Quantum Computing Modeling in Scala

Constantin Gonciulea
Distinguished Engineer, JPMorgan Chase

Scale By the Bay, November 2018

Like many of you, over the years I have design systems that manage stateful computations for various use cases inside server, mobile and web applications. One of the most challenging for me was state management on the client side, in our online banking application that was redesigned a few years ago. But what I have found out is that common principles can be applied across various contexts.

Earlier this year I started looking at Quantum Computing. Even though I took Quantum Mechanics classes in college and graduate school, I realized that Quantum Computing is better understood separately from Physics.

Looking at it with a computing mindset, quantum state is not that different from classical or probabilistic state. In this presentation I will share a common abstraction that captures the similarities and differences in classical, probabilistic and quantum state.
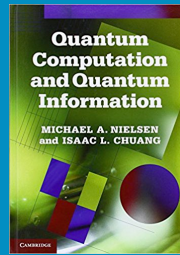
> **The postulates of quantum mechanics were derived after a long process of trial and (mostly) error, which involved a considerable amount of guessing and fumbling by the originators of the theory.**

This is a quote from the standard textbook on Quantum Computing, pointing out that after many years of trial and error, Quantum experts arrived at 4 postulates that capture the essence of Quantum Mechanics. These postulates are also a self-contained foundation for Quantum Computing.
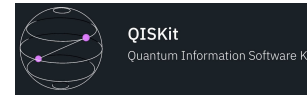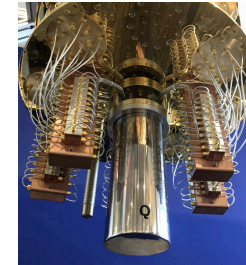
Why Scala?

Higher To Lower Abstraction
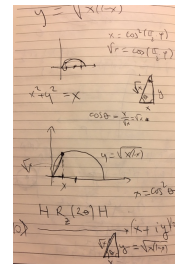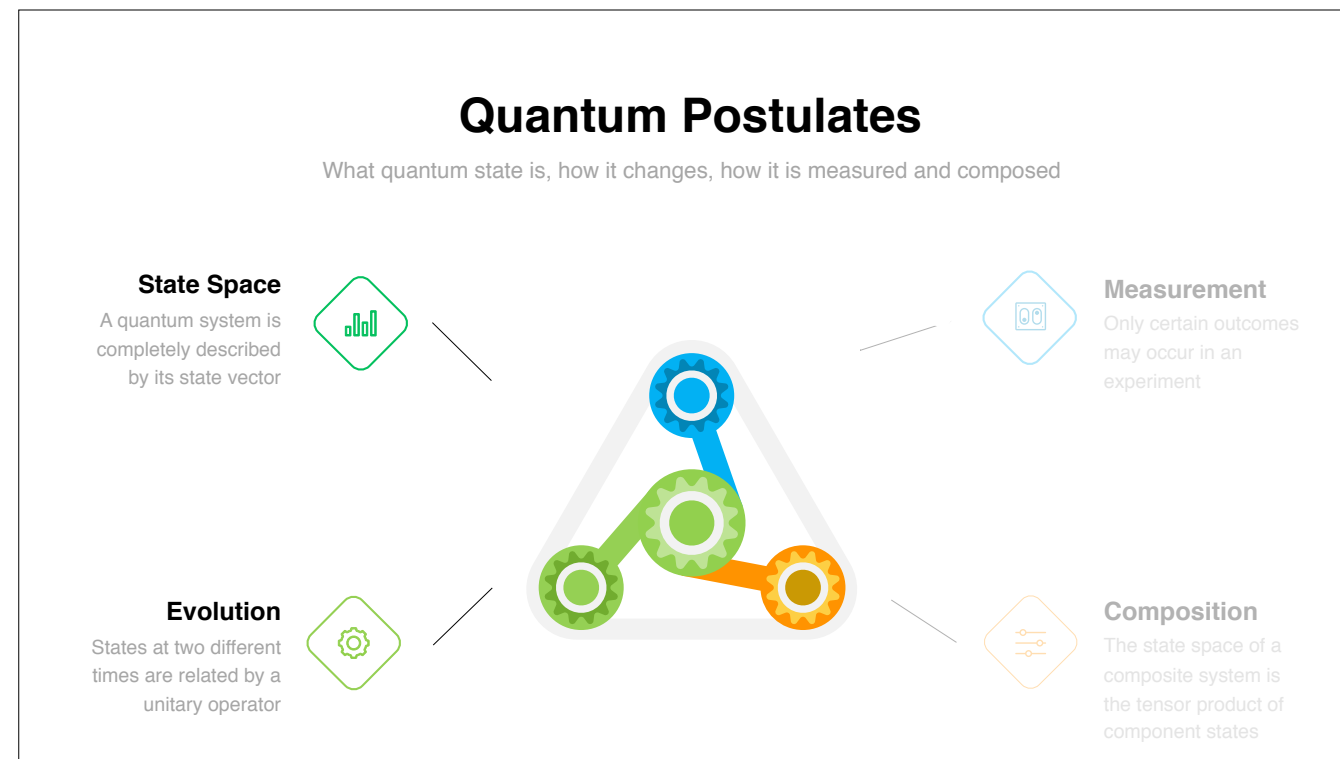

