extra gates to implement, so we need to find a qubit mapping and circuit transformations to satisfy the constraints with minimum number of extra gates involved.

4.2.2 Algorithm

Our algorithm has two stages:

First stage: Find an intial mapping

We first sort the logic qubits in descending order of their occurrence counts.

Then, for each logic qubit q in order, we allocate q to a physical qubit with the nearest out-degree(both the quantum circuit and the architecture can be regarded as a directed graph whose vertex is qubit)

After q is allocated to p, for each edge (q, q'), we try to allocate q' to some p' with edge (p, p') and has the nearest out-degree.

Repeat the above process for q' if it is successfully allocated.

After this BFS-like allocation we allocate the rest unallocated logic qubit to a free physical qubit.

Second stage: Adjust the mapping and apply circuit transformation

The mapping l in the output of the first stage may not satisfy all constraints, so we need to adjust it and apply circuit transformations to the quantum circuit.

For any two qubits operation on (p, q), if $(l(p), l(q))$ can't be implemented in current mapping, then:

1. if (p, q) appears more than once in the circuit, we use a Swap transformation to move q closer to p (Here we can use BFS to find the shortest path to p) and then re-evaluate these four cases.
2. else if $(l(q), l(p))$ can be implemented, then we use a Reversal transformation to $l(q)$ and $l(p)$.
3. else if $\exists s$ s.t. $(l(p), s)$ and $(s, l(q))$ are both available, we can use a Bridge transformation to $l(p), s, l(q)$.
4. else use Swap transformation like case 1.

4.3 Optimization

Optimization for arbitary unitary gates is sophisticated, so we only consider cases when gates are chosen from a discrete set.

In this section, we implement two basic optimization method for gate set $\{H, RZ, CX, X\}$

Our work in this section can be divided into four parts: graph conversion, gate decomposition, Hardmard reduction and gate cancellation.

Readers can refer to *generator.cpp, optimization.cpp, graph.h* for detailed information.

4.3.1 Graph Conversion

At first, we only store quantum circuit in the AST constructed by yacc and write *generator.cpp* for code generation from AST (It retains this function in the final version)

However, tree structure is not suitable for gate optimization, since we often access the adjacent gate of a given gate, and they can be distant in AST.

Therefore, in *generator.cpp*, we construct a graph from AST whose edge connects gates that are adjacent in the circuit.

In *graph.h*, we implement function $Graph :: toposort$ to generate code from a given graph.

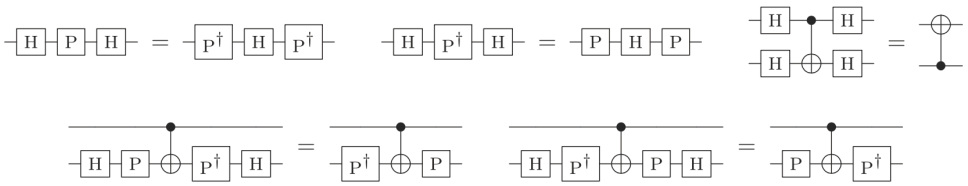

Figure 2: Subcircuit patterns in Hardmard Reduction

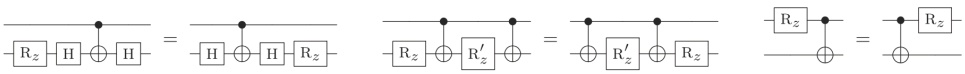

Figure 3: Commutation rules for RZ gate

4.3.2 Gate Decomposition

Since available gate set in our modified OpenQasm2.0 is different from the gate set on which we implement gate optimization, we should first decompose the given circuit in the new basis gate set.

Notice that $HXH = Z, ZX = iY, U(\theta, \phi, \lambda) = R_z(\phi)R_x(-\frac{\pi}{2})R_z(\theta)R_x(\frac{\pi}{2})R_z(\lambda)$ and $HR_x(\theta)H = R_z(\theta)$, it is easy to implement this decomposition.

4.3.3 Hardmard Reduction

In this section, we implement Hardmard Reduction to reduce the number of Hardmard gate.

It is a simple pattern (4.3.3) matching algorithm. We can traverse the graph, find serveral subcircuit pattern and replace it with a simplfied equivalent subcircuit.

4.3.4 Gate Cancellation

When a pair of conjugate transpose gates are adjacent, they can both be cancelled.

In this section, we implement gate cancellation for RZ gate, we traverse all RZ gates and try to move it by swapping with adjacent commutative subcircuit if posible (we only consider serveral built-in commutation rules (4.3.4) since it is hard to determine whether two gates are commutative) until encountering its conjugate transpose or reaching endpoints of the circuit.

5 Usage and Examples

See the `readme` file of our repository for code structure and usage. Here we give some examples of λ_Q program. Note that a λ_Q program should contain at least one circuit abstraction (i.e. κ abstraction) as its entry point, just like the main function in other programming languages. For simplicity, we assume the last circuit abstraction is the entry point.

5.1 Quantum Teleportation

This is a classical example also appeared in some other papers about quantum programming languages. It can be written in λ_Q like follows.

```
fun bell00 = κ () : One .
  a ← gate init0 ();
  b ← gate init0 ();
  a ← gate H a;
  (a, b) ← gate CNOT (a, b);
  output (a, b)

fun alice = κ (q, a) : Qubit # Qubit .
  (q, a) ← gate CNOT (q, a);
  q ← gate H q;
  x ← gate meas q;
  y ← gate meas a;
  output (x,y)

fun bob = κ ((w1, w2), q) : Bit # Bit # Qubit .
    (x1, x2) ↩ lift (w1, w2);
    q ← capp (if x2
                then (κ t : Qubit . gate X t)
                else (κ t : Qubit . output t)
         ) to q;
    capp (if x1
          then (κ t : Qubit . gate Z t)
          else (κ t : Qubit . output t)
    ) to q

fun teleport = κ () : One .
  q ← gate init0 ();
  (a, b) ← capp bell00 to ();
  (x, y) ← capp alice to (q, a);
  capp bob to ((x, y), b);
```

5.2 Simple Optimization

Here is another simple example to show the optimization power of our backend. The original λ_Q program is:

```
fun qwq = κ () : One .
  a ← gate init0 ();
  b ← gate init0 ();
  a ← gate H a;
  b ← gate H b;
  (c, d) ← gate CNOT (a, b);
  c ← gate H c;
  d ← gate H d;
```

```
  output (c, d)
```

The output of the frontend is:

```
openqasm 2.0;
qreg r0[1];
qreg r1[1];
H r0;
H r1;
CX r0, r1;
H r0;
H r1;
```

And the output of the backend is:

```
openqasm 2.0;
qreg q[2];
CX q[1], q[0];
```

The optimization really works!

6 Roles and Responsibilities

Wenhao Tang:

- λ_Q language design;
- Implementation of Frontend;

Xinzhao Wang:

- Implementation of Backend;

twh: Supplement here @wxz.

Bibliography

[] J. Paykin, R. Rand, and S. Zdancewic. Qwire: a core language for quantum circuits. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 846–858, 2017.