Introduction to Quantum Programming and Semantics

Chris Heunen

Spring 2025

1 Introduction

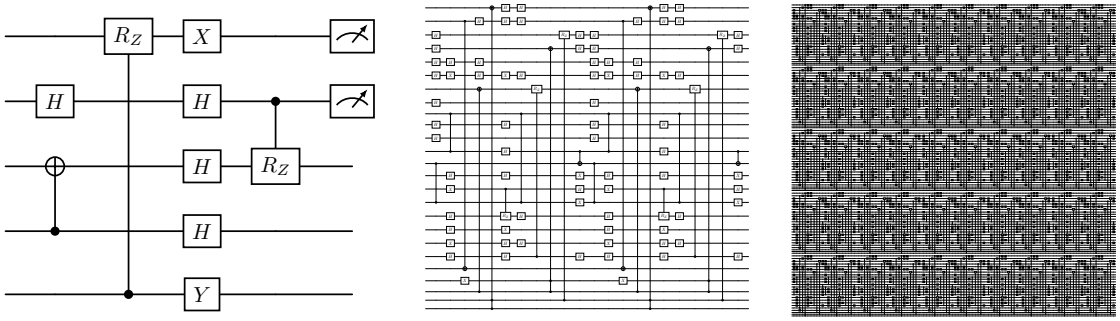
How should you tell a quantum computer what you want it to do? We'll compare and contrast several existing languages intended to do this *quantum programming*. But, as we'll see shortly, in the longer term better ways need to be developed. There is no end-all-be-all answer or even consensus yet, but 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

some sort of programming language. And the existing quantum programming languages we'll consider – OpenQASM, Qiskit, Quipper, Q#, and Silq – are all really circuit description languages. But instead of telling them what you want to compute and let the compiler figure it out, you have to tell them exactly how you want to compute something. For example, putting smaller subcircuits together into larger circuits quickly runs down the scarce resources available. To enable the programmer to deal with larger problems, patterns more abstract than quantum circuits seem desirable.

Third, optimising quantum circuits is a hard problem. How can you find a circuit that performs the same computation but is cheaper? Even with the help of a compiler it is hard if you restrict yourself to replacing quantum circuits by equivalent ones. To make progress you sometimes have to first make a quantum circuit bigger before you can make it smaller. To find good optimisations, considering abstractions beyond quantum circuits seems desirable.

What would be a good quantum programming language then? This is a brave new world, as many of the lessons of classical computer science don't apply. It is unclear what a quantum programming language should look like: if you cannot copy or delete information, you cannot push and pop things on the stack, how do you handle recursion, or even if-then-else statements? Therefore, investigating the semantics first is in a sense the only way to inform the design of a good *quantum programming language*. The quest is on to find the 'right' quantum programming primitives, that are efficient to translate into hardware instructions, that are powerful enough to support advanced algorithms elegantly, and that are intuitive enough to the human programmer to express their thoughts succinctly.

1.2 Semantics

Suppose `F` and `G` are two fragments of code, and consider the two computer programs:

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

Are P and Q the same programs or not?

Syntactically, they are clearly different: the code fragment P is different than the code fragment Q. But do P and Q compute the same? The answer depends on how fine-grained you are willing to look. From the perspective of machine *operations*, a (nonoptimising) compiler will make P and Q into different executables, because the machine will be storing `G` somewhere in memory when executing P, even though it is dead code that will never be reached, but not so when executing Q. But from perspective of the user, P and Q *behave* the same. They give the same output on every input. They *denote* the same algorithm, namely `F`.

This little analysis we have been doing is called *semantics*. We assigned to the two syntactical fragments P and Q their meaning, in the form of some mathematical objects $[\![P]\!]$ and $[\![Q]\!]$. On the first level, *operational semantics*, that mathematical object encoded all implementation details, so that we could retain the difference between the dead code. On the second level, *denotational semantics*, that mathematical object was a set-theoretical function that transforms input into output, and we didn't care how exactly the function performed that computation.

Denotational semantics is used to show that two programs implement 'the same' algorithm. For example, there are many ways to sort an array, but as long as we don't care about how fast the implementation is they all do the same. Quicksort might be faster than bubble sort, but they both sort. In other words, denotational semantics is used to prove that programs meet their specification, that they 'do what they should do'.

The example with P and Q above was insultingly simple, but you can imagine it gets a lot more complicated when for example recursion is in play. The important thing is that the assignment $P \mapsto [\![P]\!]$ *preserves structure*. For example, if we want to prove something about *concatenating* program fragments, we should have some operation that concatenates their denotations:

$$[\![\texttt{F; G}]\!] = [\![\texttt{G}]\!] \circ [\![\texttt{F}]\!]$$

Similarly, recursion requires us to talk about substitution (calling subprograms with some input) in the syntax, so the semantics had better have some notion of *function space*

$$[\![\texttt{F(X)}]\!] = [\![\texttt{F}]\!]\,([\![\texttt{X}]\!])$$

where $[\![\texttt{F}]\!]$ can live. For a third example, if the programming language describes concurrent computation, there should be some sort of *parallel composition* of denotations:

$$[\![\texttt{F par G}]\!] = [\![\texttt{F}]\!] \otimes [\![\texttt{G}]\!]$$

1.3 Diagrammatic reasoning

The trick is to choose the mathematical object in which the denotations $[\![\texttt{F}]\!]$ live cleverly. It should have the same structure as the programs you're analysing, but abstract away from all the unimportant details, so that powerful mathematical theorems can be used. Popular choices are λ-calculus or partially ordered sets. We will use a graphical language called *ZX-calculus*. ZX-diagrams may look like simplistic pictures, but they are perfectly rigorous. (Secretly, they're based on the powerful mathematical tool *category theory*, but we will only see this in passing.)

Aside from programme optimisations that are hard to automate or even spot if there are implementation details in the way, there is another reason to consider semantics abstractly. Above we analysed code in an actual programming language. If we consider *quantum* processes, there is no such neat description. The systems are just black boxes that we cannot look inside. But we can still see how the boxes behave; that is precisely the empirical method of physics! Analogously, in computer science, if I give you two executables instead of their source codes, can you still say whether they implement the same algorithm? No, because then you could solve the halting problem.

But you can still analyse the structure of the two computations using denotational semantics. You can see how the *information flows* from input to output and how it is recombined in the process. This kind of bookkeeping can be supported graphically in a beautiful way. Instead of boring algebra (like λ-calculus or partially ordered sets), string diagrams like ZX-diagrams can be manipulated with graph rewriting, in a way that can even be automated. In this way ZX-calculus itself becomes a quantum programming, where we can combine subroutines in various ways to compose larger programs. It delivers optimisations that are currently state of the art, so must capture some of the essence of quantum computing. Yet at the same time ZX-diagrams are finite combinatorial objects, unlike matrices of complex numbers, so can be manipulated easily by computers.

1.4 Sources

These notes will be your primary guide through this development. But as there is no point in reinventing the wheel, you will often be directed to read bits and pieces of the following two texts:

[KW]: A. Kissinger and J. van de Wetering, "*Picturing Quantum Software*"
Cambridge University Press, 2025
https://github.com/zxcalc/book

[HV]: C. Heunen and J. Vicary, "*Categories for Quantum Theory*"
Oxford University Press, 2019

2 Quantum circuits

Let's make sure we're all on the same page about the traditional model of what we mean when we say quantum computing, before we get into fancier models.

2.1 Complex numbers

Quantum computing is about amplitudes, which are complex numbers. (The absolute square of these amplitudes are real numbers, giving quantum computing its probabilistic character.) Read [KW 2.1.1, 2.1.2, 2.2.1, and 2.2.2] to refresh your knowledge of complex numbers, how qubits are modelled by unit vectors in $\mathbb{C}^2$, and the Bloch sphere.

2.2 Matrices

In the traditional model, qubits evolve according to unitary matrices. These are reversible operations. We will talk later about the irreversible operation of measurement, which is not modeled by a unitary matrix. Read [KW 2.1.3] to refresh your knowledge of linear algebra.

Instead of unitaries, physicists often talk about Hamiltonians. These are self-adjoint matrices, whose eigenvalues represent the possible energy levels of the system they represent. Don't be scared: talking about Hamiltonians is exactly the same as talking about unitaries, see [KW 2.2.3].

2.3 Quantum circuits

Quantum circuits are the de facto standard model of quantum computing. They are a succinct way to write down unitary matrices. Read [KW 2.3] to learn their basics.

To connect back to Lecture 1, let us remark that we can think about quantum circuits as *syntax*, or rudimentary quantum programs, and unitary matrices as *semantics*, or the meaning of quantum programs. The semantics brackets $[\![-]\!]$ in this case *interpret* a circuit as a matrix. This means that first of all you need to know how to interpret each of the gates in a quantum circuit. For example:

$$\left[\!\!\left[\text{---}\boxed{H}\text{---} \right]\!\!\right] = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

Next, notice that matrices have enough structure to interpret concatenation of two quantum circuits inductively, namely by matrix multiplication:

$$\left[\!\!\left[\equiv\boxed{C_1}\equiv\boxed{C_2}\equiv \right]\!\!\right] = \left[\!\!\left[\equiv\boxed{C_2}\equiv \right]\!\!\right] \circ \left[\!\!\left[\equiv\boxed{C_1}\equiv \right]\!\!\right]$$

(Recall that in a matrix multiplication BA, the matrix A gets applied first, and then B.) Finally, matrices also have enough structure to interpret parallel composition of two quantum circuits, namely by tensor product:

$$\left[\!\!\left[\begin{array}{l} \equiv\boxed{C_1}\equiv \\ \equiv\boxed{C_2}\equiv \end{array} \right]\!\!\right] = \left[\!\!\left[\equiv\boxed{C_1}\equiv \right]\!\!\right] \otimes \left[\!\!\left[\equiv\boxed{C_2}\equiv \right]\!\!\right]$$

But we'll discuss the details of tensor products next time. Thus you can interpret any quantum circuit as a unitary matrix: slice it apart into horizontal and vertical layers, interpret each of the gates, and then put the layers back together using matrix multiplication and tensor product.

3 Composition and tensor products

In this lecture we'll analyse our first quantum programming language, OpenQASM. It really isn't much more a computer-readable (but less human-readable) syntax for the graphical quantum circuit notation we saw last time, but it will serve as a good example to illustrate the concepts of semantics and composition we've already seen, and to introduce tensor products, and the Deutsch-Jozsa algorithm, which will be our running example throughout the course.

3.1 OpenQASM

OpenQASM is a circuit description language developed with near-term applications in mind, i.e., to program the (small and noisy) quantum computers we have today. Its main goal is to serve as an intermediate representation for higher-level compilers in order to interact with quantum hardware. For example, it can be used to interact with IBM's quantum computers or Amazon's Braket, although as we shall see later, higher-level languages are also available for this task that use OpenQASM as a compilation target.

That it is a circuit description language means that OpenQASM programs are not meant to be run directly on a quantum computer. Instead, an OpenQASM program is run on a classical computer to produce a quantum circuit, which can then be handed off to a quantum computer for execution. Why go through all of this trouble if, in the end, all we're doing is describing quantum circuits?

- *Abstraction*: In a circuit, a single line denotes a single qubit, and its precise location matters to the operation of the circuit. In OpenQASM, variables are used to represent (collections of) qubits, and can be given descriptive names to better relay their intended use, and declared in any order. The precise mapping of variables to qubits (known as *routing*) in a circuit is handled by the OpenQASM compiler. Compile-time constants can also be used to describe algorithms that are flexible regarding the number of qubits available, such as those for qubit arithmetic.
- *Typing*: Variables (and subroutines, see below) must be declared along with their types, such as `qubit` (a qubit), `bit` (a classical bit), `qubit[4]` (an array of four qubits), `int[16]` (a classical 16-bit integer), and so on. This can not only be used to catch errors in programming, but can also be used by the OpenQASM compiler to improve the quality of the quantum circuit it produces.
- *Structure*: Larger programs are typically written by in a structured fashion by combining smaller parts using things like conditionals, loops, and subroutines. Programming in this style greatly improves the ability to read, understand, and debug a program, even as it grows quite large. You will notice that quantum (or classical) circuits do not support either of these features (we say that they are *unstructured*), but OpenQASM does, with the OpenQASM compiler able to compile these down to quantum circuits.

Full documentation can be found at openqasm.com; here we give a brief overview of the core of the language. It is imperative and statically typed. Statements are separated by semi-colons and whitespace is ignored.

