First year report

Alexander S. Green
asg@cs.nott.ac.uk

December 2006

Abstract

If a function is logically reversible then it is possible to build a reversible circuit that performs the function. We will see later that it is also possible to build reversible circuits from functions that are irreversible. The process involves adding extra "heap" inputs, and extra "garbage" outputs. Reversible circuits are an interesting and important area of modern computer science. Not only do reversible circuits remove the limits on energy efficiency, but they are also a good starting point to learn about Quantum circuits, and more generally Quantum computation.

Computers as they currently operate, are highly inefficient. Processors are full of transitors which all produce heat. This heat is in effect lost to the surroundings as waste. As more and more transistors are squeezed onto smaller and smaller circuits this problem is only going to get worse. What's worse than this is the fact that even if computers can be made more efficient there is still a lower limit on the amount of energy that will be produced.

Irreversible operations lose information, and it can been shown that this loss of information directly leads to the limit mentioned above. In fact the limit can be given as $kTln2$ joules per bit of lost information (Where k is Boltzmann's constant, and T is the temperature of the system).[Lan00]

Reversible circuits can overcome this limit, as by definition they can not lose any information. Simply put, every reversible circuit composed with it's inverse, is equivalent to the identity function acting on all the inputs. It's quite easy then to see that at any point in a reversible computation you must have an equivalent amount of information as you started with.

There are however many irreversible functions that are commonly used in computers. A good example is that of the "Nand" gate which is a very common example of a universal constructor, meaning that any possible computation can be implemented using only "Nand" gates. It's easy to

see that the "Nand" function is irreversible just from it's truth table.

0	0	1
0	1	1
1	0	1
1	1	0

If the output bit is a 1, how do you know what the two input bits were?

For us though, what's more interesting is that Quantum circuits are a generalisation of reversible circuits. The reversible circuits are in fact a subset of quantum circuits, and even quantum bits or "qubits" can be thought of as a generalisation of classical bits. This report aims to give a detailed introduction to quantum circuits, building on what we know about reversible circuits, and introducing our concepts of heap and garbage to give us a universal set of quantum circuits. This report shall also introduce the current work we are doing on the subject.

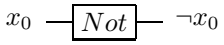
Contents

1 Reversible Circuits

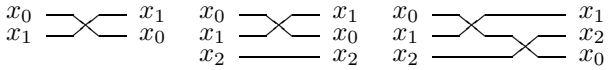
Reversible circuits can be thought of as isomorphisms between vectors (of equal length) of booleans. The booleans are a simple way of thinking of bits, with 0 as *False*, and 1 as *True*. You can see that for vectors of length n there must be $2^n!$ of these isomorphisms. For vectors of length one, this gives the expected number of two possible isomorphisms. Either $[True] \to [True]$ (and hence $[False] \to [False]$) or $[True] \to [False]$ (and hence $[False] \to [True]$). For vectors of length two, we already have 24 possible isomorphisms, so it would be useful to have some constructors we can use to build up reversible circuits.

There are five constructors that are used for this purpose, which are the Not gate, wire permutations, sequential composition, parallel composition, and finally the conditional.

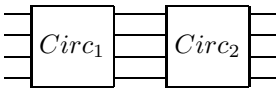
The Not gate is simply the second of the two 1 bit isomoprhisms given above, and can be denoted:



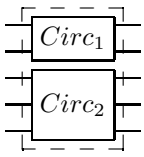
wire permutations can simply be thought of as moving bits about within the vectors. They are denoted as in the following examples:



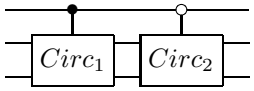
sequential composition is simply the joining (in sequence) of two reversible circuits with the same arity:



parallel composition is the joining (in parallel) of two reversible circuits. It can be thought of as the tensor product of the two:



the conditional has one wire as a control, and depending on the value of the control it will perform one of it's two argument circuits:

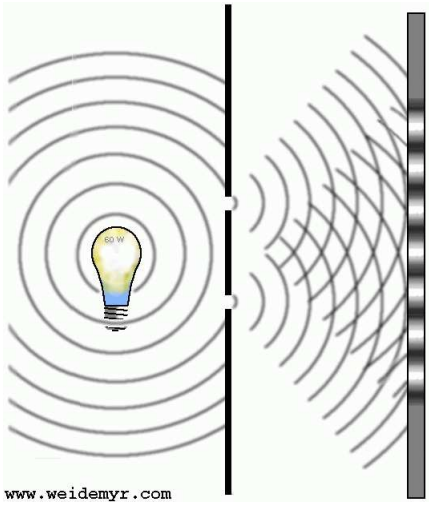


The identity function can be thought of as a wire permutation in which none of the wires actually swap places.

2 Qubits

What we have presented so far has been pretty straight-forward. The step from the classical realm into the quantum realm is a little more complicated. We've heard of the notion of a qubit, but what exactly is one.

Without wanting to go into too much detail about quantum mechanics it is nice to mention Thomas Young's double-slit experiment. Whereby light is shone through two slits towards a screen, and much to everyone's amazement a wave interference pattern appears on the screen:



What's interesting about this experiment is that light comes in discrete sized pieces known as photons. It is possible to repeat the above experiment with a light source that only emits one photon at a time. The same wave interference pattern is produced, and thus it is believed that each photon must pass through both slits simultaneously, and interfere with itself to produce the pattern. This behaviour whereby something can be in more than one state at the same time leads to the power behind quantum computation.

Another interesting extension to this experiment is to set up the apparatus as before, with a light source that produces a single photon at a time. This time adding detectors to each (or either) of the slits, that can detect when a photon passes through them. Now when you start the experiment you know which slit the photon goes through, but when you look at the screen there is no longer the wave interference pattern. This is similar to the concept of decoherence in quantum computers. A subject that will be discussed later.

A couple of very interesting books by Richard P. Feynman go into much more detail. [Fey94] [Fey71]

So, back onto the subject of qubits. What exactly is a qubit? Well, a qubit is a unit of quantum information. It has two base states ($|0\rangle$ and $|1\rangle$) that correspond very well to the states of a classical bit. However what makes the qubit special is that it can be put into a "super-position" of these two base states. In a very similar manner to the photon of light having to pass through both slits simultaneously. This means that the current value of a qubit ($|\psi\rangle$) can be thought of as a linear combination of the two base states.

