Introduction to Quantum Programming and Semantics

Lecture 1: Introduction

Chris Heunen

University of Edinburgh

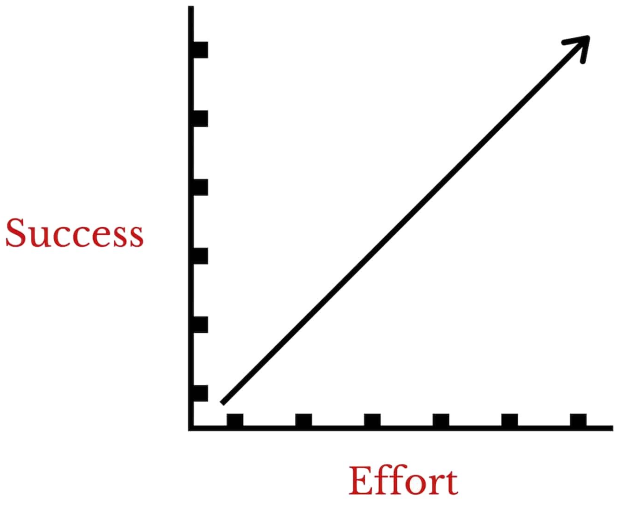
Practicalities

- Course team:
    - Chris Heunen
    - Malin Altenmuller
    - Louis Lemonnier
    - Kim Worrall
- Opencourse, Learn, Piazza
- Lectures:
    - Tuesday 2-3pm: Lister 4.1
    - Thursday 2-3pm: Appleton 2.12
    - No lectures: Thursday week 5
- Tutorials:
    - Wednesday 11am-12 week 3-8: High School Yards G.01
    - Wednesday 11am-12 week 9: Lister 3.1
- Labs:
    - Weeks 7 and 9

