Quantum Programming Languages

Evandro Chagas Ribeiro da Rosa

GIAA/GCQ-UFSC

31/08/2020

Table of Contents

Programming Languages

- Human-readable

Quantum Programming

The problem

Features:

- Superposition
- Entanglement
- Reversible

Limitations:

- No-cloning
- Reversible

Implementation constraints:

- Number of qubits
- Decoherence
- Execution environment

Quantum Programming

Quantum Programming Libraries: A different approach for the same problem

Popular quantum programming libraries:

- Cirq
- ProjectQ
- PyQuil
- Qiskit

Classification

Quantum Assembly:

- OpenQASM
- Quil
- Blackbird[1]
- QMASM[2]

Q. Circuit Description:

- LIQ$Ui|\rangle$
- QWIRE
- Quipper
- Scaffold

High-level Lang.:

- Q#
- Silq
- Ket

[1] Continuous-variable quantum optical circuits

[2] Adiabatic quantum computers

Quantum Assembly Languages
OpenQASM

- IBM Quantum Experience
- Quantum circuit

Quantum Assembly Languages

OpenQASM

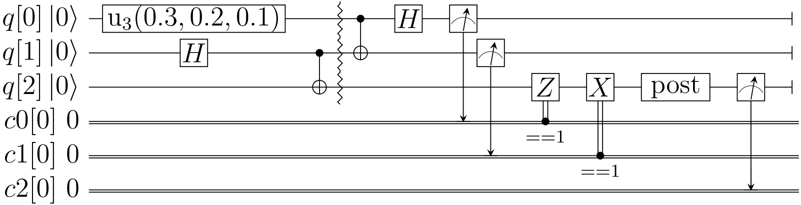
Figure 1: Quantum teleportation example [CBSG17].

```
// quantum teleportation example
OPENQASM 2.0;
include "qelib1.inc";
qreg q[3];
creg c0[1];
creg c1[1];
creg c2[1];
// optional post-rotation for state tomography
gate post q { }
u3(0.3,0.2,0.1) q[0];
h q[1];
cx q[1],q[2];
barrier q;
cx q[0],q[1];
h q[0];
measure q[0] -> c0[0];
measure q[1] -> c1[0];
if(c0==1) z q[2];
if(c1==1) x q[2];
post q[2];
measure q[2] -> c2[0];
```

Quantum Assembly Languages

OpenQASM

Figure 2: Quantum teleportation circuit [CBSG17].



Quantum Assembly Languages
OpenQASM

Table 1: Open QASM (2.0) language statements [CBSG17].

Statement	Description	Example
`OPENQASM 2.0;`	Denotes a file in Open QASM format[a]	`OPENQASM 2.0;`
`qreg name[size];`	Declare a named register of qubits	`qreg q[5];`
`creg name[size];`	Declare a named register of bits	`creg c[5];`
`include "filename";`	Open and parse another source file	`include "qelib1.inc";`
`gate name(params) qargs { body }`	Declare a unitary gate	(see text)
`opaque name(params) qargs;`	Declare an opaque gate	(see text)
`// comment text`	Comment a line of text	`// oops!`
`U(theta,phi,lambda) qubit|qreg;`	Apply built-in single qubit gate(s)	`U(pi/2,2*pi/3,0) q[0];`
`CX qubit|qreg,qubit|qreg;`	Apply built-in CNOT gate(s)	`CX q[0],q[1];`
`measure qubit|qreg -> bit|creg;`	Make measurement(s) in Z basis	`measure q -> c;`
`reset qubit|qreg;`	Prepare qubit(s) in $\|0\rangle$	`reset q[0];`
`gatename(params) qargs;`	Apply a user-defined unitary gate	`crz(pi/2) q[1],q[0];`
`if(creg==int) qop;`	Conditionally apply quantum operation	`if(c==5) CX q[0],q[1];`
`barrier qargs;`	Prevent transformations across this source line	`barrier q[0],q[1];`

[a] This must appear as the first non-comment line of the file.