Data types and variable declaration Unsurprisingly, OpenQASM has types for declaring bits and qubits:

```
bit b;
qubit q;
```

Qubits stand apart from other data types a key way: they cannot be initialised in the same statement as their declaration. In fact, the only built-in way to initialise a qubit is with the special statement 'reset', which sets the qubit to the 0 computational basis state:

```
qubit q;
reset q;
```

Unitary quantum gates There is only a single built-in quantum gate, called `U`. It parametrises a single-qubit unitary as a rotation of the Bloch sphere of the form

$$U(a,b,c) = \begin{pmatrix} \cos(a/2) & -e^{ic}\sin(a/2) \\ e^{ib}\sin(a/2) & e^{i(b+c)}\cos(a/2) \end{pmatrix}.$$

For example,

```
qubit q;
reset q;
U(pi,0,pi) q;
```

enacts an X gate on the qubit `q`. Since `q` was reset to the 0 basis state, `q` is then in the 1 basis state at the end of the program.

As the same gate can be applied many times, it would be cumbersome to use this syntax for each identical call. Luckily, there is a way to abstract gate names: a block called `gate`. For example:

```
gate X a {
  U(pi, 0, pi) a;
}
```

abstracts the X gate, while a Hadamard gate is built as:

```
gate H a {
  U(pi/2, 0, pi) a;
}
```

3.2 Composition

The very basic feature of an OpenQASM program is that it is a sequence of lines. Each line is an instruction, and subsequent lines are executed one after the other. Hence, to interpret it semantically in anyway, we need some form of sequential composition. If OpenQASM is the syntax, the semantics could be quantum circuits, which can be composed sequentially. Or the semantics could be unitary matrices, which can also be multiplied. The important point is that any domain that OpenQASM semantics are to be interpreted in must have some form of sequential composition.

3.3 Tensor products

Quantum circuits can also be composed in parallel. Similarly, OpenQASM programs can declare registers of bits and registers of qubits:

```
bit[5] b_reg;
qubit[5] q_reg;
```

Registers have a fixed size and cannot be resized after declaration.

How are we to interpret these registers semantically? By *tensor products* of matrices. Read [KW 2.1.4] to freshen up on the mathematics of tensor products.

Once we have registers of multiple qubits, it makes sense to talk about controlled gates. They are constructed by adding a `ctrl` modifier to a pre-existing gate:

```
ctrl @ U(a,b,c) q_reg[0], q_reg[1];
```

Formally speaking, this implements the gate $1 \oplus U(a,b,c)$. This can then be abstracted using the `gate` block:

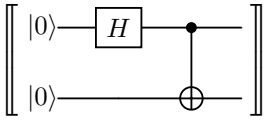
```
gate CX a, b {
   ctrl @ U(pi, 0, pi) a, b;
}
CX q_reg[0], q_reg[1];
```

constructs and runs a CX gate.

Putting it all together, here is a simple OpenQASM program generating a Bell state:

```
gate H a {
   U(pi/2, 0, pi) a;
}
gate CX a, b {
   ctrl @ U(pi, 0, pi) a, b;
}
qubit[2] = q_reg;
reset q_reg;
H q_reg[0];
CX q_reg[0], q_reg[1];
```

Indeed, it compiles into the following quantum circuit, whose matrix semantics are:



$$= \frac{1}{\sqrt{2}}\left(|00\rangle + |11\rangle \right)$$

3.4 Deutsch-Jozsa

The Deutsch-Jozsa algorithm solves a certain problem faster in the quantum case than is possible in the classical case. It is a typical example of a quantum algorithm that decides on a solution without relying on approximation. We will be using it as a running example to illustrate quantum programming languages with. The Deutsch-Jozsa algorithm solves a slightly artificial problem, but other algorithms in this family include Shor's factoring algorithm, Grover's search algorithm, and the more general hidden subgroup problem. These algorithms have an 'all or nothing' nature, meaning we can see the structural difference between no information flow and maximum information flow.

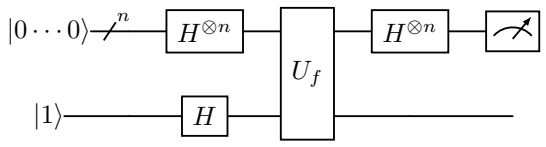
The problem is the following. Suppose you have a 2-valued function $\{0,1\}^n \xrightarrow{f} \{0,1\}$ on the of bitstrings of length n. If the function f takes just a single value on every bitstring, it is called *constant*. Another possibility is that the function takes the value 0 on exactly half the bitstrings, and takes the value 1 on the other half; in this case it is called *balanced*. Most functions are neither, but you are promised that your function is either balanced or constant. Your quest is to determine which of the two is the case.

The best classical strategy is rather simple. You have no knowledge of the structure of the function f in general, so you can only sample the function on some bitstrings. If you find two bitstrings which have different values, then f cannot be constant, so you conclude that f is balanced and are done. However, in the worst case, if f is balanced you may have to sample $2^{n-1}+1$ bitstrings until you hit two bitstrings with different values. If you sample this many bitstrings and find that f returns the same value for each one, then you can conclude that f is constant.

In both implementations and semantics, we have to be more precise about how to access the function f. A quantum computation only allows unitary operations; so we have to linearise the function $\{0,1\}^n \xrightarrow{f} \{0,1\}$ to a unitary matrix U_f, called an *oracle*. It is defined by:

$$U_f \, |xy\rangle = |x\rangle \otimes |y \text{ XOR } f(x)\rangle \, .$$

The Deutsch-Jozsa algorithm is now given by the following quantum circuit:



We will see later why this is correct, but for now: if f is constant, the probability of measuring $|0\cdots0\rangle$ on the first n qubits is 1, and if f is balanced, that probability is 0. Thus the quantum circuit indeed answers our question.

Let's implement Deutsch-Jozsa in OpenQASM. To keep it small, let's set $n = 2$, and let's choose f to be the XOR function, which is balanced. To implement the oracle, we have to turn it into a 2^3-by-2^3 unitary matrix with the following truth table on the computional basis states:

xyz	x XOR y	z XOR (x XOR y)
000	0	0
001	0	1
010	1	1
011	1	0
100	1	1
101	1	0
110	0	0
111	0	1

We can implement this unitary with two CX gates:

```
gate Oracle x y z {}
  CX x z;
  CX y z;
}
```

With the oracle in hand, we can now implement the rest of the Deutsch-Jozsa algorithm as follows:

```
// declare three qubits
qreg x;
qreg y;
qreg z;
// set the three qubits to |0>, |0>, and |1>
reset x;
reset y;
reset z;
X z;
// apply Hadamard to all three qubits
H x;
H y;
H z;
// apply the oracle
Oracle x y z;
// apply Hadamard to the first two qubits
H x;
H y;
// measure the first two qubits
bit a = measure x;
bit b = measure y;
```

We'll talk about measurement later, but executing this shows that the output bits are $a = 0$ and $b = 0$, showing that f was indeed balanced.

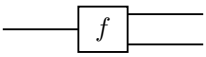
4 Graphical calculus

Just like quantum circuits can be regarded as pictures of OpenQASM programs, we will develop a graphical calculus for more general quantum programs. We will mostly use these diagrams as pieces of syntax, as shorthands for their semantic interpretation as matrices. (We will not use it in this course, but it is good to know that these graphical diagrams are much more powerful, and can be interpreted in any rich enough semantic structure, most generally a *monoidal category*, in a rigorous way that is sound and complete.)

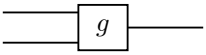
4.1 String diagrams

Forget for a minute that the gates in quantum circuits represent unitary matrices. Eventually, quantum circuits are a particular king of flow diagrams. The wires carry some kind of information, that can possibly be altered by passing through gates. This idea is much more general. For example, the gates do not have to be invertible in the way that unitary matrices are.

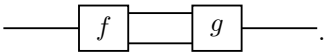
A *string diagram* also consists of (a finite number of) wires that are combined by (a finite number of) gates. Gates can have any number of input wires and any number of output wires. We are going to use string diagrams as semantics for quantum programs. Because quantum programs can be composed in sequence, we need to be able to compose string diagrams sequentially. This is done by simply wiring up the outputs of the first with the input of the second. For example, the sequential composition of a 'gate' with two outputs



with a 'gate' with two inputs

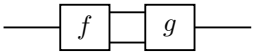


is the string diagram

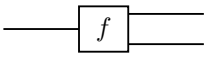
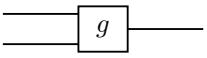


.

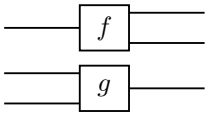
If our syntax has some notion of sequential composition written $\circ$, then the semantic relation is

$[\![g \circ f]\!]$ =  = $[\![g]\!] \circ [\![f]\!]$.

Similarly, quantum programs can be composed in parallel. Thus we also need to be able to compose string diagrams in parallel. This is done by simply stacking the string diagrams on top of each other. If two 'gates' f and g have semantics

$[\![f]\!]$ =  $[\![g]\!]$ = 

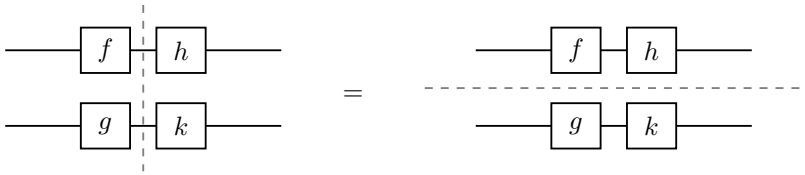
and the syntax has a notion of parallel composition written $\otimes$, then

$[\![f \otimes g]\!]$ =  = $[\![f]\!] \otimes [\![g]\!]$.

Some relationships between string diagrams are graphically natural, so it makes most sense to use string diagrams when these relationships also hold in our syntax and intended semantic models. For example, the following *interchange law* holds for quantum circuits, complex matrices, and string diagrams:

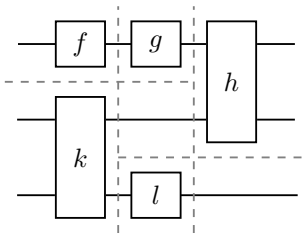
$$(h \otimes k) \circ (f \otimes g) = (h \circ f) \otimes (k \circ g),$$

provided the types (number of input and output wires) of the 'gates' match. In string diagrams:



The gray dashed lines here are not part of the diagram, and just indicate the order in which we drew the 'gates' on the paper, which is inconsequential.

This also indicates how we can read a string diagram: just like a quantum circuit, it can be decomposed into basic 'gates' using gray dashed lines, which are then recombined using sequential composition $\circ$ and parallel composition $\otimes$. For example,



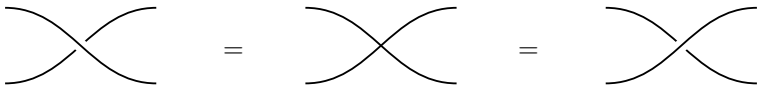
decomposes along the gray dashed lines, and then algebraically recombines into

$$(h \otimes \mathrm{id}) \circ (g \otimes \mathrm{id} \otimes l) \circ (f \otimes k).$$

Here, we have used a special 'gate' called id. It is like the identity matrix: it does nothing. More precisely, $\mathrm{id} \circ f = f = f \circ \mathrm{id}$ for any 'gate' f. In terms of string diagrams, we therefore draw it simply as a wire.

That leads to another property of string diagrams: it doesn't matter how long of a piece of wire we use for the identity 'gates'. Similarly, it doesn't matter whether wires are straight or wiggle around a bit. The following two string diagrams are the same:

Only the connectivity of the 'gates' matters. (Well, for now, wires aren't allowed to go 'back in time'; we'll consider that next time.) More precisely, two string diagram are considered to be the same when there is a 4-dimensional *isotopy* between them. That means that you can smoothly morph one into the other, without cutting any wires, and while keeping the connectivity the same. Imagine the diagrams being trapped between pieces of glass. The input and output wires are nailed fixed, but inside you can move things around as you please. Wires can cross each other, though, so we're talking about 4-dimensional glass hypercubes:



The rules of string diagrams are *sound*, meaning the following. Think of string diagrams as syntax, taking semantics in matrices. Suppose you have chosen matrices to interpret some basic string diagram 'gates'. If two string diagrams composed of those basic 'gates' are equal, then also the composite matrices are equal.

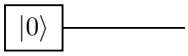
Later, we will extend the string diagram calculus with other elements, that also make it *complete*, meaning that if the composite matrices are equal, then there is a string diagrammatic proof that the string diagrams are equal.

Read [HV 0.1.5 and 1.1.1, and perhaps 1.2] for more information. Warning: string diagrams are often read bottom-to-top, or top-to-bottom, instead of left-to-right, as we have done here. Another warning: we have described 'gates' quite loosely, and [HV] makes them *much* more precise, using the notion of a *category*; we will not need that amount of precision in this course, so don't worry if you don't get it all.

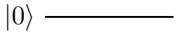
4.2 States, effects, and scalars

Unlike quantum gates, string diagram 'gates' do not have to have the same number of inputs as outputs. What happens if the number of input or output wires is zero?