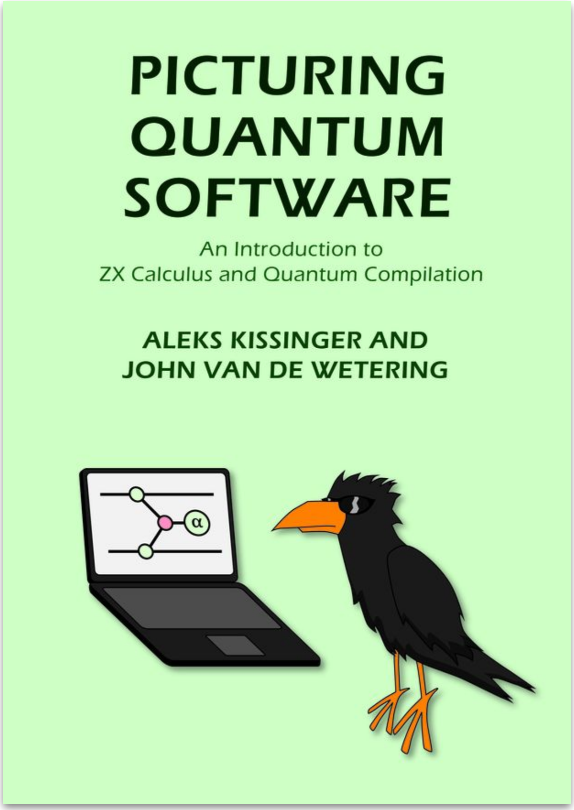


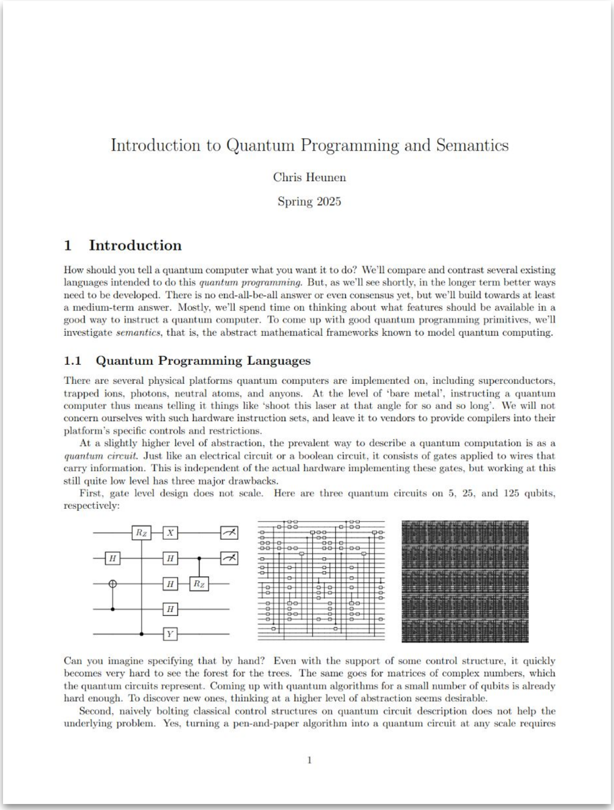

Assessment

- Labs (0%): practical
- Tutorials (0%): exercise sheets
- Coursework (30%): week 5
- Exam (70%): April-May diet



Course Material





Introduction to Quantum Programming and Semantics

Chris Heunen

Spring 2025

1 Introduction

How should you tell a quantum computer what you want it to do? We'll compare and contrast several existing languages intended to do this *quantum programming*. But, as we'll see shortly, in the longer term better ways need to be developed. There is no end-all-be-all answer or even consensus yet, but we'll build towards at least a medium-term answer. Mostly, we'll spend time on thinking about what features should be available in a good way to instruct a quantum computer. To come up with good quantum programming primitives, we'll investigate *semantics*, that is, the abstract mathematical frameworks known to model quantum computing.

1.1 Quantum Programming Languages

There are several physical platforms quantum computers are implemented on, including superconductors, trapped ions, photons, neutral atoms, and anyons. At the level of 'bare metal', instructing a quantum computer thus means telling it things like 'shoot this laser at that angle for so and so long'. We will not concern ourselves with such hardware instruction sets, and leave it to vendors to provide compilers into their platform's specific controls and restrictions.

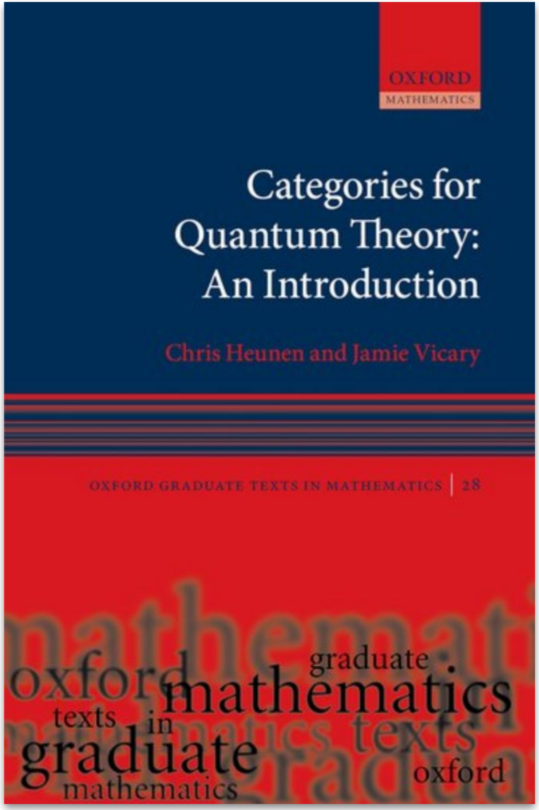
At a slightly higher level of abstraction, the prevalent way to describe a quantum computation is as a *quantum circuit*. Just like an electrical circuit or a boolean circuit, it consists of gates applied to wires that carry information. This is independent of the actual hardware implementing these gates, but working at this still quite low level has three major drawbacks.

First, gate level design does not scale. Here are three quantum circuits on 5, 25, and 125 qubits, respectively:

Can you imagine specifying that by hand? Even with the support of some control structure, it quickly becomes very hard to see the forest for the trees. The same goes for matrices of complex numbers, which the quantum circuits represent. Coming up with quantum algorithms for a small number of qubits is already hard enough. To discover new ones, thinking at a higher level of abstraction seems desirable.