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$$

with α and β as the complex amplitudes (ie. in $\mathbf{C}$) of $|0\rangle$ and $|1\rangle$ respectively, and the constraint that

$$|\alpha|^2 + |\beta|^2 = 1$$

There is however one draw back! When a qubit is measured it will only ever be in one of the base states. In effect, measuring the qubit collapses the superposition. It's a very similar concept as the extension to the double slit experiment mentioned above where adding a measurement device (the detector) causes the photon only to have travelled through one of the slits.

Luckily for us the base state that the qubit collapses into upon measurment is completely probablistic. The probablity that the qubit collapses into state $|0\rangle$ is simply $|\alpha|^2$, and the probability that the qubit collapses into state $|1\rangle$ is correspondingly $|\beta|^2$. For example, the following qubit

$$|\psi\rangle = \frac{1}{\sqrt{2}}\,|0\rangle + \frac{1}{\sqrt{2}}\,|1\rangle$$

is in an equal super-position of both the base states, and hence when measured there is an equal probability (of $\frac{1}{2}$) that the qubit will collapse into $|0\rangle$ or $|1\rangle$. This ability of qubits to be in more than one state at a time leads to the power of quantum computers. If a computation is run on a qubit in a superposition then in effect the computation has been run over both base states in parallel. We'll see shortly that as you add each new qubit you in essence double the number of base states available, and hence double the number of computations that are performed in parallel.

3 More than one Qubit

Now we know what a qubit is we can look into what happens when we have multiple qubits.

When you have more than one qubit it is possible for their states to become "entangled". We'll see what this means shortly, but first we'll quickly mention un-entangled multiple qubit states. These states are pretty much what you'd expect to get if you had more than one qubit. Each one acts individually (much like bits in classical circuits). The state of the system is simply the tensor product of the individual states of each qubit. These unentangled states are really not very interesting or useful, mainly because systems of unentangled

states can be simulated efficiently on classical computers. It is the entangled states that give us "quantum parallelism" and hence are of interest because they cannot be efficiently simulated on classical computers, and can therefore be used to create algorithms for quantum computers that are much more efficient than their classical counterparts. (We'll look at a few examples of quantum algorithms later)

Entangled states arise when qubits "depend" upon one another. Any multiple qubit state that cannot be simply thought of as the tensor product of each of the individual qubits, is said to be in an entangled state. Entangled states can be thought of as a linear superposition of all the possible "bit-string" base states of the qubits. For example a two qubit entangled state is a linear superposition of $|00\rangle$,$|01\rangle$,$|10\rangle$, and $|11\rangle$. That is:

$$|\psi\rangle = \alpha\,|00\rangle + \beta\,|01\rangle + \gamma\,|10\rangle + \delta\,|11\rangle$$

again with α,β,γ, and δ as the complex amplitudes, and the condition:

$$|\alpha|^2 + |\beta|^2 + |\gamma|^2 + |\delta|^2 = 1$$

and by the time you even have 3 entangled qubits there are already 8 possible base states. ($|000\rangle$,$|001\rangle$,$|010\rangle$,$|011\rangle$,$|100\rangle$,$|101\rangle$,$|110\rangle$, and $|111\rangle$) In fact the number of base states for an n qubit system is 2^n, which clearly shows an expontential growth in computational bases, for a linear growth in the number of qubits.

This gives us, for an n qubit system, the ability to put the "register" of qubits into an equal superposition of 2^n states, which means that if a computation is run over the qubits in the register, it in essence is run over each of the 2^n states in parallel. This is what is known as quantum parallelism, and isn't quite the same as classical parallelism as when you measure the register it will collapse into only one of the base states. The trick behind creating quantum algorithms is to make it such that the end state of the system is such that the probability of the measurment to be the "correct" answer is (arbitraily) high.

4 A Couple of Quantum Algorithms

The two most famous algorithms that effectively use quantum parallelism are Shor's factorisation algorithm [Sho94], and Grover's database search algorithm [Gro97].

4.1 Shor's Algorithm

Shor's algorithm is sometimes referred to as the quantum computer's killer-app. It is an algorithm for factoring large numbers, and has a time complexity of only $\bigcirc((logN)^3)$ for any number N. In comparison, the fastest known solution to the factoring problem on classical computers has exponential time complexity, and hence forms the basis of the RSA public key encryption system. Peter Shor's

discovery of this algorithm in 1994 lead to a massive resurgence of interest in quantum computing, not least because he had shown that given a sufficiently sized quantum computer it would be possible to break the RSA cryptographic standard.

The quantum part of the algorithm is in fact a solution to the order-finding problem, which it had previously been shown that the factoring problem could be reduced to efficiently on a classical computer. Order-finding is used to find the period of a function, and Shor's algorithm shows how this can be achieved using the inverse quantum Fourier transform.

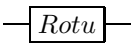
The algorithm computes the period of a function f, by first evaluating the function at every point, which on a quantum computer can all be done simultaneously by running the function over a super-position of all points. The clever part of the algorithm lies in the way that the quantum Fourier transform is then used to convert the resulting super-position into a state that will collapse into the correct solution with high probability.

4.2 Grover's Algorithm

Grover's algorithm is another influential algorithm in the area of quantum computing, as it too shows a speed up when compared to the best classical solution. It is an algorithm for searching in an unsorted database, and has a time complexity of $\bigcirc(N^{\frac{1}{2}})$ for a database of size N. Classically, the linear search is the best solution and has a time complexity of $\bigcirc(N)$. Although the speed up gained by this algorithm is in no way as impressive as the exponential speed up found by Shor, it was an important discovery, as because unlike Shor's algorithm, it is provably faster than any possible classical solution, rather than just the best known solution.

5 Quantum Circuits

Now we know a little about qubits, we can look at the quantum circuits that we use them in. As mentioned before they are an extension on the reversible circuits we have already seen. Apart from having to now think of the circuits as isomorphisms between qubit register states, we only need to add one other constructor, the Rotation gate.

