λ_Q : A Simple Quantum Programming Language

Wenhao Tang, Xinzhao Wang

EECS, PKU
June 18, 2021

Repo: `github.com/thwfhk/lambdaQ`

Outline

What is Quantum Computing?

blahblah

Three models of Quantum Computing:

1. Quantum Turing Machine
2. Quantum Lambda Calculus
3. **Quantum Circuit** (practical)

What is a Quantum Programming Language?

Most quantum languages are quantum circuit description languages.
Taxonomy of current QPL:

Functional:

- **Qwire**
- QML
- Quipper
- QuaFL
- Silq

Imperative:

- **QASM**
- QCL
- ScaffCC
- Qiskit
- Quil

Overview of λ_Q Compiler

λ_Q programs

↓

Lexer & Parser & Desugar

Haskell

↓

Type Inference

↓

Code Generation

↓

(extended) QASM programs

↓

Quantum Bit Allocation

↓

Circuit Optimization

C

↓

Code Generation

↓

QASM programs

Definition of λ_Q

$\boldsymbol{\lambda}$-Calculus with **Q**uantum Circuit.
Features:

- Separate **classical** part and **quantum** circuit.
- **Linear type system** for *Quantum No-cloning Theorem*.

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
		`capp` t `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
G	$::=$		**gate types**:
		$\mathcal{G}(W, W)$	simple gate type
Ω	$::=$		**wire contexts**:
		$\varnothing$	empty context
		$\Omega, w : W$	wire variable binding

Lexer & Parser & Desugar

Use **Parsec** package of Haskell to implement Lexer and Parser. Parsec is a **parser-combinator** library, which is a different approach to implementing parsers than parser generator.

```haskell
parseTyCir :: Parser Type
parseTyCir = do
  t1 ← parseWtype
  reservedOp "~>"
  t2 ← parseWtype
  return $ TyCir t1 t2

parsePrimType :: Parser Type
parsePrimType = (whiteSpace >>) $
      parseTyUnit
  <|> parseTyBool
  <|> try parseTyCir
  <|> parens parseType
```

Type Inference

Use **linear type system** for the quantum part, which guarantees that no quantum bit will be used twice.

Typing rules for circuits: $\boxed{\Gamma; \Omega \vdash C : W}$

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

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

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

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

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

Figure: The core part of type inference rules

Code Generation

- Nothing special.
- Use **De Bruijn Index** to maintain variables mapping.
- Some compromises on the expressiveness of QASM.

Qubit Allocation

- Problem
  Because of hardware constraints, not all qubits is physically connected in a real quantum computer.
- Goal
  Allocate logic qubits to physical qubits to satisfy hardware constraints.

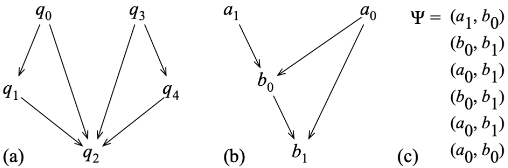

Figure 2. (a) The coupling graph of the IBM qx2 computer. (b) Interactions between qubits of the circuit seen in Figure 1. (c) Dependences that have created these interactions.

Qubit Allocation

We can use the three circuit transformations below with extra cost.

- Reversal
  Reverse the direction of one CNOT gate.
- Bridge
  If $a \to b$ and $b \to c$, we can implement CNOT from a to c.
- Swap
  Swap the states between two physical qubits.

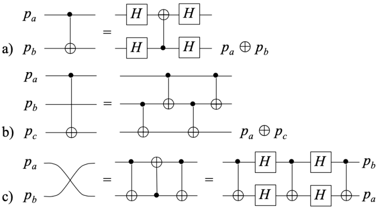

Figure 3. (a) Reversal. (b) Bridge. (c) Swap.

Qubit Allocation

We use a two stage heuristics algorithm to solve this problem.