Second, naively bolting classical control structures on quantum circuit description does not help the underlying problem. Yes, turning a pen-and-paper algorithm into a quantum circuit at any scale requires

1



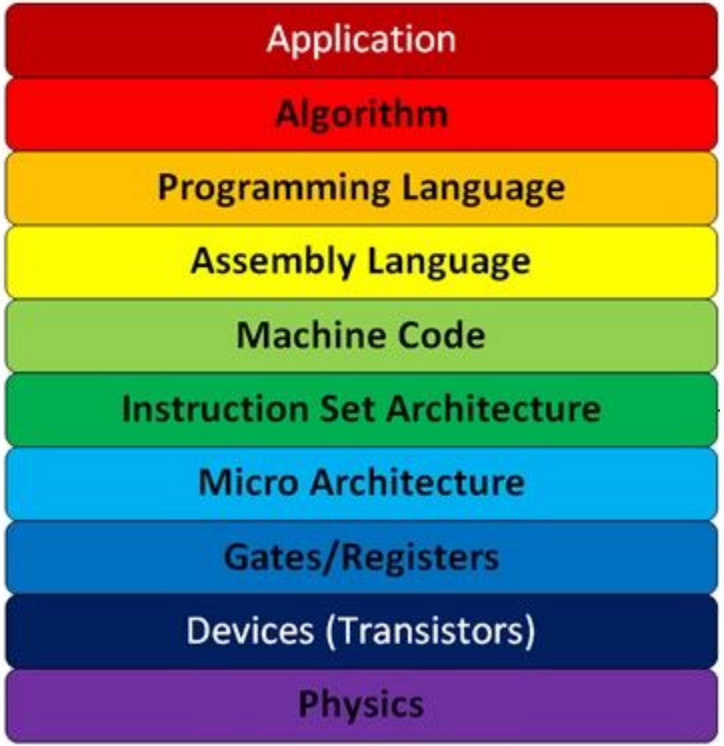
Updates

- "Too much time spent on theory"
    - No categories
- "Not enough programming"
    - More focus on practicals
- "Not enough practice with graphical reasoning"
    - Rebalanced, more introduction on diagrammatics
- "Unclear link between theory and coding"
    - Cutting edge developments

Quantum Programming Languages

Levels of abstraction

- … ?
- Circuit description languages
- Quantum circuits
- Quantum platforms: *superconducting, optical, trapped ions, neutral atoms*