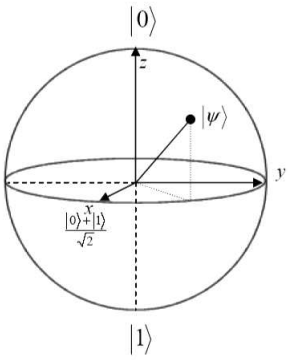


The rotation gate takes as it's argument, u, a unitary matrix representing the rotation. A couple of examples are the Quantum Not rotation, and the Hadamard operation.

$$Not = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \qquad Hadamard = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

where the quantum Not is a rotation of 180^o, and the hadamard is a rotation of 90^o. The hadamard opertation takes either of the base states of a single qubit into an equal superposition of both states.

The gate is called a rotation gate because the complex amplitudes in the linear superposition for a single qubit lend themselves nicely to picturing the state of the qubit as a single point on the surface of a unit sphere. This interpretation is commonly known as the "bloch sphere":



a rotation gate can be thought of as literally a rotation of that point about the sphere.

The fact that a rotation is given as a unitary matrix is pretty interesting, and in fact every quantum circuit can be described as a unitary matrix. The dimension of the matrix required to represent an n qubit operation is $2^n \times 2^n$, which again shows why a quantum computation can't be "efficiently" simulated on a classical computer.

5.1 Hilbert Spaces

Hilbert Spaces are a notion that is very useful for interpreting the unitary operations found in quantum computations. A Hilbert space is formally a vector space over either the Reals or Complex numbers, along with a function known as the inner product. We shall be using Hilbert spaces defined over the Complex numbers. A vector space is simply an abelian group along with scalar multiplication, meaning that there is a commutative addition operation, along with a null element refered to as zero, and the scalar multiplication function along with an identity element referred to as 1. For a Hilbert space H, scalar multiplication satisfies the following axioms: (where $x, y, z \in H$ and $\lambda, \mu, 1 \in \mathbb{C}$)
$\lambda(x + y) = \lambda x + \lambda y$
$(\lambda + \mu)x = \lambda x + \mu x$
$(\lambda\mu)x = \lambda(\mu x)$
$1x = x$
The inner product (denoted (x, y)) also satisfies the following axioms:
$(x, x) = 0 \Leftrightarrow x = 0$
$(x, x) \geqslant 0$
$(x + y, z) = (x, z) + (y, z)$
$(\lambda x, y) = \underline{\lambda(x, y)}$
$(x, y) = \overline{(y, x)}$

The inner product can be thought of as adding a geometric concept, as it can be used to introduce the magnitudes of vectors, denoted $\| x \| = \sqrt{(x,x)}$. Also we can then introduce the concept of angles between vectors, where $cos\theta = \frac{(x,y)}{\|x\|\|y\|}$. If two vectors in a Hilbert space have an inner product of zero then they are said to be orthogonal. The set of all normalised (e.g. of unit magnitude) orthogonal vectors in a Hilbert space is said to form an orthonormal basis of the Hilbert space. These geometric concepts make it easy to model many concepts central to quantum computation, such as a change of basis, or a projection.

6 Classical and Quantum Eqivalence

Classically it is sufficient to show that two circuits are equivalent by giving their common truth table. In the quantum realm one must show that both circuits are represented by the same unitary matrix. However as we saw before, the size of the matrix representing a quantum circuit is $2^n \times 2^n$ for a circuit with arity n. We also saw before that the classical circuits are a subset of the quantum circuits so a very useful thing to point out here is that if two quantum circuits are made only from elements in the classical subset of the constructors then it is sufficient to show that they are equivalent by showing that they are classically equivalent (In other words by giving the truth tables).

This holds because the unitary matrices that represent all of the quantum circuits that can be built from the classical constructors will only ever contain zeros and ones.

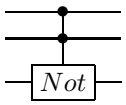
7 Classical Heap and Garbage

It has been previously mentioned that thinking about the classical reversible circuits is useful as ground work before extending the ideas into quantum circuits. So we shall first present here what we mean by heap and garbage in the realm of classical reversible circuits. We also previously saw that not all functions are logically reversible, but we need to be able to compute any arbitrary function for our circuits to be universal. A very similar example of a non-reversible function to the Nand gate we saw earlier, is the And gate. The truth table is again very useful to show this.

0	0	0
0	1	0
1	0	0
1	1	1

this time, if the result is a zero we are unable to deduce the inputs.

How is it possible to create a circuit that performs the And function, but is reversible? Well, look at the following reversible circuit:



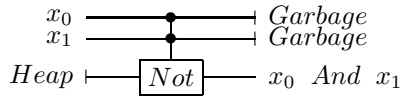
with it's truth table:

0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	0
0	1	1	0	1	1
1	0	0	1	0	0
1	0	1	1	0	1
1	1	0	1	1	1
1	1	1	1	1	0

Do you notice any connection between the first two inputs and the third output in the case when the third input is 0?

The circuit given above is known as the Toffoli gate, [Tof80], when the third input is set to zero, then the third output is always the And of the first two inputs.

So, now we can define our Heap and Garbage. The Heap shall be any extra inputs that are required for a reversible computation. They shall always have the value of $\vec{0}$. The Garbage shall be any extra outputs that are required fo a reversible computation, their values are obviously determined by the computation itself. In the And example just given, the Heap is just the third input, and the Garbage is both of the first and second outputs. The reversible And circuit with Heap and Garbage would be drawn:



it's important to note that when thinking of this as a reversible circuit that the contents of the Heap and Garbage are required, however when thinking of this as the And function the Heap and Garbage can be ignored.

8 Quantum Heap and Garbage

So, now what about making our quantum circuits with Heap and Garbage, as in the above classical example. Well it looks quite straight forward, we can simply as before introduce the notions of Heap and Garbage, with the Heap being any extra inputs we require, and the Garbage being any extra outputs that are produced. For the heap it's simply a question of redefining it such that all the heap qubits have the input value of $|\vec{0}\rangle$. But now we need to think more carefully about the garbage. We previously mentioned that when thinking of the circuits as performing a function that we can simply ignore the garbage outputs, however in the case of quantum circuits what happens if some of the useful outputs are entangled with the garbage? Obviously this could cause problems, for example if the garbage was to be measured it would cause the useful outputs to decohere. We obviously need something to control this behaviour, or at least work with it. In the next few sections i shall introduce the work we've been doing on this

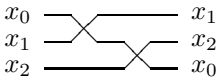
subject, looking at the problem from a categorical viewpoint, as introduced in our paper [GA06].

