Quantum Programming Languages: Interpreters

Jens Palsberg

May 14, 2020

Outline

Hook: How do quantum simulators work? Which aspects are tricky and which ones are hard to scale? We have tried a few quantum simulators already and now the time has come to look under the hood.

Purpose: Persuade you that you can write your own quantum simulator in a day.

Preview:

1. We can pick a universal gate set and write a grammar.
2. We can build a convenient program representation.
3. We can represent quantum states and implement gate operations and measurement.

Transition to Body: Let us first design the quantum language.

Main Point 1: We can pick a universal gate set and write a grammar.
[Let us go with a small, universal set of gates]
[Our grammar can make programs look like they are out of a textbook]
[We should support comments and spacing]

Transition to MP2: We can do a lot of work up front to make simulation easy.

Main Point 2: We can build a convenient program representation.
[We can generate a parser automatically]
[We can generate a syntax tree automatically]
[We can map a syntax tree into a convenient data structure]

Transition to MP3: Let us call a simulator what is really is: an interpreter.

Main Point 3: We can represent quantum states and implement operations and measurement.
[We can represent a quantum state as a weighted sum of kets]
[We can implement each gate with a small piece of code]
[We can use a random-number generator to help implement measurement]

Transition to Close: So there you have it.

Review: We can implement a quantum simulator using classical principles for implementing interpreters. The grammar, the parser, and the program representation require little work, and the rest is straightforward implementation of well-understood operations.

Strong finish: Quantum simulators are essential for developing and debugging quantum programs. They need worst-case exponential time in the number of qubits and have so far scaled to 70 qubits.

Call to action: Apply all your CS knowledge to help build more scalable quantum simulators.

Detailed presentation

Hook: How do quantum simulators work? Which aspects are tricky and which ones are hard to scale? We have tried a few quantum simulators already and now the time has come to look under the hood.

Purpose: Persuade you that you can write your own quantum simulator in a day.

Preview:

1. We can pick a universal gate set and write a grammar.
2. We can build a convenient program representation.
3. We can represent quantum states and implement gate operations and measurement.

Transition to Body: Let us first design the quantum language.

Main Point 1: We can pick a universal gate set and write a grammar.

[Let us go with a small, universal set of gates]
We know from the fourth lecture on foundations that for quantum computing, $\{CNOT, H, T\}$ is a universal set of gates. Here

$$T = \begin{pmatrix} 1 & 0 \\ 0 & \frac{1+i}{\sqrt{2}} \end{pmatrix}$$

We will use Deutsch-Jozsa as the main benchmark for the simulator and we know that $CNOT$ and H will be useful for that. However, Deutsch-Jozsa starts with the helper qubit as $|1\rangle$, which we can get as $X|0\rangle$. So, let us add X to the gate set. The result is that we want our gates to be $\{CNOT, H, T, X\}$.

[Our grammar can make programs look like they are out of a textbook]
Let us write a grammar that can support programs exactly in the style we want to write them.

$$\begin{array}{rcl} \textit{Program} & ::= & \texttt{circuit} : n \ \texttt{qubits} \ \ \textit{Instruction}^* \ \ \texttt{measure} \ i \ .. \ j \\ \textit{Instruction} & ::= & H(i) \ | \ T(i) \ | \ X(i) \ | \ CNOT(i,j) \end{array}$$

Notice that we specify the number of qubits up front; this is helpful both for the reader and for the interpreter. Notice also that we specify a range of qubits to be measured; this is sufficient for our benchmark programs.

Now we can write programs according to the grammar, such as the following implementation of the Deutsch-Jozsa algorithm.

```
// Deutsch-Jozsa for one of
// the balanced f : {0,1}^2 -> {0,1}^2

circuit: 3 qubits

X(2)

H(0)
H(1)
H(2)

// U_f
CNOT(0,2)

H(0)
H(1)

measure 0..1
```

[We should support comments and spacing]
Notice that the program uses comments and spacing. I achieved that by modifying a full-fledged grammar for a different language that supports comments and spacing. The grammar is in JavaCC notation.

Transition to MP2: We can do a lot of work up front to make simulation easy.

Main Point 2: We can build a convenient program representation.

[We can generate a parser automatically]
Once we have a grammar in JavaCC notation, we can use javacc to generate a parser. Given an input, the parser tells whether the input is a program according to the grammar, or where to find syntax errors.

[We can generate a syntax tree automatically]
We can combine the use of javacc with JTB, which enables the parser to build a syntax tree.

```
jtb quantum.jj
javacc jtb.out.jj
javac Main.java
java Main < dj2-bal1.q
```

Here `quantum.jj` contains the above grammar in JavaCC notation, while `dj2-bal1.q` contains the above program. The `Main.java` program calls the generated parser, which in turns maps `dj2-bal1.q` to a syntax tree. Then `Main.java` goes on process that syntax tree.

The generated parser is written in Java, the syntax tree is represented as a Java object, and all processing of the syntax tree is done in Java.

We have gotten to this point with just few keystrokes.