| Math | Scala Simulator | Python Simulator | Quantum Computer |

When designing quantum algorithms, you don't usually use a real quantum computer until late in the process. This has been my typical process during the last few months.

Quantum Postulates

What quantum state is, how it changes, how it is measured and composed

State Space

A quantum system is completely described by its state vector

Measurement

Only certain outcomes may occur in an experiment

Evolution

States at two different times are related by a unitary operator

Composition

The state space of a composite system is the tensor product of component states

We will start by looking at the first two postulates, which specify how quantum state is represented and how it evolves in time. More precisely, the state of a quantum system is modeled as a vector (collection of numbers), and state transformations have to keep the state valid, essentially by preserving an invariant.

Modeling state evolution is not specific to Quantum Computing, it is a core problem in any kind of computing. Since quantum principles are derived from the way Nature works, we can use the constraints that come with handling quantum state as a guide in designing a common abstraction for stateful computations. It should not be a surprise that the monadic approach aligns with the constraints of the postulates.

Monadic State Evolution

The system state is defined by labeled values (allocations)

State: `List[(B, V)]`   >>=   Change: `B => List[(B, V)]`   =   State: `List[(B, V)]`
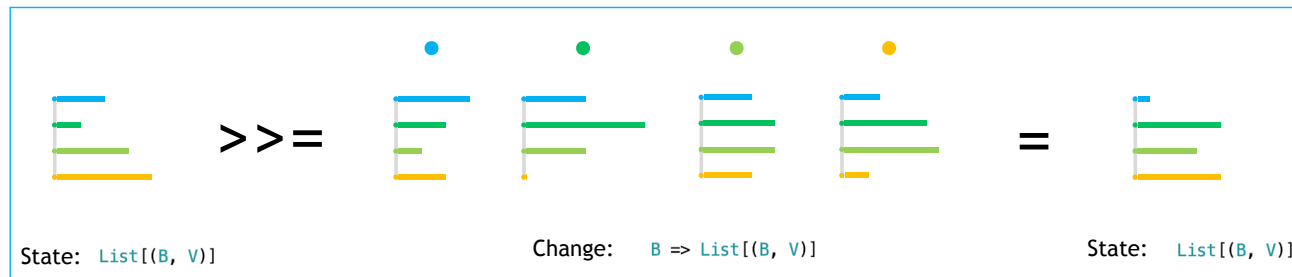
Examples: resource allocation, accounting systems, probabilistic systems, quantum systems

Compare with typical monads:

Container: `M[A]`   >>=   Change: `A => M[A]`   =   Container: `M[A]`

Let's look at a class of systems whose state is represented by a **collection of labeled values**, which we can call **allocations**. The labels are fixed, and we can think of them as buckets or bins.

Examples of such systems are common:
- accounting systems, where the buckets are accounts, and the corresponding values are balances
- investment portfolios, where buckets are investment instruments like stocks or bonds, and the corresponding values are the allocation percentages for those instruments
- probabilistic systems, where the buckets are possible outcomes (called hypotheses), and the corresponding values are the probabilities of those outcomes
- quantum systems are similar to probabilistic systems, where the buckets are possible outcomes, and the corresponding values are called amplitudes (we will talk about them later)

The state can be represented as a List of tuples for simplicity, but the MapLike data structure in the Scala language is a nice alternative. A state change is represented by a list of entries, one per bucket, and such an entry looks like the state itself. These entries are independent of the values in the buckets. They represent relative changes, like percentages, or transactions.