Existing Quantum "Programming" Languages

OpenQASM

Q#

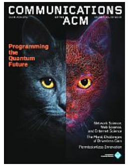
Cirq

Qiskit

Pennylane

Silq

Tket

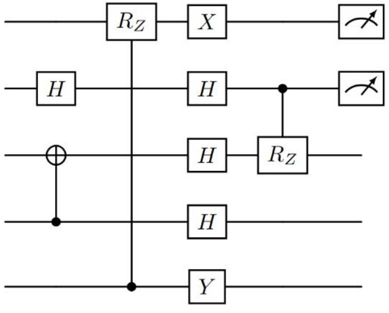


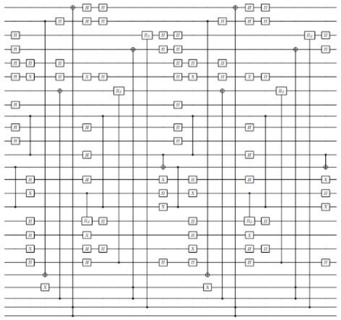
Quipper

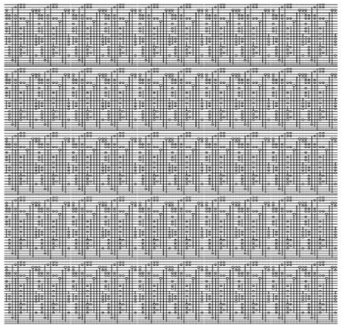
Pyquil

Problems of scale

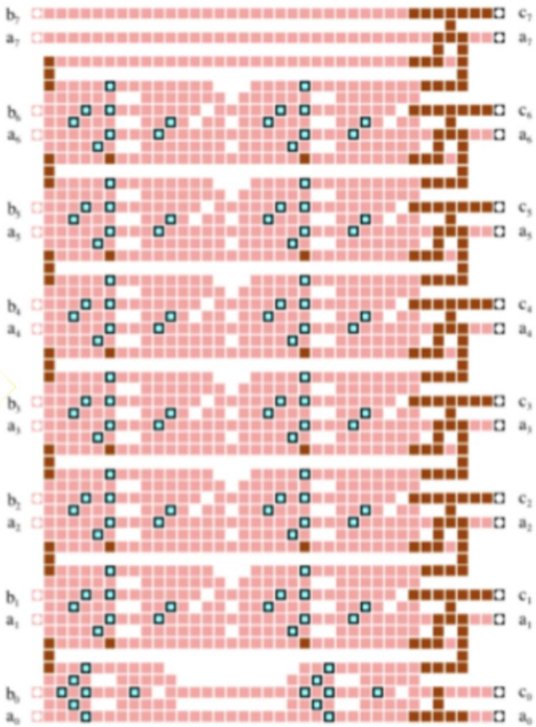
- Hard to reason
- Hard to optimise
- Algorithm discovery
- Control structures







Need for abstraction: circuits



8-bit adder: dimension 2^{1764}!

Need for abstraction: OpenQASM

```
// quantum ripple-carry adder from Cuccaro et al, quant-ph/0410184
OPENQASM 2.0;
include "qelib1.inc";
gate majority a,b,c
{
  cx c,b;
  cx c,a;
  ccx a,b,c;
}
gate unmaj a,b,c
{
  ccx a,b,c;
  cx c,a;
  cx a,b;
}
qreg cin[1];
qreg a[4];
qreg b[4];
qreg cout[1];
creg ans[5];
// set input states
x a[0]; // a = 0001
x b;    // b = 1111
// add a to b, storing result in b
majority cin[0],b[0],a[0];
majority a[0],b[1],a[1];
majority a[1],b[2],a[2];
majority a[2],b[3],a[3];
cx a[3],cout[0];
unmaj a[2],b[3],a[3];
unmaj a[1],b[2],a[2];
unmaj a[0],b[1],a[1];
unmaj cin[0],b[0],a[0];
measure b[0] -> ans[0];
measure b[1] -> ans[1];
measure b[2] -> ans[2];
measure b[3] -> ans[3];
measure cout[0] -> ans[4];
```

Need for abstraction: Q#

```
operation TestBellState(count : Int, initial : Result) : (Int, Int) {

    mutable numOnes = 0;
    using ((q0, q1) = (Qubit(), Qubit())) {
        for (test in 1..count) {
            Set (initial, q0);
            Set (Zero, q1);

            H(q0);
            CNOT(q0,q1);
            let res = M(q0);

            // Count the number of ones we saw:
            if (res == One) {
                set numOnes += 1;
            }
        }

        Set(Zero, q0);
        Set(Zero, q1);
    }

    // Return number of times we saw a |0> and number of times we saw a |1>
    return (count-numOnes, numOnes);
}
```

Need for abstraction: Qiskit

```
qc = QuantumCircuit(3, 2)
```

This will create a quantum circuit equivalent to the following (still valid) circuit declaration:

```
qr = QuantumRegister(3, name='q')
cr = ClassicalRegister(2, name='c')
qc = QuantumCircuit(qr, cr)
```

Registers are created automatically and can be accessed through the circuit as needed.

```
print(qc.qregs)
print(qc.cregs)
```

```
[QuantumRegister(3, 'q')]
[ClassicalRegister(2, 'c')]
```

Quantum/classical bit index-based addressing

In the spirit of register-less circuits, qubits and classical bits (clbits) can now be addressed directly by index, without a need for referencing a register. In the following example, `bell.h(0)` attaches a Hadamard gate to the first quantum bit.

```
bell = QuantumCircuit(2, 2)
bell.h(0)
bell.cx(0, 1)
bell.measure([0,1], [0,1])

bell.draw()
```

Need for abstraction: Quipper

```
qft’ :: [Qubit] -> Circ [Qubit]
qft’ [] = return []
qft’ [x] = do
  hadamard x
  return [x]
qft’ (x:xs) = do
  xs’ <- qft’ xs
  xs’’ <- rotations x xs’ (length xs’)
  x’ <- hadamard x
  return (x’:xs’’)
 where
   rotations :: Qubit -> [Qubit] -> Int -> Circ [Qubit]
   rotations _ [] _ = return []
   rotations c (q:qs) n = do
     qs’ <- rotations c qs n
     let m = ((n + 1) - length qs)
     q’ <- rGate m q ‘controlled‘ c
     return (q’:qs’)
```

Semantics

Meaning

Are these two programs the same?