A string diagram with no input wires is called a *state*. For example, a state called $|0\rangle$ would be drawn as a string diagram as

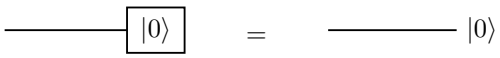


or even



Starting from the empty system, after 'executing' this string diagram, we have created a single system in some specified state. Let us emphasise that $|0\rangle$ is just a piece of syntax here, a bunch of symbols. It doesn't mean anything yet, and could equally just be called f or g. But when this syntax is interpreted semantically, we intend it to be the computational basis state $|0\rangle = \left(\begin{smallmatrix}1\\0\end{smallmatrix}\right) \in \mathbb{C}^2$.

A string diagram with no output is called an *effect*. For example, the string diagram syntax



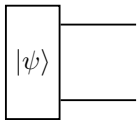
is interpreted semantically as the row vector $\langle 0| = (1\ 0)$. You can think about it like this: it represents the fact that a measurement in the computational basis happened, and the outcome was 0. This is also called *postselection*. (In one of the last lectures, about the quantum programming language Silq, we will see that this can be used constructively: if you can do a quantum *uncomputation* to make sure that the qubit is left in a specified state, then the qubit does not have to be measured destructively, possibly influencing the rest of the computation, and can instead be reused.)

Read [HV 1.1.2 and 1.1.4] for more information.

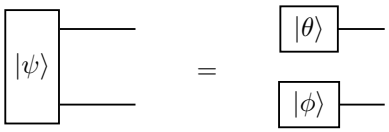
A string diagram with no input and no output at all is called a *scalar*. It is drawn simply as a box without any attached wires. According to our rule that 'only connectivity matters', these scalars can just float through the diagram at will. It doesn't matter where you draw them, the result will always be the same as a string diagram. You may as well put all scalars at the front of the diagram, but sometimes it's handier to keep them various places in the middle. It also follows that it doesn't matter in which order scalars are 'executed': scalars are commutative. Semantically, scalars will be interpreted as 1-by-1 matrices, that is, as complex numbers. (Read [HV 2.1] for more information if you want more details, but be warned that this is mostly about categories we will not need.)

4.3 Entanglement

States have zero inputs, but they can have more than one output. For example, we can easily draw *bipartite states*:



At the level of generality of string diagrams, we can already speak about entanglement, as follows. Some bipartite states will be of the form



These bipartite states are called *separable*. All other bipartite states are called *entangled*. Check that this matches your intuition for these words when you interpret the string diagram syntax semantically as matrices. (See [HV 1.1.3] for more (categorical) details.)

5 Bending space and time

Some bipartite states are very special: they are not just entangled, but maximally entangled. If we want to have any sort of graphical syntax that is to be interpreted semantically as such Bell states, we will need special string diagrams.

5.1 Cups and caps

A *Bell state* is the entangled bipartite state $(|00\rangle + |11\rangle)/\sqrt{2}$ in $\mathbb{C}^2 \otimes \mathbb{C}^2$. It cannot be written in the form $|\psi\rangle \otimes |\phi\rangle$. If we identity the space of two qubits with $\mathbb{C}^4$, the Bell state is

$$|\text{Bell}\rangle = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} \in \mathbb{C}^4.$$

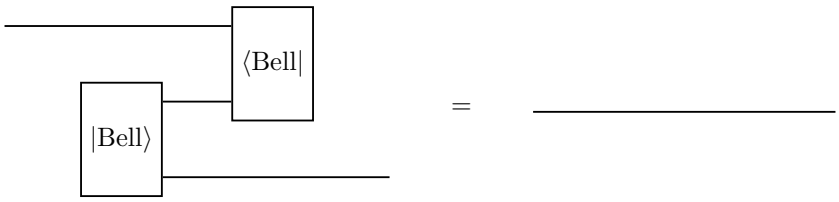
Its transpose is the row vector

$$\langle \text{Bell}| = \begin{pmatrix} 1 & 0 & 0 & 1 \end{pmatrix} \in \mathbb{C}^4.$$

Together, they satisfy a very special equation:

$$(\langle \text{Bell}| \otimes \text{id}) \circ (\text{id} \otimes |\text{Bell}\rangle) = \text{id}.$$

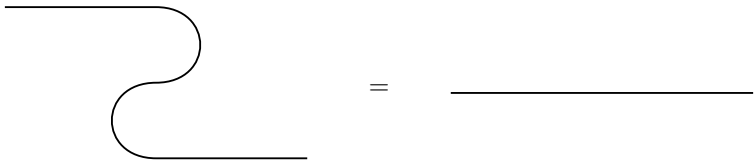
(Multiply out the matrices to see this!) It is called the *snake equation*, because it looks like this when we draw it as a string diagram:



We will adopt a purely graphical counterpart that acts like a Bell state, called a *cup*, drawn as

and a purely graphical counterpart that acts like the transpose of a Bell state, called a *cap*, drawn a

We postulate that the cup and cap must satisfy the snake equation:



(∗)

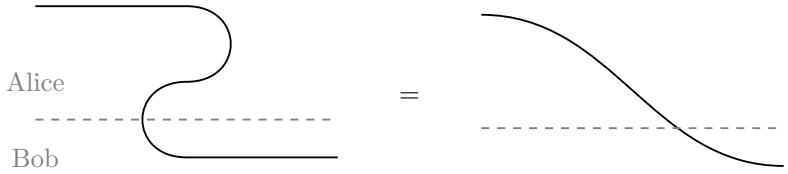
and furthermore:

String diagrams with cups and caps are no longer circuit diagrams, because time no longer unambiguously flows from left to right. In-in and out-out connections and loops are now allowed. But even with cups and caps, only connectivity matters in string diagrams. For example:

Now read [KW 2.1.7].

5.2 Teleportation

Cups and caps 'bend time into space'. This can be interpreted in a quantum world in a natural way. Look again at the snake equation (∗), but this time think about the spatial distribution of all the wires:



Reading the left-hand side diagram as a history of processes that happened: Alice receives an unknown qubit; Alice and Bob share a maximally entangled state; Alice measures the input qubit and her half of the shared pair and we postselect on a certain outcome. The right-hand side then says that after this procedure, Alice's unknown input qubit ends up as Bob's output qubit. In other words, this implements quantum teleportation. (We will deal with the postselection later in the course.)

Thus we can think of the left-hand side of the snake equation as graphical syntax for a program implementation, the right-hand side as a specification, and the equation as showing that the program satisfies the specification. Interpreting this teleportation program semantically as complex matrices (do it!) results in the usual quantum teleportation protocol. We won't do so in this course, but it can also be interpreted in other semantic domains: interpreting wires as binary relations between sets results in the usual one-time pad crypto protocol.

Now read [HV 3.2].

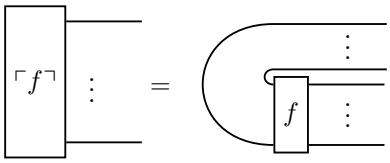
Incidentally, anticipating measurement, the teleportation protocol/circuit looks like this in OpenQASM:

```
gate H a {
   U(pi/2, 0, pi) a;
}

gate X a {
   U(pi, 0, pi) a;
}

gate Z a {
   U(0, 0, pi) a;
}

gate CX a, b {
   ctrl @ U(pi, 0, pi) a, b;
}
```

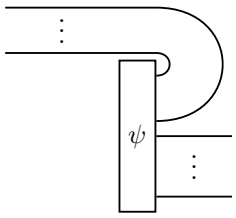
```
qubit input_state;
reset input_state;

// Create a Bell state
qubit[2] bell_state;
reset bell_state;
H bell_state[0];
CX bell_state[0], bell_state[1];

// Entangle the input state with Bell state
CX input_state, bell_state[0];

// Measure and correct
bit m1 = measure input_state;
bit m2 = measure bell_state[0];
if (m2 == true) {
    X bell_state[1];
}
if (m1 == true) {
   Z bell_state[1];
}
```

5.3 Taking names

We can make more precise that cups and caps 'trade space for time' and vice versa. Any string diagram f with m input wires and n output wires can be turned into a state $\ulcorner f \urcorner$ on $m+n$ wires:



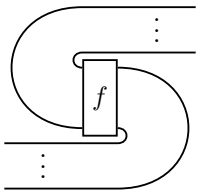
The state $\ulcorner f \urcorner$ is sometimes called the *name* of f. Vice versa, any state ψ on $m+n$ wires can be turned into a string diagram with m input wires and n output wires:



It is easy to see (do it!) that turning a string diagram into its name and then back again ends up with the original diagram, and that turning a state into a string diagram and then back again ends up with the original state. In other words, there is a one-to-one correspondence between states on $m+n$ wires and 'gates' with m input wires and n output wires. This is sometimes called the *Choi-Jamiolkowski isomorphism* or *map-state duality*.

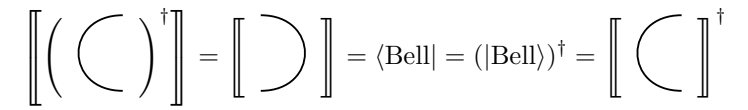
5.4 Transpose and adjoint

If f is a string diagram with m inputs and n outputs, using caps and cups we can construct a string diagram with n inputs and m outputs, called its *transpose*:



Basically the whole diagram is rotated 180 degrees. We will write f^* for this transpose diagram. Transposition is an operation that makes sense on the graphical syntax of our programs. It correspond semantically to taking the transpose of a matrix. Taking the transpose twice does nothing, both for string diagrams and for matrices. (Don't just believe me, check this!) If you're brave, read [HV 3.1.2] for some more properties of transposes.

There is another way to reverse string diagrams. If f is a string diagram with m inputs and n outputs, we can draw it reflected in a vertical mirror to get a string diagram $f^\dagger$ with n inputs and m outputs. This is called the *adjoint* of a string diagram. Of course, taking the adjoint twice does nothing: $f^{\dagger\dagger} = f$. Having the structure of adjoints on our graphical syntax means that every program can be *reversed* in some sense. This can be interpreted semantically in unitary matrices by sending the adjoint of a string diagram to the adjoint of its interpretation. For example:



Because you cannot tell the difference between a box, the 180-degree rotation of the box, and the vertical reflection of the box, it makes sense to not use rectangles for 'gates' in our graphical programming language, but to break the symmetry by adding a dot in a corner, or adding a wedge to one of the edges, so you can tell 'which way is up'. [HV] does this, but the graphical language we're working towards is in fact invariant under rotational and reflection symmetry, so we won't bother with this there.

6 Tensor networks

The graphical programming language we're building up is indeed purely graphical. But it will be handy to import some aspects of linear algebra for two reasons. First, eventually we will be able to replace postselected measurements by proper probabilistic measurements graphically, but that's rather advanced, and in the mean time it's easier to keep the probabilities explicit. Second, allowing sums of diagrams makes it easier to connect to known structures like orthonormal bases and tensor networks.

6.1 Trace and dimension

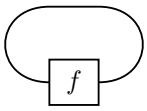
First, two more properties of cups and caps, namely trace and dimensions. You may expect this to need something like sums or bases, but it works purely graphically.

What happens when you compose a cup and a cap straight up?

Because there are no input wires nor output wires, you clearly get a scalar. Which one? Its semantic interpretation of a matrix will tell you: it's the complex number 2. In other words, it's the *dimension* of the qubit.

Moreover, this graphical notion of dimension behaves like it should: the dimension of n wires is the string diagram consisting of n circles, so correspond to the scalar 2^n. That is, the dimension of a tensor product if the product of the dimensions. In particular, the dimension of 0 wires is the scalar 1.

What happens when you compose some string diagram f with a cup and a cap?



Interpreted semantically, you get the *trace* of the matrix $[\![f]\!]$. (Work it out!) It's drawn here for a one-qubit 'gate' f, but it works equally for n-qubit string diagrams. That makes sense, because the dimension of $(\mathbb{C}^2)^{\otimes n}$ is the trace of the identity matrix on n qubits. Some properties of traces, that are unpleasant to prove with matrices, are now immediately obvious from the graphical rule that only connectivity matters. For example, as we saw last time, the trace of $g \circ f$ equals the trace of $f \circ g$.

If you have a string diagram with $m+o$ input wires and $n+o$ output wires, you can also build a 'feedback loop' on the last o wires to get a diagram with m inputs and n outputs. The trace is the special case where $m = n = 0$. The general case corresponds to a *partial trace*.

(If you feel like it, have a look at the fairly technical section [HV 3.4.5] now to see how these properties of the graphical calculus mean that you cannot have a sound semantic interpretation in infinite-dimensional Hilbert spaces.)

6.2 Bases

Let's forget for a minute that wires represent qubits, and instead let them represent complex vector spaces $\mathbb{C}^n$ of arbitrary (finite) dimension. A *basis* for this 'type' is a set of states $\{e_1, \ldots, e_m\}$ that is *minimal* in the sense that for all n-by-p matrices f and g, if $fe_i = ge_i$ for all $i \in \{1, \ldots, m\}$, then $f = g$. All bases of $\mathbb{C}^n$ are of the same size, namely n, which is the *dimension* of the space $\mathbb{C}^n$. You probably knew all this, but the point is that this is a purely string-diagrammatic definition: it only talked about composition of 'gates' and states, never about sums. It still talks about a basis as a set of states, though, and later we will see a way to make even that into a derived concept using 'gates' only.