In the portfolio allocation scenario, a change specifies how each instrument is reallocated. In the case of an accounting system, a change specifies what payments are made from each account into other accounts.

State Representation and Evolution

Unified monadic approach to classical, probabilistic and quantum state

```scala
trait UState[+This <: UState[This, B, V], B, V] {
  protected val bins: List[(B, V)]
  protected val m: Monoid[V]

  protected val normalizeStateRule: List[(B, V)] => List[(B, V)] = identity

  protected val combineBinsRule: List[(B, V)] => List[(B, V)] = { bvs =>
    bvs.groupBy(_._1).toList.map {
      case (b, vs) => (b, vs.map(_._2).foldLeft(m.empty)(m.combine))
    }
  }
  protected val distributionRule: ((B, V), List[(B, V)]) => List[(B, V)]

  def create(bins: List[(B, V)]): This

  def normalize(): This = create(normalizeStateRule(bins))

  def flatMap(f: B => List[(B, V)]): This = {
    val updates: List[(B, V)] = bins.flatMap({ case (b, v) => distributionRule((b, v), f(b)) })
    create(normalizeStateRule(combineBinsRule(updates)))
  }

  def >>=(f: B => List[(B, V)]): This = flatMap(f)
}
```
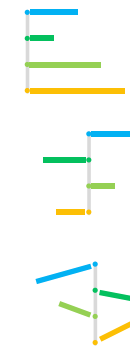
State: List[(B, V)]

Change: B => List[(B, V)]

Here is the state **representation** and **transformation** abstraction that applies to all the scenarios we mentioned. The state is captured in a collection of bin-value pairs. Typically the state is valid only if an invariant is maintained during evolution (e.g. the sum of values across bins is 1, or 0; for quantum state the values are vectors whose squared magnitudes have to add up to 1).

The flatMap method processes state changes and returns a new state. The state changes are relative, i.e. they are independent of the current bin values. Its implementation can be standardized if we have a specification of how to go from relative to absolute changes. This specification or rule is called distributionRule here. In the portfolio allocation case, that means going from percentages to balances that need to be combined.

In order to combine labeled values we need a combine operation with a neutral element, specified here by a monoid (e.g. addition or multiplication).

In some cases we need to normalize the state, in order to preserve an invariant, if the transformations do not enforce that invariant (that will be the case of probabilistic state, as we will see later).

Portfolio Balancing

Percentage based resource allocation

```scala
val bins: List[(String, Double)] = List("a" -> .2, "b" -> .1, "c" -> .3, "d" -> .4)

val m = RState[String](bins)
val changeA = List("a" -> .25, "b" -> .5, "c" -> .25)
val changeB = List("b" -> 1.0)
val changeC = List("c" -> 1.0)
val changeD = List("d" -> 1.0)

val state = m >>= Map("a" -> changeA, "b" -> changeB, "c" -> changeC, "d" -> changeD)

assert(state.bins.toSet == Set("a" -> .05, "b" -> .2, "c" -> .35, "d" -> .4))
```
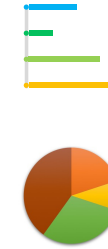
Implementation

```scala
case class RState[B](bins: List[(B, Double)]) extends UState[RState[B], B, Double] {
  val m = new Monoid[Double] {
    override val empty: Double = 0.0
    override val combine: (Double, Double) => Double = _ + _
  }

  override val distributionRule: ((B, Double), List[(B, Double)]) => List[(B, Double)] = {
    case ((b, v), cs) => cs.map { case (c, u) => (c, u * v) }
  }

  override def create(bins: List[(B, Double)]) = RState(bins)
}
```

Invariant: sum = 1

In the case of portfolio (re)balancing, the buckets are the investment instruments and the values are the corresponding allocation percentages. Of course, the percentages (in decimal form) have to add up to 1.

Consider the example of a portfolio with 4 instruments, with 20% of the total value in "a", 10% in "b", 30% in "c", and 40% in "d". And assume we want to reallocate "a" this way: 50% into "b", and 25% into "c". That means that 25% stays in "a". The expected result is.

Now let's look at the implementation. The combining operation is addition. The update rule takes an instrument "b" and distributes its value across all instruments: for each instrument "c" in "f(b)" we add the percentage of "b" specified in the change. This is equivalent to the linear algebra approach, where the change is interpreted as a matrix, and the values as a column vector.