9 Reversible Computation

We model reversible computations by a groupoid $\mathbf{FxC}^{\simeq}$, that is for every morphism $\psi \in \mathbf{FxC}^{\simeq}(a,b)$ there is an inverse $\psi^{-1} \in \mathbf{FxC}^{\simeq}(b,a)$ such that ψ, ψ^{-1} are an isomorphism. We assume that the groupoid is strict, i.e. that any isomorphic objects are equal. This entails that $\mathbf{FxC}^{\simeq}(a,b)$ is empty, if $a \neq b$, consequently we denote homsets by $\mathbf{FxC}^{\simeq}\, a = \mathbf{FxC}^{\simeq}(a,a)$. We also assume that $\mathbf{FxC}^{\simeq}$ has a strict monoidal structure $I, \otimes$ which corresponds to parallel composition of computations and a special object of Booleans,denoted by $\mathbf{N}_2$. Since we are only interested in objects which can be generated from $I, \mathbf{N}_2, \otimes$ we can use natural numbers $a \in \mathbf{N}$ to denote the object 2^a. Hence we have that $I = 0$, $\mathbf{N}_2 = 1$ and $a \otimes b = a + b$. We write $[a] = \{i \in \mathbf{N} \mid i < a\}$ for the initial segment of $\mathbf{N}$.

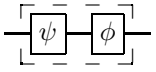
We characterise the morphisms, i.e. circuits, in $\mathbf{FxC}^{\simeq}a$ inductively and also give the inverses:

wires Given a bijection on initial segments $\phi : [a] \simeq [a]$ we write $\mathrm{wires}\,\phi \in \mathbf{FxC}^{\simeq}\, a$ for the associated *rewiring*. For example, the rewiring denoted pictorially as

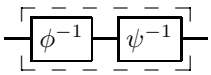


would have $\phi(0) = 2$, $\phi(1) = 0$, and $\phi(2) = 1$. The existence of wires follows from the strict monoidal structure, with the identity (id_a) being a special case of wires.

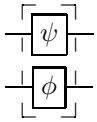
sequential composition combines two circuits of equal size (ie. with the same number of wires) in sequence. That is, given $\psi, \phi \in \mathbf{FxC}^{\simeq}a$ we construct $\phi \circ \psi \in \mathbf{FxC}^{\simeq}a$.



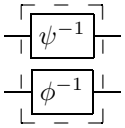
we can construct the inverse using ϕ^{-1} and ψ^{-1} to give $\psi^{-1} \circ \phi^{-1}$.



parallel composition combines any two circuits in parallel, and can be thought of as the tensor product. The size of the new circuit constructed is equal to the sum of the sizes of the original two circuits. That is, given $\psi \in \mathbf{FxC}^{\simeq}a$ and $\phi \in \mathbf{FxC}^{\simeq}b$ we can construct $\psi \otimes \phi \in \mathbf{FxC}^{\simeq}(a \otimes b)$.



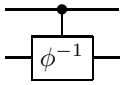
again we can construct the inverse using ψ^{-1} and ϕ^{-1}, this time to give $\psi^{-1} \otimes \phi^{-1}$.



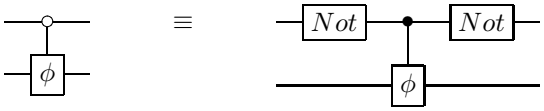
rotations count as any 1 "bit" operations. That is a rotation is any element of $\mathbf{FxC}^{\simeq}1$, and in the case of classical reversible circuits the only rotation available is the Not operation. So we have $\neg \in \mathbf{FxC}^{\simeq}1$ with $\neg^{-1} = \neg$. In the quantum case this would be any single qubit rotation.(i.e. a unitary operation in $U(2)$)

conditionals use a control wire to decide whether a computation should be performed. That is, given $\phi \in \mathbf{FxC}^{\simeq}a$ we can construct $id_a \mid \phi \in \mathbf{FxC}^{\simeq}(\mathbf{N}_2 \otimes a)$.

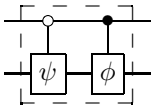
the inverse is again constructed using ϕ^{-1} giving $id_a \mid \phi^{-1}$.



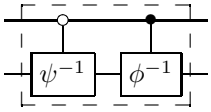
For ease of notation we shall also introduce the conditional that acts when the control wire is set to true. This conditional can be constructed from the conditional already given, and the Not operation (or rotation) as follows:



which for $\phi \in \mathbf{FxC}^{\simeq}a$ can be denoted $\phi \mid id_a \in \mathbf{FxC}^{\simeq}(\mathbf{N}_2 \otimes a)$. This naturally leads us to a choice operator, such that given two computations of the same size, the value of the control wire is used to govern which computation is done. That is, given $\psi, \phi \in \mathbf{FxC}^{\simeq}a$ we can construct $\psi \mid \phi \in \mathbf{FxC}^{\simeq}(\mathbf{N}_2 \otimes a)$, as follow:

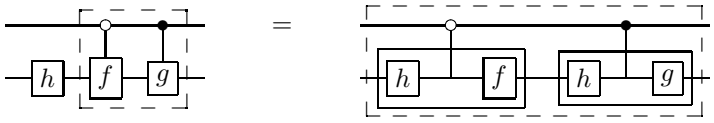


the inverse is once again given by ψ^{-1} and ϕ^{-1}, and constructed as $\psi^{-1} \mid \phi^{-1}$:

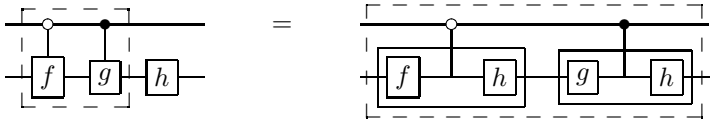


The laws governing wires, sequential composition and parallel composition follow from the categorical infrastructure. Additionally, we assume that the following equalities hold for conditionals:

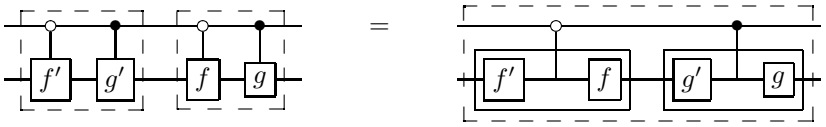
Firstly, we have for $f, g, h \in \mathbf{FxC}^{\simeq}a$ that $(f \mid g) \circ (\mathbf{N}_2 \otimes h) = f \circ h \mid g \circ h$ pictorially this can be shown as:



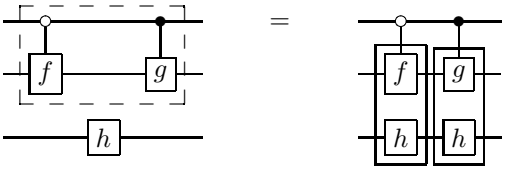
Secondly, we have for $f, g, h \in \mathbf{FxC}^{\simeq}a$ that $(\mathbf{N}_2 \otimes h) \circ (f \mid g) = h \circ f \mid h \circ g$ pictorially this can be shown as:



and thirdly, we have that for $f, f', g, g' \in \mathbf{FxC}^{\simeq}a$ that $(f \mid g) \circ (f' \mid g') = (f \circ f') \mid (g \circ g')$ again the pictorial representation for this would be:



We also have distributivity over $\otimes$ and $|$, such that given $f, g \in \mathbf{FxC}^{\simeq}a$ and $h \in \mathbf{FxC}^{\simeq}b$ we have that $(f \mid g) \otimes h = (f \otimes h) \mid (g \otimes h)$. This can again be given pictorially.



using this last axiom it is possible to simplify the first two to just be that $(h \mid h) = (id_1 \otimes h)$ or pictorially:

The next axiom that we introduce is that $id_a \mid id_a = id_{\mathbf{N}_2 \otimes a}$, and can be given (in it's most simple form) pictorially as:

Moreover, we have for $f, g \in \mathbf{FxC}^{\simeq} a$ that $(\neg \otimes id_a) \circ (f \mid g) = (g \mid f) \circ (\neg \otimes id_a)$, or pictorially that would be:

9.1 Examples of FxC$^{\simeq}$ categories

There are two obvious computational examples of $\mathbf{FxC}^{\simeq}$ categories: firstly there is the $\mathbf{FCC}^{\simeq}$ category of classical reversible circuits, and secondly there is the $\mathbf{FQC}^{\simeq}$ of quantum circuits. The difference mainly being in the rotations that are available. The extensional equality is given by interpreting circuits as permutations on $[a]$ in the classical case and as unitary operators on a-dimensional Hilbert spaces in the quantum case. Note that $\mathbf{FCC}^{\simeq} \hookrightarrow \mathbf{FQC}^{\simeq}$ and this embedding preserves extensional equality, because the unitary operators which can be obtained from definable circuits contain only 0 and 1 and hence can be obtained by embedding the corresponding permutation.

9.2 Bipermutative categories

A symmetric bimonoidal category $(\mathbf{C}, Z, \oplus, I, \otimes)$ is a category with two symmetric monoidal structures $(Z, \oplus)$ and $(I, \otimes)$ and distributivity isomorphisms $d \in A \otimes (B \oplus C) \simeq A \otimes B \oplus A \otimes C$ and $d' \in (A \oplus B) \otimes C) \simeq A \otimes C \oplus B \otimes C$ subject to a number of coherence laws [Lap72]. A bipermutative category is a symmetric bimonoidal categeory where all isomorphisms apart from $c^{\oplus} \in A \oplus B \simeq B \oplus A$ and $c^{\otimes} \in A \otimes B \simeq B \otimes A$ are identities. There are still a number of coherence laws to be satisfied such as:

$$\begin{array}{ccc} A \otimes (B \oplus C) & = & (A \otimes B) \oplus (A \otimes C) \\ {\scriptstyle A \otimes c^{\oplus}}\downarrow & & \downarrow{\scriptstyle c^{\oplus}} \\ A \otimes (C \oplus B) & = & (A \otimes C) \oplus (A \otimes B) \end{array}$$

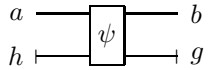
and

$$\begin{array}{ccc} A \otimes (B \oplus C) & = & (A \otimes B) \oplus (A \otimes C) \\ {\scriptstyle c^{\otimes}}\downarrow & & \downarrow{\scriptstyle c^{\otimes} \oplus c^{\otimes}} \\ (B \oplus C) \otimes A & = & (B \otimes A) \oplus (C \otimes A) \end{array}$$

Our models for $\mathbf{FCC}^{\simeq}$ and $\mathbf{FQC}^{\simeq}$ give rise to bipermutative categories, where $\mathbf{N}_2 = I \oplus I$ and all the laws stated above hold in all bipermutative categories. Hence, our development could be stated more abstractly in terms of bipermutative categories.

10 Irreversible computations

We derive a notion of irreversible computations from the given notion of reversible computation by defining the category **FxC**, where every morphism of the category represents an irreversible computation, but is in fact of the form $\psi' = (h, g, \psi)$ where h is a set of heap inputs, g is a set of garbage outputs, and ψ is the underlying reversible computation. So a morphism in $\mathbf{FxC}(a, b)$ can be given as a morphism in $\mathbf{FxC}^{\simeq}((a \otimes h), (b \otimes g))$ with the requirement that $(a \otimes h) = (b \otimes g)$. Pictorially we can represent an irreversible computation (h, g, ψ) as the reversible computation ψ where we mark heap and garbage explicitly:

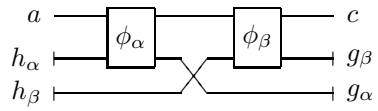


We also have that for any $\psi \in \mathbf{FxC}^{\simeq}a$ there is an equivalent circuit $\widehat{\psi} \in \mathbf{FxC}(a, a)$. More precisely this is given by the predicate:

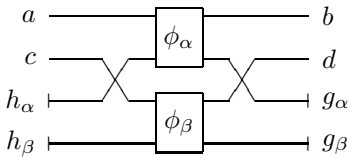
$$\frac{\psi \in \mathbf{FxC}^{\simeq}a}{\widehat{\psi} \in \mathbf{FxC}(a, a)}$$

such that $\widehat{\psi} = (0, 0, \psi)$, i.e. there is no heap or garbage.