Two states are called *orthogonal* when $\langle\phi|\psi\rangle = 0$, and a basis is called *orthonormal* when

$$\langle e_i|e_j\rangle = \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{if } i \neq j. \end{cases}$$

for all $i, j \in \{1, \ldots, m\}$.

Given an orthonormal basis $\{e_1, \ldots, e_n\}$, we can recover the *matrix* of a 'gate' f as

$$f_i^j = \langle e_j| f |e_i\rangle .$$

Notice that the cup is defined by

$$\sum_{i=1}^{n} |ii\rangle$$

for any orthonormal basis $\{|0\rangle, |1\rangle\}$ of $\mathbb{C}^2$. (And the same holds for any $\mathbb{C}^n$.) Similarly, the trace defined above is given by

$$\mathrm{Tr}(f) = \sum_{i=1}^{n} \langle i| f |i\rangle .$$

6.3 Sums

Another useful property of orthonormal bases is that they diagonalise the identity matrix:

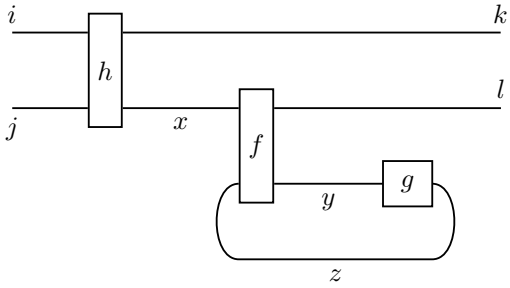
$$\mathrm{id} = \sum_{i=1}^{n} |i\rangle \langle i| .$$

This will be very useful to decompose string diagrams into linear combinations of simpler string diagrams. For example,

$$(g \circ f)_i^j = \sum_{k=1}^{n} g_k^j f_i^k . Nowread[KW2.1.5].$$

6.4 Tensor networks

All these sorts of calculations of sums of string diagrams culminate in *tensor networks*. These are a notation used often in many-body quantum physics as well as in machine learning. They look, for example, as follows:



They model a calculation to be done: how to *contract* a wire. In the above example, we could contract x, y, and z to get a 'tensor' with inputs i and j and outputs k and l. Contraction simply happens by multiplying the matrices in each box and sum over the connecting wires:

$$\sum_{x,y,z} f_{x,z}^{l,y} g_y^z h_{i,j}^{k,x} .$$

Computationally, the trick is to find algorithms that can do the tensor contraction efficiently in cases of interest, without calculating every entry of the very high-dimensional tensor with over all possible indices.

Special cases of tensor contraction are *tensor product*, *matrix multiplication*, and *trace*:

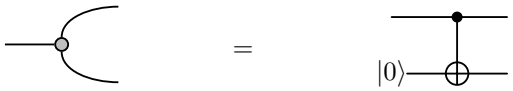
$$\begin{aligned}(f \otimes g)_{i,j}^{k,l} &= f_i^k g_j^l \\ (g \circ f)_{i,j} &= \sum_k f_i^k g_k^j \\ \mathrm{Tr}(f) &= \sum_i f_i^i\end{aligned}$$

Now read [KW 2.1.6].

7 Copying and deleting

A choice of basis, like the computational basis $\{|0\rangle, |1\rangle\}$ for $\mathbb{C}^2$, that we talked about last time, is a choice. To talk about matrices you need to make this choice, as developing in a different basis will lead to a different matrix. There is no such thing as a generic choice. Let's examine what we can say once we have made a choice of basis, trying not to mention the basis elements themselves at all.

Having made a choice of basis is the key difference between quantum information and classical information, between `qubit` and `bit` in OpenQASM. For example, it let's us *copy* classical information. The following quantum circuit, for example, copies the computational basis in the sense that it sends $|0\rangle$ to $|00\rangle$, and $|1\rangle$ to $|11\rangle$ – but it copies no other basis, so for example $|+\rangle = (|0\rangle + |1\rangle)/\sqrt{2}$ gets sent to $(|00\rangle + |11\rangle)/\sqrt{2}$ and not to $|+\rangle \otimes |+\rangle$.



Let's draw a copying circuit like that as on the left above, and investigate what properties such a 'gate' has, graphically. The key properties are:

- *associativity*: copying an input, and then copying one of the outputs once more should result in three equal outputs, and so should be the same as copying an input and then copying the other output once more;
- *commutativity*: copying an input should result in two equal outputs, so should be the same as copying an input and then swapping the two outputs;
- *unitality*: copying an input and then forgetting either of the outputs should result in the same as (just doing nothing to) the input.

Read [HV 4.1.1] to find out about these properties.

7.1 No-cloning

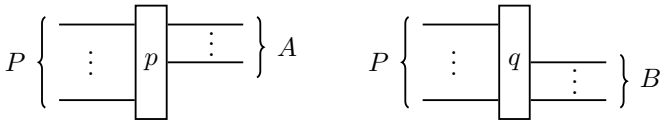
We can phrase the famous no-cloning theorem of quantum information theory purely graphically. It says that you cannot have a uniform copying machine (that can perfectly clone an unknown input) in a quantum setting. In other words: you cannot have cups and caps (entanglement), and at the same time have a uniform family of copying map on all wires, at the price of everything degenerating. The same holds a uniform deleting machine: not possible in the quantum setting.

Read [HV 4.2] (but skip Proposition 4.16); the text talks about categories, but all proofs are purely graphical and should make sense to you.

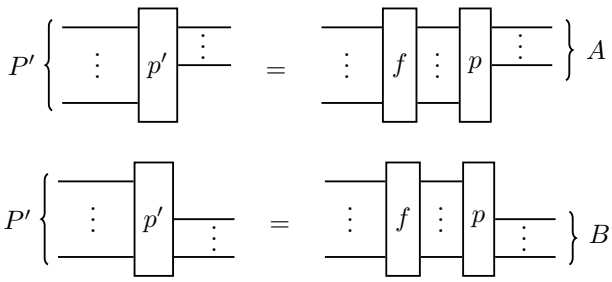
7.2 Products

As we saw in Section 4.3, entanglement means that bipartite states are not entirely determined by states of the two components. Vice versa, in a classical setting, giving a bipartite state is the same as giving one state for each component.

This is another graphical way to say that a theory is classical. Given two bunches of wires A and B, a *product* is a third bunch of wires P, together with two 'gates' $p\colon P \longrightarrow A$ and $q\colon P \longrightarrow B$ satisfying the following property: if P' is any other bunch of wires, and $p'\colon P' \longrightarrow A$ and $q'\colon P' \longrightarrow B$ any other 'gates', then there is a unique gate $f\colon P' \longrightarrow P$ such that $p' = p \circ f$ and $q' = q \circ f$. In pictures: if



then there is a unique f such that



This kind of property is called a *universal property*, because P is universal, in the sense that if there is any other P' pretending to do the same job, it must in fact factor through P.

To differentiate from tensor products, we will also call this a *cartesian product* or *categorical product*. Notice that the no-cloning theorem said that you cannot simultaneously have cups and caps and products, at the price of your whole theory degenerating.

Now read [HV 4.3] to find out the precise way in which uniform copying and deleting say that the wires in string diagrams must be carrying classical information.

7.3 Categories

You could interpret the string diagrams differently: wires do not just represent qubits or even vector spaces $\mathbb{C}^n$, but any other sort of *objects*; and 'gates' do not just represent unitary gates or even linear matrices, but any other sort of *morphisms*. All that is needed for a sensible interpretation is that morphisms of the right type can be composed sequentially

$$(B \xrightarrow{g} C) \circ (A \xrightarrow{f} B) = (A \xrightarrow{g \circ f} C)$$

and in parallel

$$(A \xrightarrow{f} B) \otimes (A' \xrightarrow{f'} B') = (A \otimes A' \xrightarrow{f \otimes f'} B \otimes B')$$

in a way that makes things like the interchange law true. Such a structure is called a *(monoidal) category*. (Especially the tensor product necessitates a lot of bookkeeping, that makes string diagrams much easier to work with than monoidal categories, but in principle they are nearly the same.)

For example, wires could represent sets, and 'gates' could represent binary relations between sets. This is a well-defined monoidal category, in which you can even interpret cups and caps sensibly. In fact, this is the category in which the teleportation protocol becomes interpreted as one-time pad cryptography.

Read [HV 0.1 and Example 3.18] for more information.

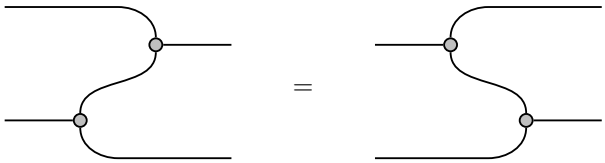
8 Classical data

A choice of orthonormal basis induces a copying map. Can you recognise when a copying map is induced by a choice of orthonormal basis?

8.1 Frobenius algebras

The key properties of a copying map that means it copies an orthonormal basis are the following. (Read [HV Sections 5.4.1 and 5.4.2] if you're interested in the proof.)

- associativity
- commutativity
- the *Frobenius law*: this property guarantees that the basis vectors are orthogonal (from which it follows that they form a basis), and uses both the copying map and its adjoint.



- *speciality*: this property guarantees that basis vectors have length 1, and uses both the copying map and its adjoint.

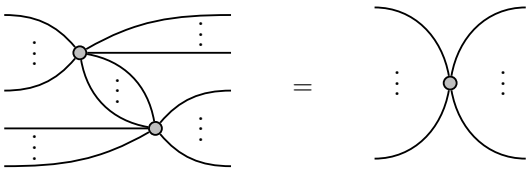
A wire is called a *Frobenius algebra* when it is equipped with a copying map satisfying these properties. Many other properties follow. Read [HV Definition 5.1, Lemma 5.4, Definition 5.5].

For example, if a wire carries a Frobenius algebra, then it automatically also has cups and caps: read [HV Theorem 5.15 and Proposition 5.16].

8.2 Spiders

For Frobenius algebras, 'only connectivity matters' holds in the extreme: any connected string diagram built from copying maps and their adjoints using sequential and parallel composition is the same. This property is also called the *spider theorem*. Read [HV Section 5.2].

Consequently, we can equivalently define a Frobenius algebra as a family of *spiders*: a family of 'gates' with m inputs and n outputs for any natural numbers m and n, satisfying the *spider fusion* rule



Colloquially: when two spiders shake legs, they fuse. In particular, this includes *wire rules* as edge cases:

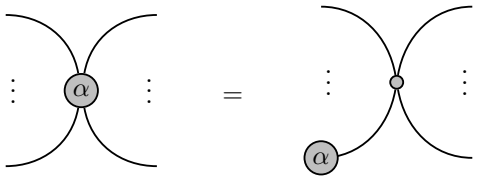
8.3 Phases

With only spiders available, we can only make graphical 'programs' that correspond semantically to specific matrices, namely the partial isometries diagonal in the corresponding orthonormal basis. What if we want to 'program' in string diagrams a matrix with other diagonal entries?

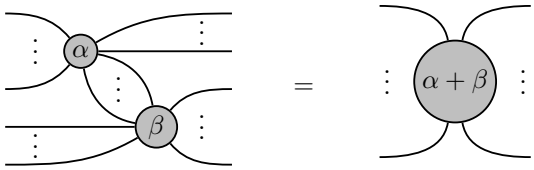
That is where phases come in. A *phase* is a state that interacts in a specific way with the copying map.

They correspond to unitary matrices that are diagonal in the orthonormal basis corresponding to the copying map. Using them we can code string diagrams for diagonal matrices with complex numbers on the unit circle as entries. This includes for example controlled rotation gates.

Purely abstractly from the string diagrams, phases for a copying map form a group: two phases can be added (using the adjoint of the copying map). Phases play particularly nice with spiders. We can decorate a spider with a phase by just writing it inside the dot:



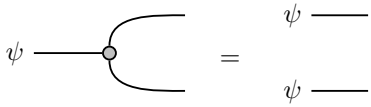
The spider fusion rule then becomes:



Now read [HV 5.5].

8.4 Bits

Armed with spiders, we can now make sense of bits. Whereas arbitrary wires respresent the data type of a qubit, states of a wire that are *copyable* by a chosen Frobenius algebra correspond to basis states, and thus to bits:



Let's have a look at our next quantum programming language, where bits are used in an essential way.

8.5 Qiskit

Qiskit is an open source programming language developed by IBM and built on top of the Python programming language. It integrates with OpenQASM. In fact, Qiskit more accurately refers to a family of integrated frameworks for quantum computation. The most important of these are:

- Qiskit Terra, a circuit description language with a Python-like syntax and the foundation of the Qiskit software stack. It also includes tools for the visualisation of circuits and quantum states, and for the optimisation of circuits for specific hardware devices.