Account Balances

State of a closed accounting system

```scala
val bins: List[(String, Double)] = List("a" -> 2, "b" -> 3, "c" -> 5, "d" -> -8, "e" -> -2)

val z = ZState[String](bins)
val changeA = List("a" -> -1.0, "b" -> 1.0)
val changeB = List("b" -> -2.0, "c" -> 1.0, "d" -> 1.0)

val state = z >>= Map("a" -> changeA, "b" -> changeB, "c" -> Nil, "d" -> Nil, "e" -> Nil)

assert(state.bins.toSet == Set("a" -> 1.0, "b" -> 2.0, "c" -> 6.0, "d" -> -7.0, "e" -> -2.0))
```
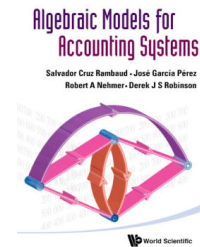
Implementation

```scala
case class ZState[B](bins: List[(B, Double)]) extends UState[ZState[B], B, Double] {
  val m = new Monoid[Double] {
    override val empty: Double = 0.0
    override val combine: (Double, Double) => Double = _ + _
  }
  override val distributionRule: ((B, Double), List[(B, Double)]) => List[(B, Double)] = {
    case ((b, v), cs) => List((b -> v)) ++ cs
  }

  override def create(bins: List[(B, Double)]) = ZState(bins)
}
```

Invariant: sum = 0

Let's quickly go through the example of an accounting system. The invariant for the state of such a system is that the sum of all account balances has to be zero. The book shown to the right presents an algebraic approach that is similar to what we are showing here, including this invariant.

A change specifies for each account what payments are made into other accounts.

The combining operation is addition. The update rule says that we need to combine the current balance for each account with the relative changes specified by the payments into that account
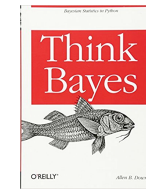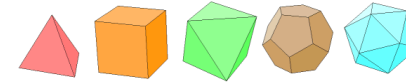
Probabilistic State

Each possible outcome is assigned a probability

Repeatedly rolling one of the 5 platonic solid dice yields the following sequence: 6, 6, 8, 7, 7, 5, 4.
Guess which die was used?

```
     Priors:
     4  0.2     ##########
     6  0.2     ##########
     8  0.2     ##########
     12 0.2     ##########
     20 0.2     ##########

     After a 6 is rolled:
     4  0.0
     6  0.3921 ###################
     8  0.2941 ##############
     12 0.1960 #########
     20 0.1176 #####

     After 6, 8, 7, 7, 5, 4 are rolled after the first 6:
     4  0.0
     6  0.0
     8  0.9432 ##############################################
     12 0.0552 ##
     20 0.0015
```

For the probabilistic state, let's go through an example first. Assume that we choose one of the 5 dice shaped as regular polyhedra, having 4, 6, 8, 12 and 20 faces, respectively. After rolling the chosen die a few times, we are trying to guess which die was chosen.

Before seeing the result of the first roll, we assign the same probability to each die. If the first roll is a 6, we know for sure that the die was not the tetrahedron (which has 4 faces). The probabilities of the other dice are updated, and intuitively this histogram shown here makes sense. The precise updated probabilities are obtained by multiplying the current probabilities by the corresponding likelihoods of the rolled number to be a face of the dice and then normalizing the state. We need to normalize because the likelihoods are not probabilities, so they do not add up to 1.

If we roll the chosen die more times we can improve our guess. For example, if we see the sequence shown here, we can rule out the die with 6 faces (the cube), because numbers higher than 6 are present in the sequence. The die with 8 faces is the most probable because a wider range of numbers would have been expected from having more faces.