- Find an initial mapping
  - Sort logic qubits by their occurrence frequency
  - BFS and greedy
- Extend the initial mapping
  - If an edge (p_0, p_1) appears two or more times, swap p_1 closer to p_0 and re-evaluate
  - else check if Reversal and Bridge can be used
  - else swap p_1 closer to p_0 and re-evaluate

Circuit Optimization

Pattern Matching

- Find a subcircuit of some patterns and replace it by another subcircuit.

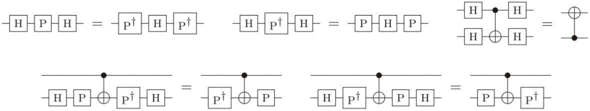

Fig. 4 Hadamard gate reductions. The two rules illustrated on the bottom can be applied even if the middle CNOT gate is replaced by a circuit with any number of CNOT gates, provided they all share the target of the original CNOT

Circuit Optimization

Gate Cancellation

- If U and $U^\dagger$ is adjacent, both of them can be removed.
- Commutation rules ($UV = VU$)

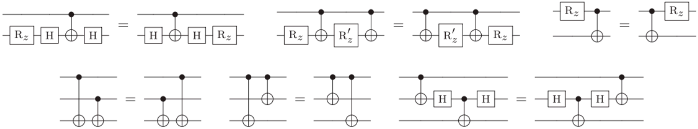

Fig. 5 Commutation rules. Top: Commuting an R_z gate to the right. Bottom: Commuting a CNOT gate to the right

Examples

```
-- bell00 : Circ(1, qubit ⊗ qubit)
fun bell00 = / () : One .
  a ← gate init0 ();
  b ← gate init0 ();
  a ← gate H a;
  (a, b) ← gate CNOT (a, b);
  output (a, b)

-- alice : Circ(qubit ⊗ qubit, bit ⊗ bit)
fun alice = / (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)

-- bob : Circ(bit ⊗ bit ⊗ qubit, qubit)
fun bob = / ((w1, w2), q) : Bit # Bit # Qubit .
    (x1, x2) ⤆ lift (w1, w2);
    q ← capp (if x2
               then (/ t : Qubit . gate X t)
               else (/ t : Qubit . output t)
         ) to q;
    capp (if x1 then (/ t : Qubit . gate Z t) else
    (/ t : Qubit . output t)) to q

-- teleport : Circ(1, qubit)
fun teleport = / () : One .
  q ← gate init0 ();
  (a, b) ← capp bell00 to ();
  (x, y) ← capp alice to (q, a);
  capp bob to ((x, y), b);
```

```
[PARSE SUCCESS 🥳]: 4 functions founded.
[TYPE SUCCESS 🥳]:
  bell00 : 1 ⇝ Qubit ⊗ Qubit
  alice : Qubit ⊗ Qubit ⇝ Bit ⊗ Bit
  bob : Bit ⊗ Bit ⊗ Qubit ⇝ Qubit
  teleport : 1 ⇝ Qubit
[GENERATION SUCCESS 🥳]:
  qreg r0[1];
  qreg r1[1];
  qreg r2[1];
  H r1;
  CX r1, r2;
  CX r0, r1;
  H r0;
  measure r0 → r3;
  measure r1 → r4;
  if (r4 == 1) X r2;
  if (r3 == 1) Z r2;
```

```
openqasm 2.0;
qreg a[5];
CX a[0], a[1];
CX a[1], a[2];
CX a[2], a[3];
CX a[3], a[4];
CX a[0], a[4];
```

```
OPENQASM 2.0;
qreg q[5];
CX q[0], q[1];
CX q[1], q[2];
CX q[2], q[3];
CX q[3], q[4];
// swap 4 3
CX q[3], q[4];
H q[3];
H q[4];
CX q[3], q[4];
H q[3];
H q[4];
CX q[3], q[4];
// swap 4 2
CX q[2], q[3];
H q[2];
H q[3];
CX q[2], q[3];
H q[2];
H q[3];
CX q[2], q[3];
// bridge 0 1 2
CX q[1], q[2];
CX q[0], q[1];
CX q[1], q[2];
CX q[0], q[1];
```