- Qiskit Aer, which provides facilities for classically simulating those circuits, along with realistic noise models for those simultations.

Along with classical simulations, Qiskit can also be used to interact with various quantum hardware providers through the cloud.

Qiskit is provided as a Pypi package for Python. As a result, its syntax follows the rules of Python. For starters, we import the package as usual:

```
import qiskit as qs
```

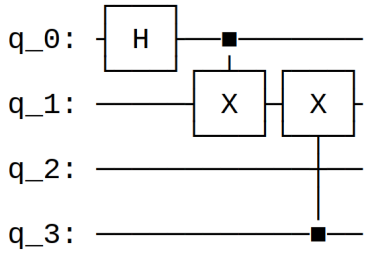
The core class for describing a quantum circuit in Qiskit is, unsurprisingly, the `QuantumCircuit` class. Its constructor takes up to two integer arguments, used to specify the number of quantum and classical registers (qubits and bits):

```
qc1 = qs.QuantumCircuit(4) # Construct a QuantumCircuit with 4 qubits
qc2 = qs.QuantumCircuit(4,3) # ... or with 4 qubits and 3 bits
```

We can then enact gates on the circuit by calling the corresponding methods of the class:

```
qc1.h(0)
qc1.cx(0,1)
qc1.cx(4,3)
```

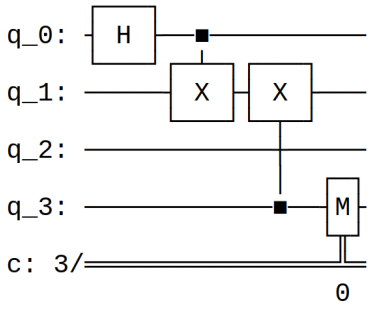
We can then visualise this circuit with `qc1.draw()`:



As you can see, the register of qubits is 0-indexed. Qiskit implements a large number of quantum gates in this way, see the documentation (or calling `help(qs.QuantumCircuit)` within Python).

Measurements are a little different: they of course require access to a classical bit to store the outcome of the measurement.

```
qc2.h(0)
qc2.cx(0,1)
qc2.cx(4,3)
qc2.measure(2,0)
```



We can then compose circuits with matching registers as follows:

```
qc3 = qs.QuantumCircuit(4)
qc3.cx(2,4)
qc3.cx(1,3)
qc1.compose(qc3).draw()
```

We can also build a `QuantumCircuit` whose quantum register is split into named subregisters. To understand this, we first need to introduce the `QuantumRegister` class, with an optional name:

```
qreg1 = qs.QuantumRegister(2,"register1")
qreg2 = qs.QuantumRegister(2,"register2")
```

`ClassicalRegister` has an analogous syntax for constructing classical registers. Then, we can build a `QuantumCircuit` from these registers:

```
qc3 = qs.QuantumCircuit(qreg1, qreg2)
qc3.h(0)
qc3.cx(qreg1[0],qreg1[1])
qc3.cx(qreg2[1],qreg2[0])
```

We can also initialise qubits with the following syntax:

```
qc3 = qc1.copy()
qc3.initialize([0,1], 2)
```

The first argument to `initialize` describes a qubit state as a pair of coefficients in the computational basis. The second argument describes the target qubit in the circuit for initialisation.

We can then compose circuits with matching registers:

```
qc4 = qs.QuantumCircuit(4)
qc4.cx(2,3)
qc4.cx(1,3)
qc1.compose(qc4)
```

Finally, we can output a `QuantumCircuit` to an OpenQASM string using the `qasm` method.

Qiskit also makes it possible to interact with various quantum hardware providers, as well as to run a simulation of a given quantum circuit on a classical backend. This allows one to extract (real or simulated) output states or outcome probabilites for measurements. The frontend for running circuits on actual quantum hardware is described in the documentation and depends on the provider in question.

We focus here on the classical simulation backends. These are provided via the `qiskit.providers.aer` submodule, specifically the AerSimulator class.

```
from qiskit.providers.aer import AerSimulator
```

In order to run the simulation, we need to set a initial state for each qubit. Here is an elementary example:

```
qc5 = qs.QuantumCircuit(1,0)
qc5.initialize([1,0],0)
qc5.h(0)
qc5.save_statevector()
simulator = AerSimulator()
qobj = qs.assemble(qc5, shots=1024)
result = simulator.run(qobj).result()
result.get_statevector()
```

with result:

```
Statevector([0.70710678+0.j, 0.70710678+0.j], dims=(2,))
```

We can also get the resulting probabilities for different outcomes of a measurement in the computational basis:

```
result.get_counts()
```

Putting this all together, we can simulate the output state for one of our circuits for a given input state:

```
qc6 = qs.QuantumCircuit(4)
qc6.initialize([1,0],0)
qc6.initialize([1,0],1)
qc6.initialize([1,0],2)
qc6.initialize([1,0],3)
qc6 = qc6.compose(qc1)
qc6.save_statevector()
qobj = qs.assemble(qc6)
result = simulator.run(qobj).result() # Do the simulation and return the result
result.get_statevector()
```

This returns the result:

```
Statevector([0.70710678+0.j, 0.+0.j, 0.+0.j,
 0.70710678+0.j, 0.+0.j, 0.+0.j,
 0.+0.j, 0.+0.j, 0.+0.j,
 0.+0.j, 0.+0.j, 0.+0.j,
 0.+0.j, 0.+0.j, 0.+0.j,
 0.+0.j],
 dims=(2, 2, 2, 2))
```

If we add measurements to each qubit at the end of the circuit (using the `measure_all` method), we can then simulate outcome counts for the measurement. By default, these counts are made over 1024 simulated runs of the circuit.

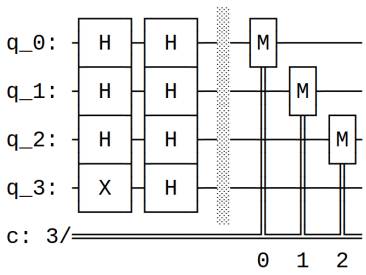
```
qc6.measure_all()
qobj = qs.assemble(qc6)
result = simulator.run(qobj).result() # Do the simulation and return the result
result.get_counts()
```

Although the topic is beyond the scope of this tutorial, one of the main selling points of Qiskit Aer is that it provides reaslistig noise models for simulating quantum circuits run on NISQ devices.

We can now put everything together to see our running example, the Deutsch-Jozsa algorithm, in Qiskit. For a complete explanation, see here.

```
# initialization
import numpy as np

# importing Qiskit
from qiskit import IBMQ, Aer
from qiskit.providers.ibmq import least_busy
from qiskit import QuantumCircuit, assemble, transpile

# import basic plot tools
from qiskit.visualization import plot_histogram

# set the length of the n-bit input string.
n = 3
```

```
const_oracle = QuantumCircuit(n+1)

output = np.random.randint(2)
if output == 1:
const_oracle.x(n)

dj_circuit = QuantumCircuit(n+1, n)

# Apply H-gates
for qubit in range(n):
dj_circuit.h(qubit)

# Put qubit in state |->
dj_circuit.x(n)
dj_circuit.h(n)

dj_circuit = QuantumCircuit(n+1, n)

# Apply H-gates
for qubit in range(n):
dj_circuit.h(qubit)

# Put qubit in state |->
dj_circuit.x(n)
dj_circuit.h(n)

# Add oracle
dj_circuit += balanced_oracle

# Repeat H-gates
for qubit in range(n):
dj_circuit.h(qubit)
dj_circuit.barrier()

# Measure
for i in range(n):
dj_circuit.measure(i, i)

# Display circuit
dj_circuit.draw()
```



A range of Jupyter notebooks are also provided which give further examples of quantum computation in the language.

9 Complementarity

With only one spider available, we can only draw string diagrams corresponding to some special matrices, namely those that interact well with the corresponding orthonormal basis. Part of the source of the power of quantum computing is that you can swap between two orthonormal bases (with things like the Hadamard gate and the Quantum Fourier Transform).

So let's have two bases, that is, two spiders. To differentiate them, they are typically with different colours: red and green, white and black, or sometimes others. Spiders of one colour still behave as before. But are there any rules that govern when spiders of different colours meet?

It turns out to be most advantagous to not just have any spiders, but to pick spiders that correspond to orthonormal bases that are *mutually unbiased*. This means that the bases are, in a sense, maximally incompatible: creating a state in the one basis and then measuring it in the other basis gives absolutely no information. This property of *complementarity* can be expressed purely graphically!

This is nearly the last part of the puzzle we will need for our graphical programming language. Using just two complementary spiders we can write programs for the CX gate, the CCX gate, and the H gate, which are known to be universal. In other words, you can write any quantum computation you want in string diagrams now.

Read [HV 6.1, 6.2]. (Bonus: if you're interested in why the complementarity law is in a sense the only choice, read [HV 6.3].)

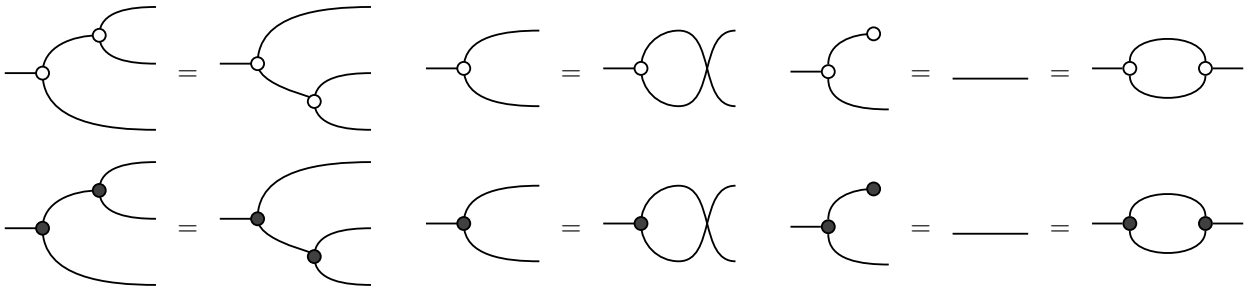
10 ZX calculus

We have now introduced all the ingredients we need for the graphical programming language of the ZX calculus: wires, scalars, and two colour spiders spiders. We summarise below. Read [KW 3.1, 3.2] (and [KW 3.3] if you feel like more) for more details.

10.1 Rules

There are two types of rules following which we may manipulate diagrams. First, the *graphical rules* that we already talked about above. We may deform a diagram in any way we like, as long as the connectivity is respected. The second type of rule pertains to the basic building blocks. These rules govern how you may combine several building blocks. There are quite a few of them, and we will now simply list them. We can think of these rules as *program transformations*.

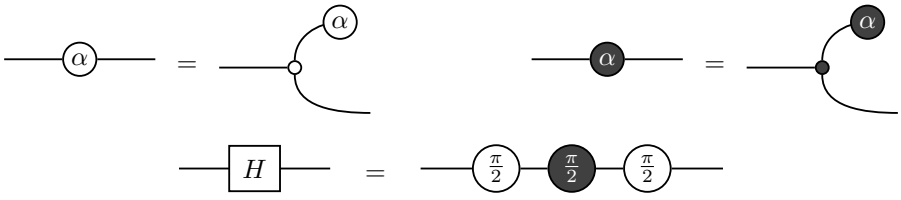
Monoid rules



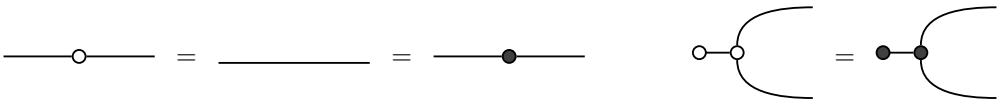
Frobenius rules

Fusion rules

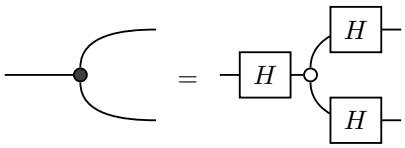
Identity rules For the following rules we first introduce a bit of shorthand notation:



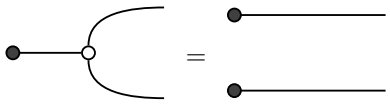
The identity rules now say:



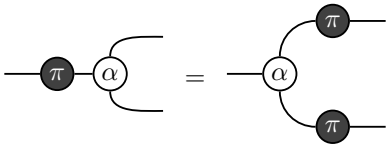
Colour change rule



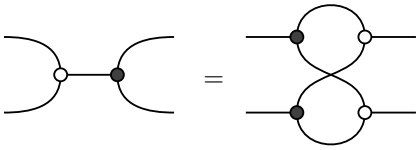
Copy rule



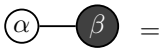
π-Copy rule



Bialgebra rule



Scalars rule



This rule says that we will ignore global scalar factors. It is more of a convenience, and we could have dropped this rule for the price of carefully inserting scalars into all the other rules.