Probabilistic State

Bayes Theorem

```scala
case class PState[B](bins: List[(B, Double)]) extends UState[PState[B], B, Double] {
  val m = new Monoid[Double] {
    override val empty: Double = 1.0
    override val combine: (Double, Double) => Double = _ * _
  }

  override val distributionRule: ((B, Double), List[(B, Double)]) => List[(B, Double)] = {
    case ((b, v), cs) => cs.map { case (c, u) => (c, u * v) }
    //case ((b, v), cs) => List((b -> v)) ++ cs  // both work
  }

  override val normalizeStateRule = { bvs: List[(B, Double)] =>
    val sum = bvs.map(_._2).foldLeft(0.0)(_ + _)
    if (sum == 1.0) bins else bvs.map {
      case (b, v) => (b, v / sum)

    }
  }

  override def create(bins: List[(B, Double)]) = PState(bins)
}
```

Invariant: normalized sum = 1

Change: data point likelihoods

```scala
val change = Map(
  "a" -> List("a" -> 0.2),
  "b" -> List("b" -> 0.7),
  "c" -> List("c" -> 0.0))
```

Let's look at the implementation. The combining operation is multiplication. As mentioned before, we need a normalization procedure, which just scales all values so they add up to 1.

A change triggered by a new data point is represented by the likelihoods of that data point being compatible with the possible outcomes. For a given outcome there is only one "reallocation", because outcomes are independent, only the likelihood corresponding to one outcome at a time is non-zero.

The update rule is the same as the one used in the portfolio balancing, and the same as a linear transformation. Each outcome probability is multiplied by the likelihood of the new data point assuming that particular outcome. Interestingly enough, the update rule used for an accounting system also works, because combining the current probability with the corresponding likelihood is done by multiplying them.
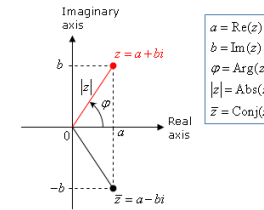
Quantum State

Complex numbers (2 -dimensional vectors) as values

```scala
case class QState[B](bins: List[(B, Complex)]) extends UState[QState[B], B, Complex] {
  val m = new Monoid[Complex] {
    override val empty: Complex = Complex.zero
    override val combine: (Complex, Complex) => Complex = Complex.plus
  }

  override val distributionRule: ((B, Complex), List[(B, Complex)]) => List[(B, Complex)] = {
    case ((b, v), cs) => cs.map { case (c, u) => (c, u * v) }
  }

  override def create(bins: List[(B, Complex)]) = QState(bins)
}
```


$a = \mathrm{Re}(z)$
$b = \mathrm{Im}(z)$
$\varphi = \mathrm{Arg}(z)$
$|z| = \mathrm{Abs}(z)$
$\bar{z} = \mathrm{Conj}(z)$

Invariant:
sum of squared magnitudes
= 1

The buckets of a quantum system state are the possible outcomes that can be measured. We will discuss measurement on the next slide, but as far as representation and evolution, quantum state is very similar to portfolio allocation, except the "allocations" are complex numbers, called amplitudes, whose squared magnitudes add up to 1.