We note that we can define sequential composition for irreversible computations: given $\alpha = (h_\alpha, g_\alpha, \phi_\alpha) \in \mathbf{FxC}(a, b)$ and $\beta = (h_\beta, g_\beta, \phi_\beta) \in \mathbf{FxC}(b, c)$ we define $\beta \circ \alpha \in \mathbf{FxC}(a, c)$, as



The identity can be obtained by lifting the reversible identity $id_a^{\mathbf{FxC}} = \widehat{id_a^{\mathbf{FxC}^{\simeq}}}$. It is straightforward to verify that **FxC** thus constructed is a category by using the monoidal identities in the underlying category of reversible computations. Moreover, **FxC** inherits the monoidal structure from $\mathbf{FxC}^{\simeq}$, e.g. given $\alpha = (h_\alpha, g_\alpha, \phi_\alpha) \in \mathbf{FxC}(a, b)$ and $\beta = (h_\beta, g_\beta, \phi_\beta) \in \mathbf{FxC}(c, d)$, we obtain $\alpha \otimes \beta \in \mathbf{FxC}(a \otimes c, b \otimes d)$ as:



The neutral element of the tensor, i.e. the empty circuit, can be obtained by lifting $I^{\mathbf{FxC}} = \widehat{I^{\mathbf{FxC}^{\simeq}}}$.

10.1 Examples of FxC categories

We can now extend our two example $\mathbf{FxC}^{\simeq}$ categories to **FxC** categories. We shall call these **FCC** for the category of finite classical computations, and

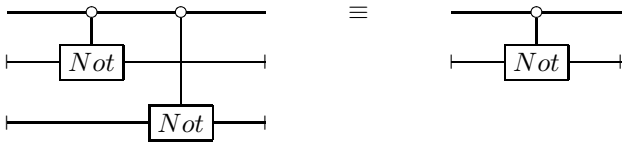
FQC for finite quantum computations. The extensional equality in the classical case is given by interpreting morphisms as functions on finite sets: $(h, g, \phi) \in \mathbf{FCC}(a, b)$ is interpreted as $\pi_g \circ [\![\phi]\!] \circ (0^h, -) \in [a] \to [b]$, where $[\![\phi]\!] \in [a \otimes h] \to [b \otimes g]$ is the associated permutation, $(0^h, -) \in [a] \to [a \otimes h]$ initialises the heap and $\pi_g \in [b \otimes g] \to b$ projects out the garbage.

In the quantum case we interpret circuits as superoperators (e.g. see [Sel04], or [VAS06] for an implementation in Haskell). Superoperators are morphisms on density operators, which are positive operators on the a-dimensional Hilbert space. A superoperator $f \in \mathbf{Super}(a, b)$ is a linear function mapping density operators on a to density operators on b, which preserve the trace and are stable under $\otimes$. Analogously to the classical case, we interpret $(h, g, \phi) \in \mathbf{FQC}(a, b)$ as $\mathrm{tr}_g \circ [\![\phi]\!] \circ 0^h \otimes - \in \mathbf{Super}(a, b)$, where $[\![\phi]\!] \in \mathbf{Super}(h \otimes a, g \otimes b)$ is the superoperator associated to the unitary operator given by interpreting the reversible circuit ϕ. $0^h \otimes - \in \mathbf{Super}(a, a \otimes h)$ initialises the heap and $\mathrm{tr}_g \in \mathbf{Super}(g \otimes b, b)$ is a partial trace which traces out the garbage.

11 Equivalence

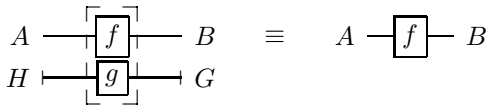
In the reversible case the equality of definable circuits is the same in the classical case and in the quantum case, but this doesn't hold for irreversible computations. For example, in the classical case the following two circuits would be equivalent:

However, this equivalence does not hold when we move into the category of finite quantum computations **FQC**. This is because in quantum computation the control wire (or qubit) can become entangled with the target wire (qubit). However there is another similar equivalence that holds in **FQC**:



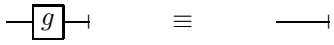
This is akin to von Neumann's measurement postulate. So, how now can we characterise the equivalences which should always hold?

We have developed three laws to try and characterise these equivalences, that hold in both **FCC** and **FQC**. The first law is that of garbage collection. It states that if a circuit can be reduced into two smaller circuits such that one part of the circuit only acts on heap inputs and on garbage outputs, then that part of the circuit can be removed.

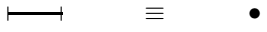


The second law is of the uselessness of garbage processing. This states that if a circuit can be reduced into two smaller circuits such that one part of the circuit only has an effect on garbage outputs, then that part can be removed.

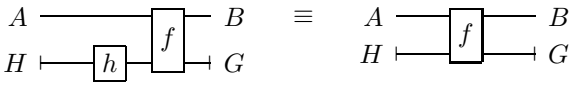
this can be alternately stated as saying that if the only outputs of (part of) a circuit are garbage outputs, then this is equivalent to just having garbage.



and similarly we can now simplify the first law to state that a wire that simply connects the heap to the garbage is equivalent to having nothing.

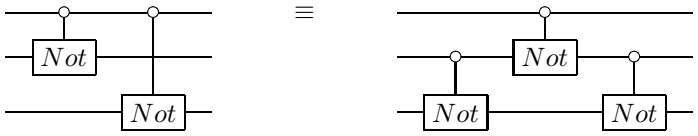


The third law is of the uselessness of heap preprocessing. This states that if a circuit can be reduced into two smaller circuits such that one part of the circuit only has effect on heap inputs, and the effect on the zero vector is the identity, then that part can be removed.
$if \quad h\vec{0} = \vec{0} \quad then$



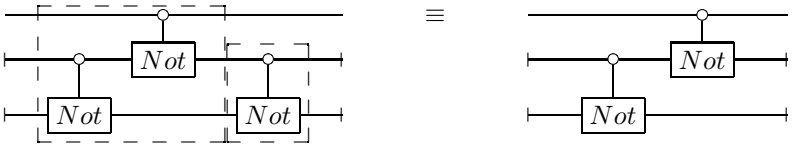
An alternate notation for this would again be to state that if (part of) a circuit only has heap inputs, and its effect on the zero vector is the identity, then this is equivalent to just having a heap.
$if \quad h\vec{0} = \vec{0} \quad then$

We can already use these laws to give a proof of the measurement postulate. The first step is to show the equivalence of

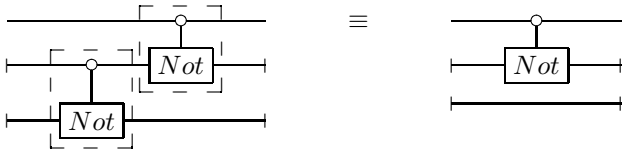


This is simple as you will notice there is no heap or garbage, so we know that the circuits are in $\mathbf{FQC}^{\simeq}$, and in fact only use the elements from $\mathbf{FCC}^{\simeq}$. Thus equivalence can be shown by looking at the truth tables, which are the same.