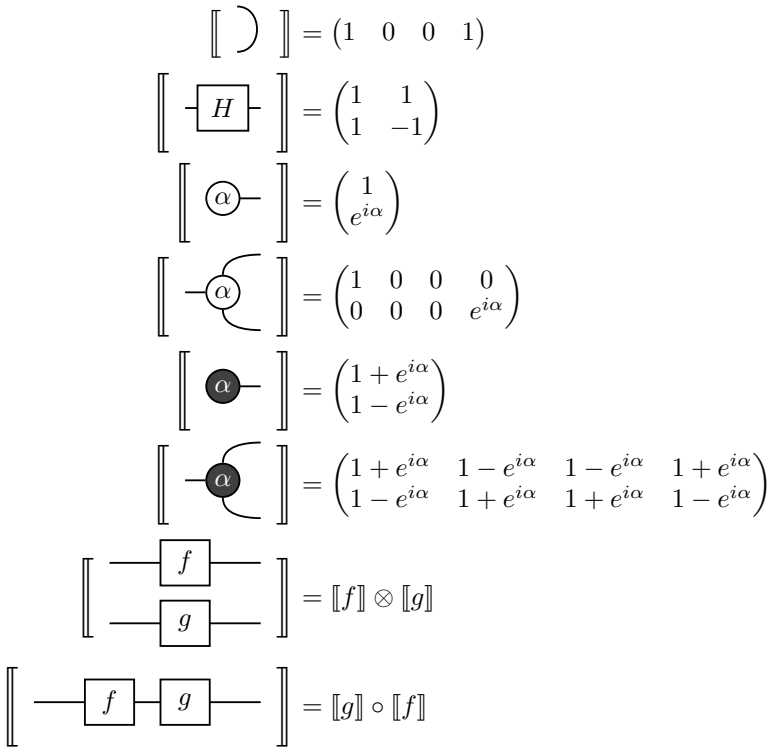
10.2 Soundness

The bialgebra rule says that these two orthonormal bases corresponding to the white and black spiders are at a maximal angle to one another, or complementary. The copy and π-copy rules are happy coincidences that hold for these two complementary bases. The *standard model* interprets a diagram of the ZX-calculus as an actual matrix in a way that respects the rules:

$$\left[\!\!\left[\quad \right]\!\!\right] = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

$$\left[\!\!\left[\quad \right]\!\!\right] = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\left[\!\!\left[\quad \right]\!\!\right] = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$



All the rules remain valid under this standard interpretation. (Try it out!) In other words, this standard interpretation is *sound*: any graphical manipulations done with ZX diagrams yield valid equalities between matrices under the standard interpretation.

Theorem 10.1 (ZX calculus is sound)**.** *Let D_1, D_2 be diagrams in the ZX calculus. If D_1 equals D_2 under the axioms of the ZX calculus, then $[\![D_1]\!] = [\![D_2]\!]$.* □

10.3 Completeness

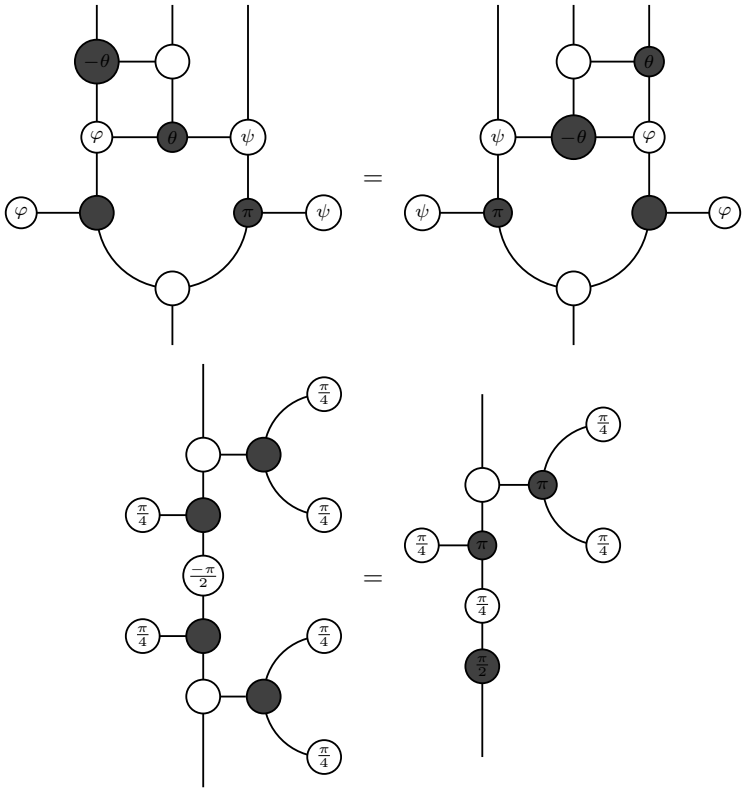
The ZX calculus captures the essence of quantum computation and is very efficient at it.

First, any possible linear transformation from m qubits to n qubits can be approximated up to arbitrary precision with ZX diagrams. In other words, the ZX calculus is *approximately universal*.

Theorem 10.2 (ZX calculus is approximately universal)**.** *For any 2^m-by-2^n matrix f, and any error margin $\varepsilon > 0$, there exists a diagram D in the ZX calculus, that only includes phases that are integer multiples of $\frac{\pi}{4}$, such that $\| [\![D]\!] - f \| < \varepsilon$.*

Second, the ZX calculus is *complete*. That is, if two matrices are equal, and are both given by some ZX calculus diagrams, is there always a graphical proof of this using only the axioms of the ZX calculus? This is

the case if we assume the following further two axioms (that are sound under the standard interpretation):



for any phases φ, ψ, θ that are multiples of $\frac{\pi}{4}$. Let's call the ZX calculus with these two extra rules the $\frac{\pi}{4}$-*ZX calculus.*

Theorem 10.3 (ZX calculus is complete)**.** *Let D_1, D_2 be diagrams in the $\frac{\pi}{4}$-ZX calculus. If $[\![D_1]\!] = [\![D_2]\!]$, then $D_1 = D_2$ under the axioms of the $\frac{\pi}{4}$-ZX calculus.*

10.4 PyZX

The ZX calculus is very amenable to automation. All a ZX calculation really is, is a bunch of finite labelled graphs, and a sequence of one of finitely many rules. Computers can handle this very well, and in fact search for such proofs for us themselves. The general problem of deciding whether two ZX diagrams implement the same unitary matrix, however, is hard (in a complexity-theoretic sense).

There are various software packages to handle ZX diagram rewriting with various degrees of efficiency, including Haskell and Java-based `Quantomatic`, Python-based `PyZX`, Rust-based `QuiZX`, and online `ZXLive`.

10.5 Quantum circuits

We will see in later lectures that you can translate quantum circuits into ZX diagrams and then optimise those ZX diagrams quickly. In fact, this leads to state of the art quantum circuit optimisation techniques. However, not all ZX diagrams correspond to circuits, and diagram rewriting may turn a circuit diagram into a non-circuit diagram. This is a consequence of the fact that the ZX calculus can express not just unitary matrices, but in fact all matrices.

11 Classical quantum circuits

Whereas general ZX diagrams are hard (in a complexity-theoretic sense) to rewrite algorithmically, special classes of ZX diagrams are much easier.

11.1 CNOT circuits and reversible computing

For example, quantum circuits made up of only CX gates are much easier to reason about (than arbitrary quantum circuits). This is because they can be represented much more efficiently than by an arbitrary 2^n-by-2^n matrix with entries in the complex numbers, namely by a 2^n-by-2^n matrix with entries in $\{0,1\}$. This means we can do linear algebra over the field $\mathbb{Z}/2\mathbb{Z}$ with 2 elements, which comes down to parity checking. Read [KW 4.1] for the details.

Another way to think about this is to consider computations that are *classical*, but *reversible*. Here, by classical, we mean that input states drawn from the computational basis always lead to outputs in the computational basis, and not superpositions of them. By reversible, we mean that the outputs can be derived from the inputs (duh), but also the inputs can be derived from the outputs. In a sense, the whole computation can be run in reverse. Notice that this is the case for CX:

$$\begin{aligned} |00\rangle &\mapsto |00\rangle \\ |01\rangle &\mapsto |01\rangle \\ |10\rangle &\mapsto |11\rangle \\ |11\rangle &\mapsto |10\rangle \end{aligned}$$

This model of computation is interesting for classical computer science too, because of the following reasons. By Landauer's principle, deleting the information of a bit takes work (in the thermodynamic sense of increasing entropy and disspating energy or heat), as does copying a bit. Therefore, if we built a computer that only runs reversible computations, by never copying or deleting bits, it would take much less energy to run (and therefore be faster).

But is it feasible to do only reversible computations? Yes, it is! Any function $f\colon \{0,1\}^n \to \{0,1\}^n$ on n-bit strings can be turned into a reversible one by *Bennett's trick*

$$\begin{aligned} \{0,1\}^n \times \{0,1\}^n &\to \{0,1\}^n \times \{0,1\}^n \\ (x,y) &\mapsto (x, y \oplus f(x)), \end{aligned}$$

where $\oplus$ denotes the XOR of two bits. This is also called *reversiblising* the function. The price to pay is that you need twice the memory. We will need this in a later lecture to handle oracular quantum algorithms.

In fact, the situation with CX above is typical. The CCX gate, also called the *Toffoli gate*, is known to be *universal* for classical reversible computing: any permutation between sets of size 2^n can be implemented by a circuit of Toffoli gates.

(Categorically, CNOT circuits thus correspond to the category of finite sets and bijections, whereas general quantum circuits correspond to the category of finite-dimensional Hilbert spaces and unitary matrices. From this perspective it is not surprising that you can handle the former algorithmically much easier than the latter.)

(You can base good programming languages on this idea. Start with classical reversible computing `Pi`, and add some constructs to upgrade it to quantum computing. Options include adding the Hadamard gate, adding square roots, or adding two copies of the basic reversible classical programming language and make them interact in a complementary way. Three these options led to the languages `HadamardPi`, `SqrtPi`, and `QuantumPi`. All three are sound, complete, and universal. Moreover, their rules are fully equational, unlike the ZX calculus, which either has uncountably many scalars, or rules like Euler decomposition that involve quantifiers. This means you can truly automate them. Moreover, the few rules make physical sense, unlike some of the ZX calculus rules which seem arbitrary. Finally, these languages by design cannot describe

non-unitary matrices, unlike ZX diagrams; any program you write down in them is in fact a valid physical computation, and universality says that vice versa any unitary matrix you can in fact implement. This is state of the art research, but if we have time towards the end of the course, and/or if you ask nicely, we can have a look at them!)

11.2 Phase-free ZX-diagrams

A similarly restricted class of ZX-diagrams that are pleasant to work with algorithmically are *phase-free* ones: those that have no Hadamard gates and that only allow phases 0.

There *is* an efficient algorithm to bring such diagrams into a normal form. Hence you *can* efficiently check whether two graphical programs in this form implement the same computation. Also, there *is* an efficient algorithm that turns a phase-free ZX diagram into a quantum circuit.

Read [KW 4.2].

12 Measurement

We have been cagey about measurement so far by using postselection. Today we look at a quantum programming language that really needs mid-circuit measurements: `Q#`. We will still remain a bit cagey about measurement, but will try to convince you that you can deal with it gracefully in a graphical programming language, and in practice keep using the mongrel method of sums of diagrams instead. First, read [KW 2.6.1].

To get a taste for how to deal with measurement fully graphically, read [HV 7.1 and 7.2] (but skip the bits about categories and only look at the bits about graphical diagrams).[1] Basically, we have to distinguish two types of wires: fat ones to represent quantum data, and thin ones to represent classical data. A spider from a fat wire to a thin one now represents measurement. (And a spider from a thin wire to a fat one represents conditional state preparation.) This way we can have both classical and quantum data types in our language, where classical data can be copied and deleted but quantum data cannot, and in fact use classical data for control flow. For example, this way we can 'program' measurement-based quantum computation graphically, where the rest of the program depends in a crucial way on the outcome of a measurement made halfway its execution.

12.1 Q#

Q# is a domain-specific programming language (DSL), aimed at writing quantum algorithms without having to worry about implementation details such as the physical layout of a quantum computer. The Q# language was developed by Microsoft, and is part of their wider Quantum Development Kit, which includes:

- Q# libraries implementing several standard quantum operations and algorithms.
- Integration with classical programming languages such as Python, C#, and F#, through Microsoft's Dot.net framework.
- Ways to run your Q# programs on different targets, such as simulators, resource estimators, or actual quantum hardware using Microsoft's Azure Quantum programme. There is an orchestration language in which you can specify to run the Q# program on a simulator a number of times, and once on quantum hardware, for example.

