λ_Q : A Simple Quantum Programming Language

Wenhao Tang 1800013088
Xinzhao Wang 1800013102

Department of EECS
Peking University

June 27, 2021

Contents

1 Motivation

Quantum computing is getting more and more popular these days. twh: Add some background of quantum computing here @wxz. blah
blah
blah
blah
blah

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), 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`. 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 [**?**]. 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.

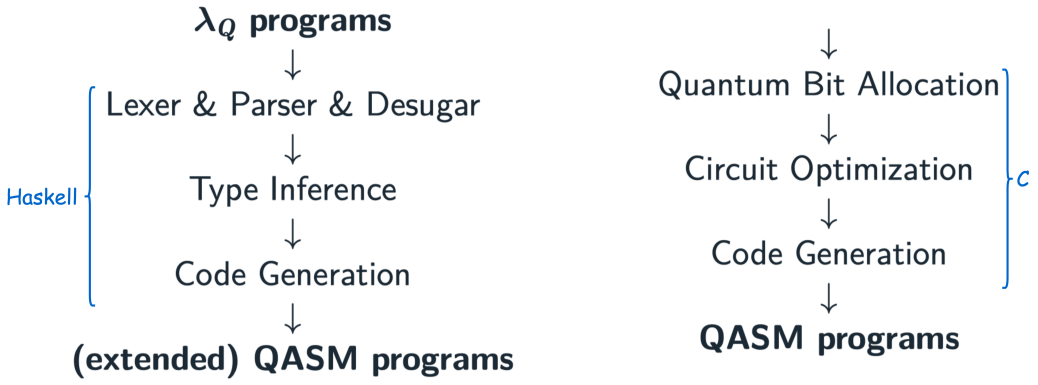

Figure 1: Structure of λ_Q compiler.

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

		$\texttt{Bool}$	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**:
		$\texttt{1}$	wire unit type
		$\texttt{Bit}$	bit type
		$\texttt{Qubit}$	qubit type
		$W \otimes W$	wire product type
p	::=		**wire patterns**:
		$()$	empty
		w	wire variable
		(p, p)	wire pair
C	::=		**circuits**:
		$\texttt{output}\ p$	output a pattern
		$p_2 \leftarrow \texttt{gate}\ g\ p_1; C$	gate application
		$p \leftarrow C; C$	circuit composition
		$x \leftarrow \texttt{lift}\ p; C$	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
		$\texttt{meas}$	measurement gate
		$\texttt{discard}$	disgard gate
		$\texttt{H}$	Hadamard gate
		$\texttt{X}$	Pauli-X gate
		$\texttt{Z}$	Pauli-Z gate
		$\texttt{CNOT}$	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: $\boxed{\Gamma; \Omega \vdash C : W}$

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

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

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

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

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

2.2.4 Typing rules for terms

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

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

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

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

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

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

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

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

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

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

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

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

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

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]

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 defini-

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

tion 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 term doesn't violate these rules. The type inference for linear type 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 checker for each syntax of quantum circuits to inference 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 for more details.

3.4 Intermediate Code Generation

There are two main problems for the code generation from λ_Q to the extended QASM. The first is to maintain the register and variable bindings. We make use of the De Bruijn index maintain the traditional bit variables mapping. 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 statement 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. This part together with the syntax of extended QASM consists of about 300 lines.

4 Backend