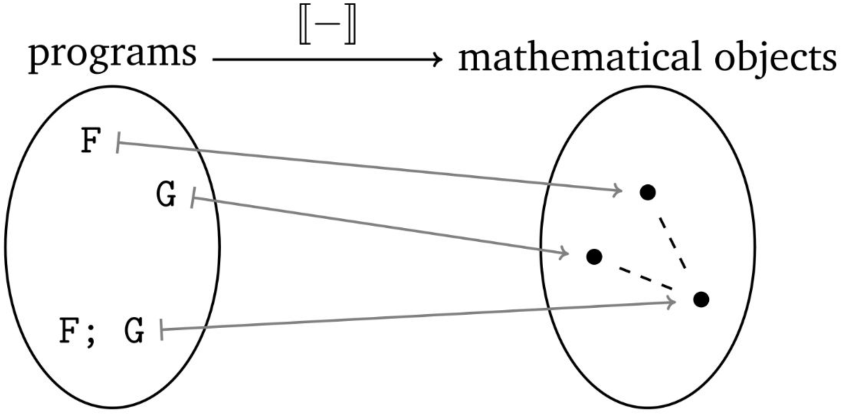
$$P = (\texttt{if 1 = 1 then F else G})$$
$$Q = (\texttt{if 1 = 0 then F else F})$$

- Different syntax
- Different operationally
- But denote same algorithm: $[\![P]\!] = [\![Q]\!] = [\![\texttt{F}]\!]$

Denotational semantics

- *Operational*: (efficiency)
  remember implementation details

- *Denotational*: (correctness)
  see what program does conceptually



Why?

- Ground programmer's unspoken intuitions
- Justify/refute/suggest program transformations
- Understand programming through models

Programming and semantics

- What if P, Q executables instead of source code? Black box.
  But can still analyse *information flow.*

- Empirical method: know how quantum theory works, but why?

- Cannot copy or delete, how to handle recursion?

- Investigate semantics to design good programming language

- “Semantics = programming language”

Diagrammatic reasoning

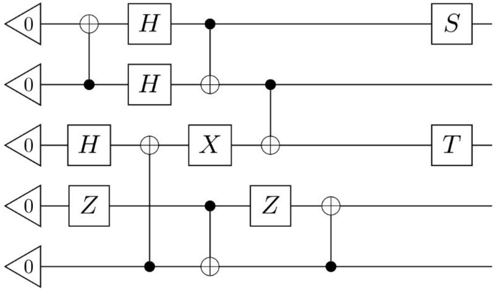
All about pictures

- can manipulate like flow chart
- mathematically rigorous backend
- complete for quantum computation
- higher level than quantum circuits
- built up from basic elements: Z and X observables

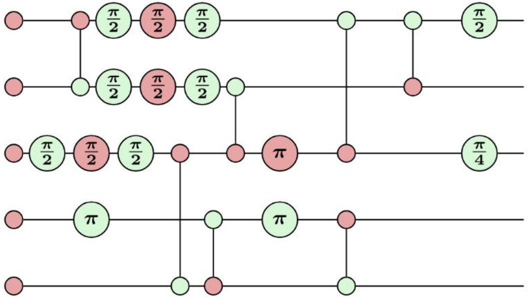
Workflow

```
INIT 5
CNOT 1 0
H 2
Z 3
H 0
H 1
CNOT 4 2
...
```