These last two points make Q# an interesting mix between an imperative and functional programming language: Q# itself is imperative, but it can be used in a functional way. It is also not (necessarily) a circuit description language, as quantum instructions are dispatched in order, and you can use measurement results as classical data in the continuation of your program.

The easiest way to use Q# is through Microsoft's Integrated Development Environment called Visual Studio Code, following the installation instructions. There is official documentation that is quite comprehensive. The following survey will be much briefer.

Q# is an imperative language. Functions, called `operation`s, take in variables, and may return values. For example, here is an operation that creates $|\pm\rangle$ states:

```
operation PlusMinus(b : Bool) : Qubit {
    use q = Qubit();
    if b { X(q); }
    H(q);
    return q;
}
```

[1] If you really want to know you can also look at:

- B. Coecke and S. Gogioso, "*Quantum in Pictures*", 2023.
- B. Coecke and A. Kissinger, "*Picturing Quantum Processes*", Cambridge University Press, 2017, Chapter 8.

The first line says that the input is a `Boolean` value, and the output is a `Qubit`. The second line introduces a variable `q`, that is initialised to be of type `Qubit`, standardly to the value $|0\rangle$. If the input Boolean was true, an `X` gate is applied to turn `q` into $|1\rangle$. Then, `H` applies a Hadamard gate, and finally, the qubit is returned.

Apart from Hadamard, you can have CX gates, with `CNOT`, which takes two `Qubit`s. In fact, `CNOT` will accept a list of qubits to target, and a qubit to control them with, like so:

```
operation Oracle(cs : Qubit[], t : Qubit) : Unit is Adj {
    for c in cs {
        CNOT(c, t);
    }
}
```

Here, `Unit` is a type that says that this operation does not return a value (like `void` in C). The annotation `is Adj` indicates that this operation implements a unitary transformation and has an adjoint specialisation. (There is also an annotation `is Ctl` indicating that the operation has a controlled specialisation, and `is Adj+Ctl` to say it has both.)

Here is an operation that implements $\alpha|0\rangle + \beta|1\rangle \mapsto \alpha|00\rangle + \beta|11\rangle$:

```
operation Share(a : Qubit) : (Qubit, Qubit) {
    use b = Qubit();
    CNOT(b,a);
    return (a,b);
}
```

To measure, there is an operation `M` that takes a `Qubit` and has output type `Result`. A variable of type `Result` can only have two values: `Zero` and `One`.

```
operation GenerateRandomBit() : Result {
    use q = Qubit();
    H(q);
    return M(q);
}
```

Measurement using `M` automatically discard the qubit. When you no longer need an unmeasured `Qubit`, you need to `Reset` it. The compiler will complain if you have not discarded all qubits in use by the end of the program.

There are other types than `Qubit` and `Result`, such as `Bool`, `Int`, `Double`, and `String`, and you can build custom data types on those basic ones such as arrays.

Deutsch-Jozsa

To execute a Q# program takes some syntax. Here is a full implementation of the Deutsch-Jozsa algorithm, using the operation `Oracle` above:

```
namespace DeutschJozsa {

    open Microsoft.Quantum.Canon;
    open Microsoft.Quantum.Intrinsic;
    open Microsoft.Quantum.Measurement;

    operation DeutschJozsa(n:Int, oracle:((Qubit[], Qubit)=>Unit)) : Bool {
      mutable isFunctionConstant = true;
      use (qn, q2) = (Qubit[n], Qubit());
      X(q2);
      ApplyToEachA(H, qn);
```

```
    H(q2);
    oracle(qn, q2);
    ApplyToEachA(H, qn);
    if (MeasureAllZ(qn) != Zero) {
      set isFunctionConstant = false;
    }
    ResetAll(qn);
    Reset(q2);
    return isFunctionConstant;
  }

  @EntryPoint()
  operation Main() : Bool {
    Message($"The operation Oracle is ");
    mutable b = DeutschJozsa(1,Oracle);
    if (b) {
      Message($"balanced.");
    } else {
    Message($"constant.");
  }
  return b;
}
```

As you can see, a very nice feature of the Q# framework is that you can continue to compute using the result of a measurement as if nothing happened. This happens in `Main` above, where a `Result` type is coerced into `Bool` so the conditional `if` can act on it.

There are several advanced libraries and features to explore. For example:

```
open Microsoft.Quantum.Diagnostics;
```

provides an operation called `DumpMachine`, which will output the (simulated) current state of the computation. For example, if a qubit is initialised in state $|0\rangle$, `DumpMachine` will say so:

```
|0>:     1.000000 +  0.000000 i  ==     ******************** [ 1.000000 ]
|1>:     0.000000 +  0.000000 i  ==                          [ 0.000000 ]
```

13 Clifford circuits

Phase-free ZX diagrams make for boring quantum programs. Let's allow a few more phases. If multiples of $\pi/2$ are allowed, you can get all *Clifford* circuits: those quantum circuits built from CX and H gates together with the gate

$$S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}.$$

These diagrams can still be brought under efficient algorithmic control. Read [KW 5.1, 5.2, 5.3].

Again, this is perhaps not surprising, because Clifford circuit can be simulated efficiently on a classical computer by the *Gottesman-Knill theorem*. In other words, there is no quantum advantage yet if you stick to Clifford circuits only. Somehow there is a phase transition if you go from phases being multiples of $\pi/2$ to multiples of $\pi/4$, which we saw can approximate all quantum circuits. Nevertheless, it's cool that all of this can be done purely graphically. (Bonus: read [KW 5.4] if you want to more about how to prove the Gottesman-Knill theorem using ZX methods only.)

14 Circuit optimisation

Alright, let's drop all the guard rails now, and consider arbitrary quantum circuits, or equivalently, arbitrary ZX diagrams. Although there are no known efficient algorithms anymore, there are still various useful ways the structure of ZX diagrams can be analysed.

First, there are *phase gadgets*, which can capture the algebraic phase polynomials graphically, govern CX+phase quantum circuits. They can be phrased in ZX calculus, and there are easy rules about how they combine, that can drastically simplify CX+phase quantum circuits. Read [KW 7.1].

Second, ZX calculus gives state of the art *quantum circuit optimisation* methods. We already saw how to turn a quantum circuit into a ZX diagram. Then, you can simplify the ZX diagram, for example via a path sum. Finally, you somehow need to turn the diagram back into a quantum circuit. This is the difficult part; it's called *quantum circuit synthesis*. But it can be done! Read [KW 7.2] to see how.

(Bonus: read [KW 7.6] if you want to learn more about optimisation not of quantum circuits but of arbitrary ZX diagrams.)

15 Quantum simulation

Phase gadgets have a more versatile cousin called *Pauli gadgets*. These give a way to regard quantum computations other than a circuit of logic gates, and can still be compiled in a structured way. Read [KW 7.3 and 7.4].

Other than computation, one of the most promising applications of quantum computers is *quantum simulation*. This is the situation where a, say, chemist, brings you a Hamiltonian and an initial state, and asks you about the final state after letting the system evolve for a specified duration. This question can be answered (approximately) using Pauli gadgets. Read [KW 7.5].

16 Oracles

Our example algorithm, Deutsch-Jozsa, solves a typical *promise problem*. The input (namely, functions $\{0,1\}^n \longrightarrow \{0,1\}$) is not completely unrestricted, but you're promised that it will take a certain form (namely, only constant and balanced functions are allowed). It does this by using the input function as an *oracle*. The figure of merit is how often the oracle is consulted during the computation.

But where do you find an oracle in the first place? If it is to be used in a quantum circuit, it had better be a unitary gate. So first you have to turn the function $f\colon \{0,1\}^n \longrightarrow \{0,1\}$ into a reversible function using Bennett's trick (that we saw in Lecture 11), and then turn that into a permutation matrix (which is automatically unitary).

Can you recognise which unitary gates arise this way as oracles for classical functions? Yes you can! Even purely graphically. Read [HV 6.2]. (Bonus: check [KW 3.3.4] for a concrete way to build oracles.)

16.1 Higher order functions

How should oracles be modelled in a quantum programming language? Ideally, we'd like for example the Deutsch-Jozsa algorithm as a piece of syntax that can take the oracle as an input rather than having the oracle hardwired in the code. This feature is called supporting *higher-order functions*: functions are able to take (other) functions as input, and return functions as output.

Semantically, this can be achieved using map-state duality (see Lecture 5.3). But it poses some questions about the programming language's type system. The no-cloning theorem means that quantum types need to be *linear*, in the sense that you have to use a variable of such a type exactly once: you cannot neglect to use it, and you cannot use it twice. In particular, the type system has to make sure that higher-order functions, that are passed along, are only called exactly once. (Categorically, this means you need *exponential objects*: your category needs to be *monoidal closed*.)

16.2 Quipper

Quipper is a quantum circuit description language. It is functional in nature, so when writing Quipper programs you can think in terms of gates being applied in real time to qubit variables, just as you would for a classical functional program.

Quipper is implemented as a domain-specific language within the functional programming language Haskell. Quipper was developed at Dalhousie University by Peter Selinger, and remains open source. Its original goal was to to accurately estimate and reduce the computational resources required to implement quantum algorithms on a realistic quantum computer. It comes with a standard library implementation of 7 nontrivial reference quantum algorithms; compiling them into quantum circuits tells you how many qubits are needed exactly.

Our brief survey will roughly follow the article An Introduction to Quantum Programming in Quipper. For much more information, see the homepage and the official documentation. Warning: Quipper is not easy to install.

Circuit model

Quipper can express circuits that allow both quantum and classical wires and operations in the same circuit. Quantum operations can be controlled by a classical wire, but not the other way around. An explicit measurement operation turns a quantum wire into a classical wire.

Quipper's circuit model also has scope: some (ancilla) wires are only used in part of a circuit, and may be explicitly initialised and terminated part-way the circuit.

There are three distinct phases of execution:

- *Compile time*: when Quipper syntax is translated into an executable classical program.

- *Circuit generation time*: when the resulting classical program generated a quantum circuit. Inputs whose value is known at this stage are called *parameters*. For example, the Deutsch-Jozsa algorithm is really a family of circuits, one for each natural number `n`.
- *Circuit execution time*: when the resulting quantum circuit is executed on (simulated) quantum hardware. Inputs whose value are only known at this stage are called *inputs*. For example, the inputs to Deutsch-Jozsa on `n` qubits are n input qubits.

To keep the different kinds of input separate, Quipper uses three basic types for bits and qubits.

- *Bits*: have type `Bool` at circuit generation time, and have type `Bit` as a classical Boolean input to a circuit.
- *Qubits*: have type `Qubit` and are only available as inputs at circuit execution time.
- `Bool` can be converted into `Bit`, but not the other way around. Because measurements can only occur at circuit execution time, their outcome is a `Bit`, not a `Bool`.

Quipper is *lazy*. That is, when it evaluates a circuit-producing function, that circuit is produced on the fly. That is useful for very large circuits, which therefore don't need to be stored in memory all at once. Circuits can also be consumed lazily.

Embedded language

Quipper is implemented as an embedded language within the host language Haskell, and the quipper compiler is really a wraparound for the Haskell compiler. Hence, Quipper really is a collection of data types, combinators, and functions within Haskell. However, not every Haskell program is a good Quipper program, and there is a preferred style of writing Quipper programs. If you don't adhere to that style, there is no guarantee that Quipper acts as expected. Because of this more or less mandated programming style, you can also think of Quipper as a language in its own right, without thinking about Haskell.

Primitive operations

Here is a very simple Quipper function:

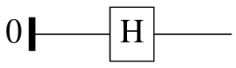
```
plus_minus :: Bool -> Circ Qubit
plus_minus b = do
  q <- qinit b
  r <- hadamard q
  return r
```

The first line is the type of the function. You could in fact leave it out, and the Quipper compiler will infer it. But for now it's useful to see that the input to the function is a boolean parameter. The output type `Circ Qubit` says that the function results in a circuit that returns a single qubit. The body of the function uses three operations, that are evaluated in the given order. First, a variable `q` of type `Qubit` is initialised with value `b`. Here, `False` corresponds to $|0\rangle$ and `True` corresponds to $|1\rangle$. Then, a `hadamard` operator is applied to the qubit, resulting in a new qubit `r`. Finally, `r` is returned as the output of the whole function.

You could evaluate this function as follows:

```
import Quipper
main = print_simple PDF (plus_minus False)
```

To obtain the following circuit as a PDF file:



Apart from the built-in function `hadamard`, you also have `qnot` for the X gate, and the infix operator `'controlled'`, that turns any operation into a controlled one. This code:

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

gives a 2-qubit circuit, where the second qubit is initialised $|0\rangle$, and a single CNOT gate is applied, that is, a circuit for the function $\alpha|0\rangle + \beta|1\rangle \mapsto \alpha|00\rangle + \beta|11\rangle$.