The third controlled not is eliminated using the second law:



The controlled Not operations preserve the zero vector, so we can eliminate the first one using the third law:

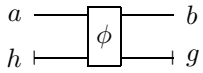


Finally the bottom wire can be removed by use of the first law:

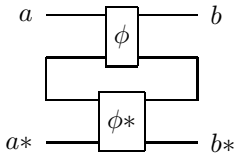
12 Further work

12.1 Dagger Categories

In his recent paper [Sel05], Selinger introduced the concept of dagger categories. It has been suggested that our model introduced above can be given alternately as morphisms in Selinger's CPM category. The circuit



can be thought of as the construction



in the CPM category.

12.2 The Measurement Calculus

One of the latest ideas in quantum computing is that of the one-way quantum computer [RB01]. The new idea involved is that qubits are used as a resource that get used up as the computation progresses. Physicists are especially excited about this idea as they believe that it is more likely that this sort of quantum computer could actually be created. One-way quantum computation is also known as Measurement based quantum computing, as it relies on the computation propogating through the "cluster" of qubits via measurements in various bases.

A computation is modelled as a sequence of measurements, whose various bases can be dependent on the outcome of previous measurements. The computation propogates through the "cluster" via the entanglements set up between neighbouring qubits duing the initialisation of the cluster.

To make the process of designing these computations easy, a Measurement Calculus has been proposed [DKP04]. It has been shown that any quantum circuit can be described in the Measurement calculus, and thus it should be possible to create a compiler for creating measurement based computations from quantum circuit descriptions. This idea could also be extended to QML, and QML programs could be compiled into measurment calculus patterns.

12.3 The Quantum IO Monad

Haskell provides Monadic programming constructs to enable computations that may involve side effects, one of the main Monads provided in Haskell is the IO Monad, which contains all the various IO functions that can be used in a Haskell program. Monads are used to contain impure functions that can produce side effects, and wrap them so as to create pure functions whose results may include a description of any side-effects that occured. These Monadic bindings enable any side effects that may occur to propogate through the program until they can be sensibly dealt with. A monad in Haskell is defined as a type constructor, along with a *return* function, and a bind function denoted $\gg\!=$. The return function is used to put values of any datatype into the monad, and the bind function is used to apply functions of datatypes outside the monad to the value of that datatype contained within the monad. The result of the function being bound must be in the monad.

The type constructor for the QIO monad is:

$$\begin{aligned}&\textbf{data } \mathit{QIO}\ a = \mathit{QReturn}\ a\\ &\qquad\qquad \mid \mathit{MkQbit}\ (\mathit{Qbit} \to \mathit{QIO}\ a)\\ &\qquad\qquad \mid \mathit{ApplyU}\ \ U\ (\mathit{QIO}\ a)\\ &\qquad\qquad \mid \mathit{Meas}\ \mathit{Qbit}\ (\mathit{Bool} \to \mathit{QIO}\ a)\end{aligned}$$

the return function can simply be defined as the QReturn constructor, and the bind function needs to be given for each constructor as follows:

$$\begin{aligned}&\textbf{instance } \mathit{Monad}\ \mathit{QIO}\ \textbf{where}\\ &\quad \mathit{return} = \mathit{QReturn}\\ &\quad (\mathit{QReturn}\ a) \gg\!= f \ \ = f\ a\\ &\quad (\mathit{MkQbit}\ g) \gg\!= f \ \ = \mathit{MkQbit}\ (\lambda x \to g\ x \gg\!= f)\\ &\quad (\mathit{ApplyU}\ \ u\ q) \gg\!= f = \mathit{ApplyU}\ \ u\ (q \gg\!= f)\\ &\quad (\mathit{Meas}\ x\ g) \gg\!= f \ \ = (\lambda b \to g\ b \gg\!= f)\end{aligned}$$

The type constructors of the QIO monad describe the operations that can be performed on a sort of "Quantum Register". *MkQbit* relates to making qbits available for the computation, *ApplyU* relates to the application of an actual quantum computation, and is used to apply a unitary operation to the relevant qbits that have been initialised. We'll see shortly the definition of a unitary operation. The last constructor, *Meas* relates to the final measurement of the Qbits after the computation has taken place.

So, we've seen that a quantum computation is defined as a unitary operation, which are defined in the monoid of unitary operations, denoted U. A monoid in Haskell is again defined as a type constructor, with one of the constructors being denoted as the identity element, or *mempty*. The binary operation of the monoid is denoted *mappend* and the definition must be given. In the case of the U monoid, the type constructors and the corresponding operations are defined by:

```
data U = UReturn
  | Rotate Qbit Rot U
  | Swap Qbit Qbit U
  | Cond Qbit U U

instance Monoid U where
  mempty = UReturn
  mappend Ureturn u          = u
  mappend (Rotate x r u) u'  = Rotate x r (mappend u u')
  mappend (Swap x y u) u'    = Swap x y (mappend u u')
  mappend (Cond x u u') u''  = Cond x u (mappend u' u'')
```

The type constructors of U represent the various operations that are required such that any unitary operation can be constructed, they correspond very closely to the operations described early for Quantum Computations. The *Rotate* constructor is for any single qubit rotation, and the actual rotation to be performed is the given *Rot* which we'll look at more closely later. The *Swap* constructor is used to swap the position of the 2 given qubits. Finally the *Cond* constructor is for conditionals, and the looks at the relevant qubit to decide whether or not the given unitary is run. The *mappend* operation is used to build up bigger unitary operations from smaller ones.