Quantum Assembly Languages
Quil

- Forest SDK - Rigetti
- Classical and quantum States
- `DEFGATE`; `DEFCIRCUIT`

Figure 3: Quil example [SCZ16].

```
LABEL @START
H 0
MEASURE 0 [0]
JUMP-WHEN @END [0]
H 0
H 1
CNOT 1 0
JUMP @START
LABEL @END
Y 0
MEASURE 0 [0]
MEASURE 1 [1]
```

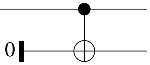
Quantum Circuit description languages

Quipper

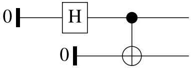
- Scalable quantum circuit description
- Haskell embedded language

Figure 4: Quantum teleportation example [GLR+13].

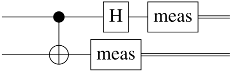
```
plus_minus :: Bool -> Circ Qubit
plus_minus b = do
    q <- qinit b
    r <- hadamard q
    return r

share :: Qubit -> Circ (Qubit, Qubit)
share a = do
  b <- qinit False
  b <- qnot b `controlled` a
  return (a,b)
```



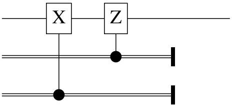
```
bell00 :: Circ (Qubit, Qubit)
bell00 = do
  a <- plus_minus False
  (a,b) <- share a
  return (a,b)
```



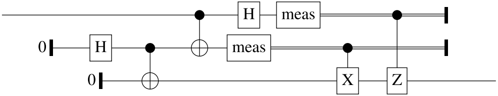

```
alice :: Qubit -> Qubit -> Circ (Bit,Bit)
alice q a = do
    a <- qnot a ‘controlled‘ q
    q <- hadamard q
    (x,y) <- measure (q,a)
    return (x,y)
```




```
bob :: Qubit -> (Bit,Bit) -> Circ Qubit
bob b (x,y) = do
    b <- gate_X b ‘controlled‘ y
    b <- gate_Z b ‘controlled‘ x
    cdiscard (x,y)
    return b
```




```
teleport :: Qubit -> Circ Qubit
teleport q = do
    (a,b) <- bell00
    (x,y) <- alice q a
    b <- bob b (x,y)
    return b
```



High-level languages
Q#

- Beyond quantum circuit
- Classical and quantum computation
- Python and .NET languages (C#, F#)
- Function (`'T -> 'U`)
- Operation (`'T => 'U`)
- `Controlled`; `Adjoint`; (`'T => Unit is Ctl + Adj`)

Figure 5: Approximated QFT of Q# [SGT+18].

```
namespace Microsoft.Quantum.Canon {
    open Microsoft.Quantum.Primitive;

    operation ApproximateQFT ( a: Int, qs: BigEndian) : () {
        body {
            let nQubits = Length(qs);

            for (i in 0 .. (nQubits - 1) ) {
                for (j in 0..(i-1)) {
                if ( (i-j) < a ) {
                    (Controlled R1Frac)( [qs[i]], (1, i - j, qs[j]) );
                    }
                }
                H(qs[i]);
            }

            // Apply the bit reversal permutation
            // to the quantum register
            SwapReverseRegister(qs);
        }

        adjoint auto
        controlled auto
        controlled adjoint auto
    }
}
```

High-level languages

Silq

- Safe uncomputation and intuitive semantics
- Linear type system* (`const`; duplication)

Figure 6: Benefit of Silq's automatic uncomputation [BBGV20]

```
d := a || b || c;
```

Silq

```
with_computed (OR a b) $
  \t -> OR t c
```

Quipper

```
using(t=Qubit()){
  OR(a,b,t);
  OR(t,c,d);
  Adjoint OR(a,b,t);
}
```