To *measure* a qubit, there is a function `measure`:

```
measurement :: Qubit -> Circ Bit
measurement q = do
  x <- measure q
  return x
```

If you no longer need a measurement result, or any other `Bit`, you can discard it:

```
cdiscard x
```

Deutsch-Jozsa

The Deutsch-Jozsa algorithm can now be written in Quipper as follows:

```
deutschjozsa :: (Qubit -> Qubit -> Qubit -> Circ (Qubit, Qubit, Qubit))
                -> Circ (Bit, Bit)
deutschjozsa (oracle) = do
  x <- qinit False
  y <- qinit False
  z <- qinit True
  hadamard x
  hadamard y
  hadamard z
  (x,y,z) <- oracle x y z
  hadamard x
  hadamard y
  (a,b) <- measure (x,y)
  return (a,b)
```

Here, `oracle` is a function of type `Qubit -> Qubit -> Qubit -> Circ (Qubit, Qubit, Qubit)`; a circuit that turns 3 input qubits into 3 output qubits. Unlike in OpenQASM, we can feed the Quipper version of `deutschjozsa` any such function we like. It could be, as in Lecture 3:

```
oracle x y z = do
  qnot_at z `controlled` [x, y]
  return (x,y,z)
```

But it could also be any other function `oracle` of the right type.

Advanced mechanisms

Let us briefly touch on some of the more advanced mechanisms that Quipper offers.

Quantum data types From the basic data types `Bit` and `Qubit`, you can build up more complicated data types, using tuples and lists. For example: `(Qubit,[Qubit])` is a type whose elements are pairs of a qubit and a list of qubits. Every quantum data types has an associated classical data type, in this case `(Bit,[Bit])`, and an associated boolean data type, in this case `(Bool,[Bool])`.

Generic functions The power of functional programming in Quipper comes to the fore with generic functions. We already saw `measure`, `cdiscard`, and `print_simple`. These are functions that can accept inputs of many different types. We won't go into detail here, but for an example, here is a generic version of the `plus_minus` function above:

```
plus_minus_generic a = do
  qs <- qinit a
  qs <- mapUnitary hadamard qs
  return qs
```

This function does not just apply to `Bool`, but can also produce, for example, a circuit of type `(Qubit,[Qubit])` out of `(Bool,[Bool])`.

Recursion In Quipper you can write circuit-producing functions that are recursive over any parameters known at circuit generation time. So, for example, you can recurse over a list of qubits. For a nice recursive implementation of the Quantum Fourier Transform, see An Introduction to Quantum Programming in Quipper.

Circuit operations You can also write functions that take a circuit, say of type `Circ [Qubit]`, and make new circuits based on it. For example, repeating a circuit a number of times, or reversing a circuit, or use a given circuit as a subcircuit in a larger circuit.

Finally, any classical reversible function can be turned into a circuit automatically by Quipper. Write a function such as

```
reversiblefunction :: (Bool, Bool, Bool) -> (Bool, Bool, Bool)
reversiblefunction x y z = y z x
```

and then turn it into a circuit by using the operation `build_circuit` just before:

```
build_circuit
reversiblefunction :: (Bool, Bool, Bool) -> (Bool, Bool, Bool)
reversiblefunction x y z = y z x
```

Semantics Quipper is one of few quantum programming languages with well-founded semantics. Haskell is well-studied academically, and many of its mathematical models extend to Quipper. For example, the crucial `Circ` construction is in fact a monad. However, some aspects of Quipper that would be useful for a quantum programming language are not well reflected in Haskell:

- *Dependent types*: Haskell has no mechanism to recognise that a parameter of type `Bool` is available at circuit generation time, but a `Bit` is only used at circuit execution time.
- *Linear types*: qubits cannot be copied or deleted, but terms of any Haskell type can.

There is a family of descendants of Quipper, called Proto-Quipper, that aim to make Quipper a standalone language and semantics to remedy these issues.

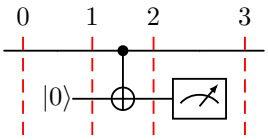
17 Advanced topics

Finally, depending on time and demand, let's discuss some advanced topics.

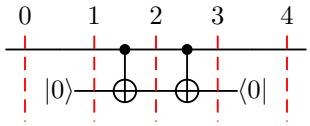
17.1 Uncomputation

Almost every quantum computation uses *auxiliary qubits*, that are not part of the input or output, but only store some temporary information. After they are no longer useful, they can be discarded. The easiest way to do this is by measuring them. That way, you can reuse your precious qubits. Over the course of a larger quantum program, such resource reuse can make a big difference.

But you have to be careful, as this can mess up the rest of your computation!



In the quantum circuit above, the top qubit is only used as a control, and hence you may expect the output to equal the input. But the mere fact that the auxiliary qubit is initialised, interacts with the data qubit, and is discarded changes this expected behaviour. Suppose that the input at step 0 is $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$. This evolves to $\frac{1}{\sqrt{2}}(|00\rangle + |10\rangle)$ at step 1, and to $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ at step 2. But after the measurement, at step 3, the output is $|0\rangle$ with probability $\frac{1}{2}$, and $|1\rangle$ with probability $\frac{1}{2}$. This behaviour occured because the auxiliary qubit was discarded while it was in a *dirty* state, that is, it was not guaranteed to be in the same state it was initialised in. To prevent this behaviour, the auxiliary qubit needs to be cleaned before discarding it. We can do this by inserting an *uncomputation* before discarding:



The inserted circuit between steps 2 and 3 uncomputes everything that happened to the auxiliary qubit, guaranteeing that it is measured in the same state it was initialised in, and therefore has no effect on the rest of the computation. Indeed, if the input state at step 0 is $|+\rangle$, then at step 3 the state is now $\frac{1}{\sqrt{2}}(|00\rangle + |10\rangle)$, and so the output at step 4 is again $|+\rangle$, as expected.

You could leave inserting uncomputations to the programmer, much like in a language like C the programmer is responsible for memory management. But it would be much nicer if the compiler could take care of this automatically, much like *garbage collection* in languages like Java. This is not an easy problem, however: uncomputations could be correctly be inserted in several ways and in several places, and optimising this is a hard (in the sense of complexity theory problem).

17.2 Silq

SILQ is a high-level quantum programming language developed by researchers at ETH Zurich. While it bears some similarity to Q#, it has a number of features that sets it apart, such as:

- a sophisticated type system with, among other things, type-level distinction between classical and quantum types (using `!`), classical and quantum subroutines (using the `qfree` annotation), and quantum subroutines with and without measurement (using the `mfree` annotation);
- automatic uncomputation of subroutines, preventing accidental measurement when variables leave scope.

- SILQ programs also support using various unicode symbols, though they all have ASCII alternatives and so are not strictly necessary. In this tutorial, we will avoid their use.

SILQ programs operate much like Q# programs in that they are allowed to mix classical and quantum computations, perform measurements at arbitrary places and branch on the result, and so forth. Unlike circuits, which are run directly on a quantum computer, we can think of SILQ programs instead as programs running on a classical computer *controlling* a quantum computer: this classical computer is able to continuously send quantum instructions, as well as the initial classical data to perform these on, to the quantum computer, and receive measurements back once the instructions have been executed. This is known as the *QRAM model* of quantum computation.

SILQ is most easily installed as a Visual Studio Code plugin, as described in the installation instructions, though brave souls can also install it from source.

Like most other programming languages, a SILQ program consists of a number of function definitions. A SILQ program is runnable if it contains a `main` function. The simplest possible SILQ program is something like

```
def main() {
  return false;
}
```

To run this program in Visual Studio Code, save it and press F5 (it should return 0).

SILQ supports a number of types and annotations to aid in programming. The simplest types are the primitive types such as `B` (Booleans), `N` (natural numbers), `Z` (integers), `int[n]` and `uint[n]` (n-bit signed and unsigned integers), and so on.

Given some types `s` and `t`, we can also form complex types such as `s -> t` (functions taking inputs of type `s` and producing outputs of type `t`), `s[]` (lists of elements of type `s`), `s^n` (for some natural number `n`; n-ary tuples of type `s`), and `!s` (the type `s` restricted to classical values).

While most of these should be self-explanatory, the classical types `!t` require some further exposition. A value of type `!t` is a value of type `t` which is guaranteed to be classical, i.e., not in a superposition of states. Distinguishing between classical and quantum values is useful, since we can do more with classical values than quantum ones. For example, classical values can be thrown away and duplicated, but quantum ones cannot, e.g., the function

```
def discard (n : !N) {
  return true;
}
```

is well-typed (since it is defined only on classical natural numbers), while the function

```
def discardQuantum (n : N) {
  return true;
}
```

is not (as it would try to discard arbitrary quantum natural numbers, which is not possible without first measuring). The type of the built-in measurement function `measure` also reflects this distinction between classical and quantum values, in that it has type `t -> !t` (for every type `t`).

It can also be useful to distinguish between functions that measure (parts of) their input, and functions which do not. A function that does not measure any (parts of) its arguments can be annotated with the `mfree` annotation, and so will not just have type `s -> t` but in fact `s -> mfree t`. Requiring that an argument function has an `mfree`-type can be used to ensure that a superposed quantum state can be safely handed off to this function without worry that the state is destroyed by a potential measurement.

A similar annotation is the `qfree` annotation: a function of type `s -> qfree t` is not just a function from `s` to `t`, but one which neither introduces nor eliminates superpositions. In other words, a `qfree` function is a classical function which can be applied to a quantum state without worry of introducing or eliminating

terms in a superposed state. This is very useful for describing oracles, as these are entirely classical functions (and, indeed, these algorithms only function correctly when they are).

A final very useful type-level feature of SILQ is generic parameters. A generic parameter is a classical value known at compile-time, on which a function definition may depend. This can be useful for defining families of functions which are flexible in that they can be defined generically for one or more external parameters. For example, one would expect an algorithm for adding n-qubit quantum integers to work the same for any choice of n, and so such an algorithm should be *generic* in n. Generic parameters generalise the *uniform families* found in Quipper. While normal arguments are given in parentheses, generic parameters are given in square brackets. For example, the function

```
def bitwise_not [n:!N] (bits : B^n) qfree {
  for i in [0..n) {
    bits[i] := X(bits[i]);
  }
  return bits;
}
```

takes a generic parameter `n` of type `!N` (classical natural numbers), and can be applied to a vector of Booleans of any length (since the input argument `bits` is of type `B^n`, which depends on the generic parameter `n`). Whatever the value of the `n` is, this function loops through the vector and applies the X gate to each entry. This allows us to apply this same function to vectors of different length, e.g., with the definition of `bitwise_not` above,

```
def main() {
  xs := bitwise_not(false, false, true);
  ys := bitwise_not(true, true);
  return (xs,ys);
}
```

is not only well-typed, it also returns `((1,1,0),(0,0))` when run as expected.

Note also that `bitwise_not` can be given the `qfree` annotation since all it ever does is apply X to each qubit in the vector, and X neither introduces nor eliminates superpositions. If we were to change this to apply an arbitrary function `B -> B` to each entry in the vector, e.g.,

```
def bitwise_map [n:!N] (bits : B^n, f : !(B -> B)) {
  for i in [0..n) {
    bits[i] := f(bits[i]);
  }
  return bits;
}
```

we would have to give up the `qfree` annotation, as we can not guarantee that `f` does not introduce or eliminate superpositions. For example,

```
def main() {
  xs := bitwise_map((false, true), H);
  return xs;
}
```

is well-typed (`H` is the built-in Hadamard gate), and running it returns the state $\frac{1}{2}(|00\rangle + |10\rangle - |01\rangle - |11\rangle)$, showing that the SILQ type system was justified in refusing to award `bitwise_map` the `qfree` annotation.

More information about SILQ (including a complete list of types, annotations, and other features) can be found in the official documentation.

Deutsch-Jozsa

```
def DeutschJozsa[n:!N](const f : B^n -> B^n) {
  // Initialise values.
  state := (0 : int[n]) as B^n;
  state[n] := X(state[n]);

  // Apply Hadamards.
  state := bitwise_map(state, H);

  // Apply oracle.
  state := f(state);

  // Apply Hadamards again.
  state := bitwise_map(state, H);
  state[n] := H(state[n]);

  // Measure and return result.
  return measure(state) == ((0 : int[n]) as B^n);
}
```

This function returns `false` if the given function is constant and `true` if it is balanced.

17.3 Pi

Depending on time, we can discuss a satisfying series of quantum programming languages based on the universal programming language for classical reversible computing, Pi, such as HadamardPi, SqrtPi, and QuantumPi.

17.4 Quantum Programming Zoo

There are many more quantum programming platforms than we discussed here. For example, you may be interested in Qurts, which has a Rust-style type system to deal with automatic uncomputation, or Qwire, which aims to certify circuits with correctness proofs rather than efficient compilation.

More generally, have a look at the Quantum Programming Zoo. Happy quantum programming!