[We can map a syntax tree into a convenient data structure]
JTB generates a directory of classes that the parser uses to represent a syntax tree. For example, one of those classes is the following one, here in excerpt.

```
/**
 * Grammar production:
 * f0 -> OneQubitGate()
 * f1 -> "("
 * f2 -> IntegerLiteral()
 * f3 -> ")"
 */
public class OneQubitInstruction implements Node {
   public OneQubitGate f0;
   public NodeToken f1;
   public IntegerLiteral f2;
   public NodeToken f3;

   public OneQubitInstruction(OneQubitGate n0, IntegerLiteral n1) {
      f0 = n0;
      f1 = new NodeToken("(");
      f2 = n1;
      f3 = new NodeToken(")");
   }

   public void accept(visitor.Visitor v) {
      v.visit(this);
   }
}
```

The generated syntax tree supports the Visitor pattern via the `accept` method. The Visitor pattern enables us to write code that processes a syntax tree without requiring us to edit or recompile classes such as the one above.

I wrote a visitor (152 lines of well-commented code) that maps a syntax tree into the following Java data structure, here in excerpt.

```
public class Program {
  int size;
  ArrayList<Operation> ops;
  int measureLo;
  int measureHi;
}
```

The class `Operation` represents the application of a gate to its arguments. The code for `Program`, `Operation`, and two other classes is 73 lines.

The big win here is that the key data is readily available and that the list of operations is an `ArrayList`, which enables convenient processing.

Notice that we can easily extend the list of supported gates by expanding the grammar and adding to the mapper from a syntax tree to a `Program`.

Transition to MP3: Let us call a simulator what is really is: an interpreter.

Main Point 3: We can represent quantum states and implement operations and measurement.

[We can represent a quantum state as a weighted sum of kets]
A quantum state with n qubits can be written in the following form:

$$\Sigma_{x \in \{0,1\}^n} \; a_x|x\rangle$$

The order of the components $a_x|x\rangle$ is unimportant. Let us represent each $a_x|x\rangle$ as a `WeightedKet`, here in excerpt.

```
public class WeightedKet {
  public int size;
  public Complex amplitude;
  BitSet ket;

  public WeightedKet(int size) {
         this.size = size;
    this.amplitude = new Complex(1.0,0.0);
          this.ket = new BitSet(size);
  }
}
```

Now we can represent a quantum state as an `ArrayList`.

```
ArrayList<WeightedKet> state = new ArrayList<WeightedKet>();

public void run() {
  this.state.add(new WeightedKet(this.p.getSize()));

  for (Operation op : p.getOps()) {
    step(op);
    state.sort(new MyComparator());
    consolidate();
  }

  measure();
}
```

The initial state is is $|0^n\rangle$. The main loop executes the operations in order, with three things to be done for each operation:

1. execute the operation on every `WeightedKet` in the `state`;
2. sort according to the bit pattern in each ket.
3. combine kets with the same bit pattern.

For example, consider the following state before the final $H(1)$ in the program above.

$$
\begin{aligned}
& & & 0.5 \times |100\rangle - 0.5 \times |101\rangle + 0.5 \times |110\rangle - 0.5 \times |111\rangle \\
\text{after applying } H(1)\text{:} & & = & \tfrac{0.5}{\sqrt{2}} \times |100\rangle - \tfrac{0.5}{\sqrt{2}} \times |101\rangle - \tfrac{0.5}{\sqrt{2}} \times |110\rangle + \tfrac{0.5}{\sqrt{2}} \times |111\rangle + \\
& & & \tfrac{0.5}{\sqrt{2}} \times |110\rangle - \tfrac{0.5}{\sqrt{2}} \times |111\rangle + \tfrac{0.5}{\sqrt{2}} \times |100\rangle - \tfrac{0.5}{\sqrt{2}} \times |101\rangle \\
\text{after sorting:} & & = & \tfrac{0.5}{\sqrt{2}} \times |100\rangle + \tfrac{0.5}{\sqrt{2}} \times |100\rangle - \tfrac{0.5}{\sqrt{2}} \times |101\rangle - \tfrac{0.5}{\sqrt{2}} \times |101\rangle - \\
& & & \tfrac{0.5}{\sqrt{2}} \times |110\rangle + \tfrac{0.5}{\sqrt{2}} \times |110\rangle + \tfrac{0.5}{\sqrt{2}} \times |111\rangle - \tfrac{0.5}{\sqrt{2}} \times |111\rangle \\
\text{after consolidating:} & & = & \tfrac{1}{\sqrt{2}} \times |100\rangle - \tfrac{1}{\sqrt{2}} \times |101\rangle
\end{aligned}
$$

[We can implement each gate with a small piece of code]

```
for (WeightedKet wk : state) {
      // ...
      if (op1.gate.equals("X")) {
         wk.ket.flip(op1.arg);
      }
      // ...
      if (op2.gate.equals("CNOT")) {
         if (wk.ket.get(op2.arg1)) {
            wk.ket.flip(op2.arg2);
         }
      }
      // ...
}
```

Here we see how to implement that X flips a qubit, while $CNOT$ does a conditional flip.

[We can use a random-number generator to help implement measurement]
For measurement, generate a random number r between 0 and 1, and in the state $\Sigma_{x \in \{0,1\}^n} \; a_x |x\rangle$, sum up $|a_x|^2$ until the sum goes above r. For the a_x that pushed the sum above r, output x as the measurement. The simulator is 196 lines of code.

Transition to Close: So there you have it.

Review: We can implement a quantum simulator using classical principles for implementing interpreters. The grammar, the parser, and the program representation require little work, and the rest is straightforward implementation of well-understood operations.

Strong finish: Quantum simulators are essential for developing and debugging quantum programs. They need worst-case exponential time in the number of qubits and have so far scaled to 70 qubits.

Call to action: Apply all your CS knowledge to help build more scalable quantum simulators.