Q#

Figure 7: Examples of invalid Silq programs, their error messages, and possible fixes (where applicable) [BBGV20].

```
def useConsumed(x:𝔹){
  y := H(x); // consumes x
  return (x,y);
} // undefined identifier x

def useConsumedFixed(const x:𝔹){
  // ψ1 = Σ_{v=0}^{1} γ_v |v⟩_x
  // ψ2 = Σ_{v=0}^{1} γ_v |v⟩_x ⊗ |v⟩_x
  y := H(x);
  // ψ3 = Σ_{v=0}^{1} γ_v |v⟩_x ⊗ (|0⟩_y + (−1)^v |1⟩_y)
  return (x,y);
}
```

```
def discard[n:!ℕ](x:uint[n]){
  y := x % 2; // '%' supports quantum inputs
  return y;
} // parameter 'x' is not consumed

def nonQfree(const x:𝔹,y:𝔹){
  if H(x) { y := X(y); }
  return y;
} // non-lifted quantum expression must be consumed

def nonConst(c:𝔹){
  if X(c) { phase(π); } // X consumes c
} // non-lifted quantum expression must be consumed
```

```
def nonConstFixed(const c:𝔹){
  // ψ1 = Σ_{v=0}^{1} γ_v |v⟩_c
  if X(c) { phase(π); }
  // ψ2 = Σ_{v=0}^{1} (−1)^{1−v} γ_v |v⟩_c
}

def condMeas(const c:𝔹,x:𝔹){
  if c { x := measure(x); }
} // cannot call function
// 'measure[𝔹]' in 'mfree' context

def revMeas(){
  return reverse(measure);
} // reversed function must be mfree
```

High-level languages

Ket

- Cloud-based quantum computation
- Generic quantum programming
- Dynamic quantum execution
- Runtime quantum code generation; Future

```python
def bell(aux0, aux1):
    q = quant(2)
    if aux0 == 1:
        x(q[0])
    if aux1 == 1:
        x(q[1])
    h(q[0])
    ctrl(q[0],x,q[1])
    return q

def teleport(a):
    b = bell(0, 0)
    ctrl(a, x, b[0])
    h(a)
    m0 = measure(a)
    m1 = measure(b[0])
    if m1 == 1:
        x(b[1])
    if m0 == 1:
        z(b[1])
    return b[1]

a = quant(1)
b = teleport(a)
result = measure(b)
result.get()
```

```
LABEL @entry
      ALLOC   q0
      ALLOC   q1
      ALLOC   q2
      H       q1
      CTRL    q1   X         q2
      CTRL    q0   X         q1
      H       q0
      MEASURE q0
      INT     i0   ZE        c0
      MEASURE q1
      INT     i1   ZE        c1
      INT     i2   1
      INT     i3   i1        ==     i2
      BR      i3   @if.then0        @if.end1
LABEL @if.then0
      X       q2
      JUMP    @if.end1
LABEL @if.end1
      INT     i4   1
      INT     i5   i0        ==     i4
      BR      i5   @if.then2        @if.end3
LABEL @if.then2
      Z       q2
      JUMP    @if.end3
LABEL @if.end3
      MEASURE q2
      INT     i6   ZE        c2
```

References I

[BBGV20] Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. Silq: a high-level quantum language with safe uncomputation and intuitive semantics. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 286–300, New York, NY, USA, jun 2020. ACM.

[CBSG17] Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. Open Quantum Assembly Language. jul 2017.

[GLR+13] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. An Introduction to Quantum Programming in Quipper. apr 2013.

[SCZ16] Robert S. Smith, Michael J. Curtis, and William J. Zeng. A Practical Quantum Instruction Set Architecture. aug 2016.

References II

[SGT+18] Krysta M. Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. Q#: Enabling scalable quantum computing and development with a high-level domain-specific language. mar 2018.

"Quantum computers raise interesting problems for the design of programming languages[...]" (Deutsch, 1985)