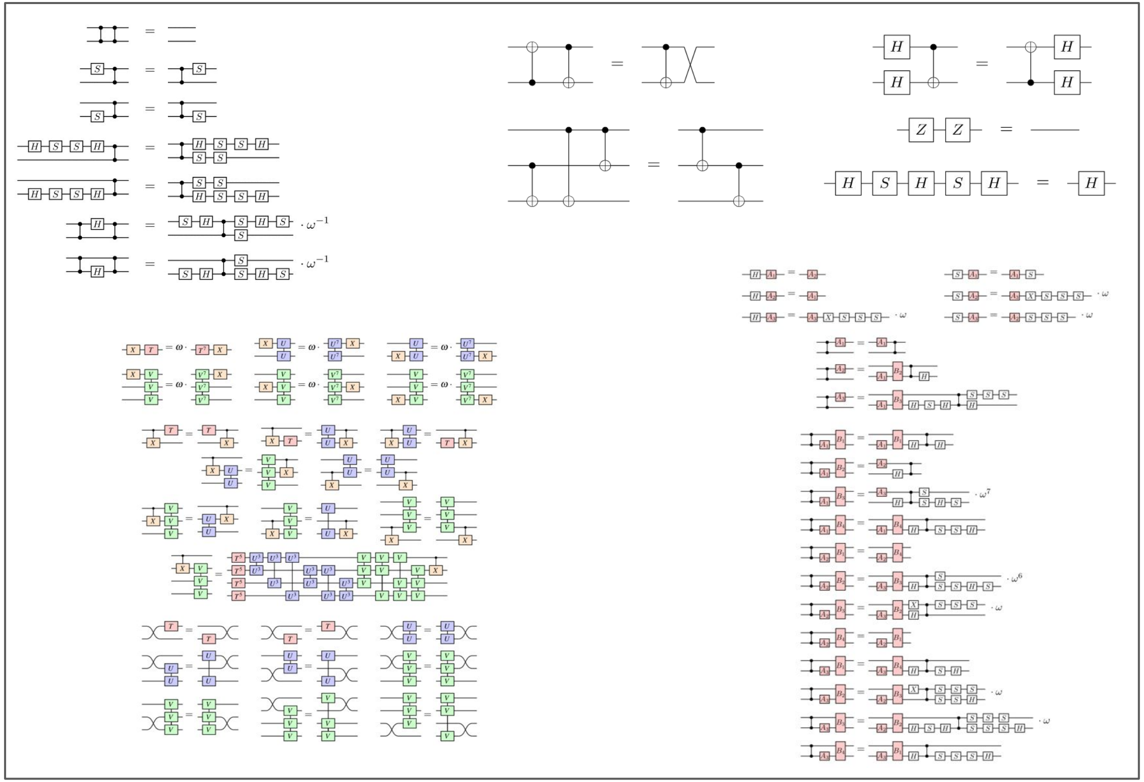
↔



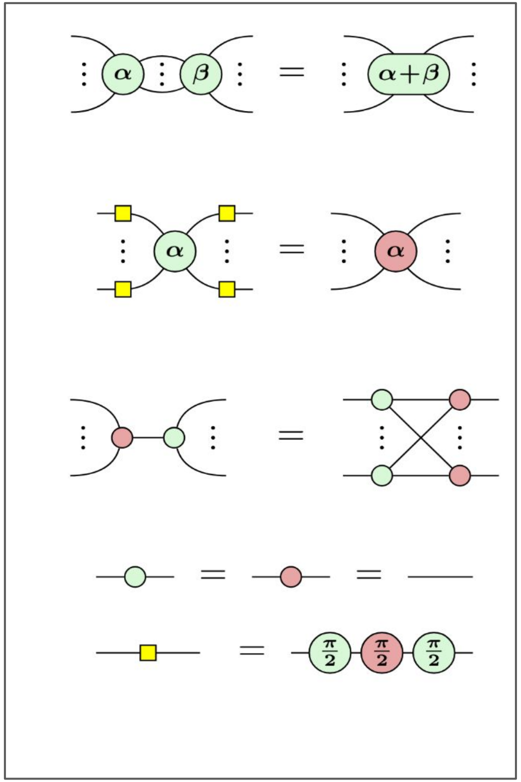
↔



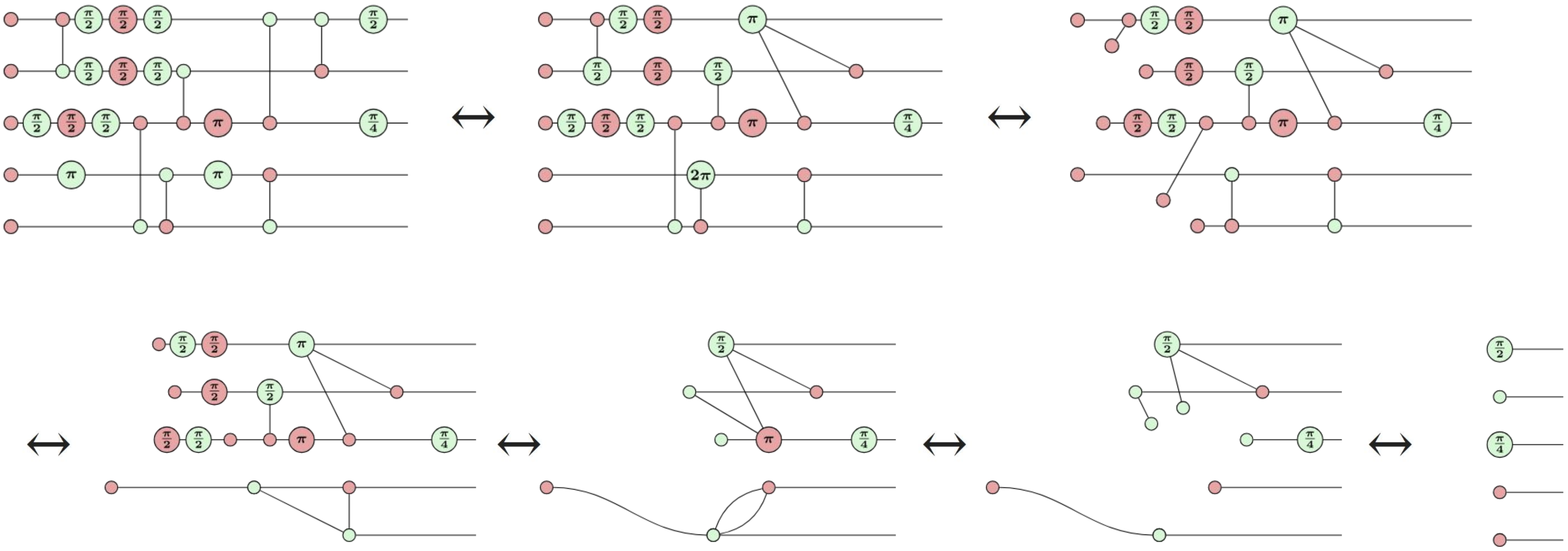
Rules of engagement



vs



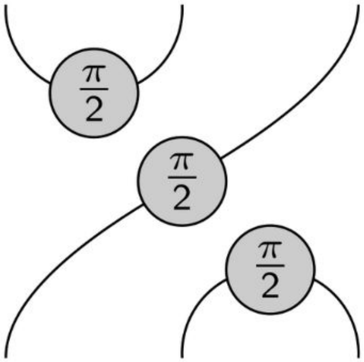
Graphical rewriting



ZX calculus

$$\frac{1}{4}\begin{pmatrix} -1+i & 1+i & 1+i & -1+i & 1+i & 1-i & 1-i & 1+i \\ 1+i & 1-i & 1-i & 1+i & -1+i & 1+i & 1+i & -1+i \\ 1+i & 1-i & 1-i & 1+i & 1-i & -1-i & -1-i & 1-i \\ 1-i & -1-i & -1-i & 1-i & 1+i & 1-i & 1-i & 1+i \\ 1+i & 1-i & 1-i & 1+i & 1-i & -1-i & -1-i & 1-i \\ 1-i & -1-i & -1-i & 1-i & 1+i & 1-i & 1-i & 1+i \\ -1+i & 1+i & 1+i & -1+i & 1+i & 1-i & 1-i & 1+i \\ 1+i & 1-i & 1-i & 1+i & -1+i & 1+i & 1+i & -1+i \end{pmatrix}$$

vs.



The plan:

- To find good **syntax** and **constructs** for quantum programming,
- Compare and contrast **existing languages**, and
- Investigate **semantics** *first*, specifically
- Founding **graphical programming language**