All single qubit rotations can be defined as a unitary 2 by 2 matrix, and so a rotation as used in the U monoid can be defined as 4 complex numbers that represent the entries in the corresponding unitary matrix. Some common single qubit rotations can be given as examples, including the X rotation, which corresponds to the classical Not, and the Hadamard rotation which is used to take any qubit from it's base state into an equal superposition of both base states.

```
type RR = Float
type CC = Complex RR
type Rot = ((CC, CC), (CC, CC))

rx, rh ∈ Rot
rx = ((0, 1), (1, 0))
rh = ((1 / sqrt 2, 1 / sqrt 2), (1 / sqrt 2, −1 / sqrt 2))
```

because all rotations must be unitary, it is possible to create thier inverses using the conjugate transpose as follows:

```
rrev ∈ Rot → Rot
rrev ((x00, x01), (x10, x11)) = ((conjugate x00, conjugate x10), (conjugate x01, conjugate x11))
```

and as all unitaries are by definition unitary, it is possible to create their inverses too:

```
urev ∈ U → U
urev URReturn       = URReturn
urev (Rotate x r u) = mappend (urev u) (rotate x (rrev))
urev (Swap x y u)   = mappend (urev u) (swap x y)
urev (Cond x u u')  = mappend (urev u') (cond x (urev u))
```

It's possible now to build up quantum computations, but the aim of the QIO monad is to enable these computations to be simulated. This is achieved through lifting the U monoid into another monoid named Unitary. Quantum computations that have been constructed in the U monoid must be lifted into "Pure" computations in the Unitary monoid, where qubits are assigned to the computations such that the computations can be run, thus changing the state of the qubits. The way a computation is then actually simulated by a user is by the use of a PMonad, which is a monad along with a *merge* function that describes how to display the state of the qubits.

The IO Monad can be extended into a PMonad by adding a *merge* function that uses the random number generator of the IO Monad to probabalisticly give each of the qubits a true or false value.

```
class Monad m ⇒ PMonad m where
  merge ∈ RR → m a → m a → m a

instance PMonad IO where
  merge pr ift iff = do pp ← Random.randomRIO (0, 1.0)
    if pr > pp then ift else iff
```

Another PMonad can be defined such that the two probabilities are given as part of the result instead of being used to "collapse" the qubit amplitudes, as in the example above.

```
data Prob a = Prob{ unProb ∈ Vec RR a}

instance Monad Prob where
  return = Prob ∘ return
  (Prob ps) >>= f = Prob (ps >>= unProb ∘ f)

instance PMonad Prob where
  merge pr (Prob ift) (Prob iff) = Prob ((pr <*> ift) <+> ((1 − pr) <*> iff))
```

Then the eval function only needs to be coded once, but takes the PMonad as one of it's arguments.

With the QIO Monad it's now possible to create quantum computations and evaluate them. Some example QIO programs are given below:

```
rbit ∈ QIO Bool
rbit = do x ← mkQbit
  applyU (rotate x rh)
  b ← meas x
  return b
```

The above program would create a random bit, by creating a qubit, applying the Hadamard rotation, and then measuring the qubit. Another example is a small program that creates a 2 qubit bell state. It proceeds by creating a qubit, applying the Hadamard to it, then making another qubit which is entangled with the first qubit using a conditional Not rotation. The 2 qubits are measured, and

a pair of the two measurements is returned. The bell state means that the qubits are entangled such that they will always collapse to the same state as one another.

$bell \in QIO\ (Bool, Bool)$
$bell = \mathbf{do}\ x \leftarrow mkQbit$
  $applyU\ (rotate\ x\ rh)$
  $y \leftarrow mkQbit$
  $applyU\ (cond\ x\ (rotate\ y\ rx))$
  $b \leftarrow meas\ x$
  $c \leftarrow meas\ y$
  $return\ (b, c)$

Now that we have created the QIO Monad, we would like to come up with larger examples, including implementations of Shor's and Grover's algorithms. It should also be possible to use larger quantum data structures than individual qubits, creating them in the same way that classical data structures are defined from classical bits. It should again be possible to construct QIO programs from QML programs, and thus use the evaluation functions to simulate the running of QML programs.

References

[DKP04] Vincent Danos, Elham Kashefi, and Prakash Panangaden. The measurement calculus. arXiv:quant-ph/0412135, 2004.

[Fey71] Richard Phillips Feynman. *Lectures on Physics: Quantum Mechanics v. 3 (World Student S.).* Addison Wesley, 1971.

[Fey94] Richard P. Feynman. *Character of Physical Law (Modern Library).* Random House USA Inc, 1994.

[GA06] Alexander Green and Thorsten Altenkirch. From reversible to irreversible computations. to appear in the proceedings of QPL 2006, June 2006.

[Gro97] L. Grover. Quantum Mechanics helps in searching for a needle in a haystack. *Physics Review Letters*, 79(2):325–328, 1997.

[Lan00] R. Landauer. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development*, 44(1):261–269, 2000.

[Lap72] M. Laplaza. Coherence for distributivity. *Lecture Notes in Mathematics*, 281:29–72, 1972.

[RB01] R. Raussendorf and H. J. Briegel. A one-way quantum computer. *Phys. Rev. Lett.*, 86(22):5188–5191, May 2001.

[Sel04] Peter Selinger. Towards a quantum programming language. *Mathematical. Structures in Comp. Sci.*, 14(4):527–586, 2004.

[Sel05] Peter Selinger. Dagger compact closed categories and completely positive maps. In Peter Selinger, editor, *Proceedings of the 3rd International Workshop on Quantum Programming Languages*, Electronic Notes in Theoretical Computer Science. Elsevier Science, 2005.

[Sho94] P Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings, 35th Annual Symposium on Foundations of Computer Science*. CA: IEEE Press, 1994.

[Tof80] Tommaso Toffoli. Reversible computing. In *ICALP*, pages 632–644, 1980.

[VAS06] Juliana Kaizer Vizzotto, Thorsten Altenkirch, and Amr Sabry. Structuring quantum effects: Superoperators as arrows. *Mathematical Structures in Computer Science*, 16(3), 2006. Also arXiv:quant-ph/0501151.