The combining operation is addition for complex numbers or vectors. The update rule represents a linear transformation.
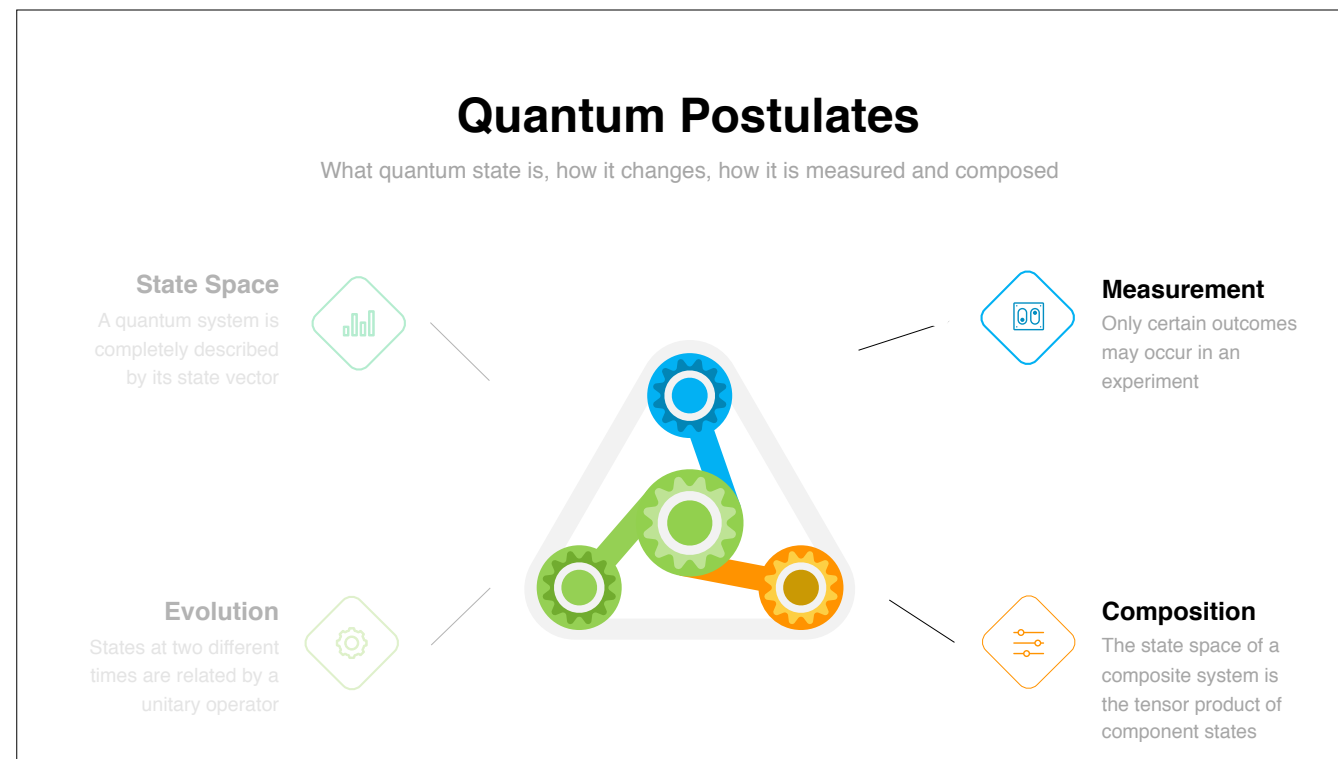
For the simple case of 2 buckets, we can think of a coin with an amplitude associated to each side. In general the coin is biased, as the magnitudes of the amplitudes of the two faces are not equal. Transforming the state changes the bias of the coin.

When such a quantum coin is tossed (the system state is measured), each side will materialize with a probability that is the squared magnitude of the amplitude corresponding to that face. Note that the sum of the probabilities is 1.

The X transformation swaps the two amplitudes. The Z transformation changes the sign of the second amplitude.
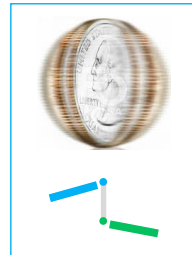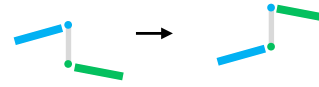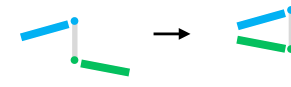
The Hadamard transformation, shown here. It can be applied to any valid state, but let's look at the case when we start with a biased quantum coin that always turns heads, so the amplitude of one side is 1. The new amplitude of each face is 1 over square root of 2, meaning that each side has a 50% probability of showing up. The quantum coin is in equal superposition.
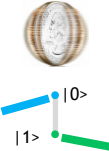
Quantum Postulates

What quantum state is, how it changes, how it is measured and composed

State Space
A quantum system is completely described by its state vector

Measurement
Only certain outcomes may occur in an experiment

Evolution
States at two different times are related by a unitary operator

Composition
The state space of a composite system is the tensor product of component states

The last two postulates of quantum computing address state measurement and state composition.

In classical computing the state of a system can be read or measured any time, and the measurement reflects the current state of the system. In a quantum system, the state is not accessible, and a measurement yields one of the possible outcomes, with a probability that is equal to the squared magnitude of that outcome. In addition, the new state of the system after the measurement is a state that corresponds to the outcome that was measured. This is called state collapsing.

The buckets in in a quantum system that is the composition of two quantum systems is a collection of pairs of buckets from the two systems. The Scala language's Tuple concept is all that is needed to represent composition of quantum states, but it is more practical to use indexed structures to simplify the representation.

Composition and Measurement

Qubits, superposition, entanglement

We can use quantum bits (essentially quantum coins previously described) as the units of composition.

We are showing here quantum systems consisting of 1 and 2 qubits.

The possible outputs of a system composed of n quantum bits are all 2^n binary words of length n. The quantum state consists of 2^n complex numbers, one per bucket/outcome. Repeated measurements will yield all binary words, each of them with a probability equal to the squared magnitude of its corresponding amplitude.
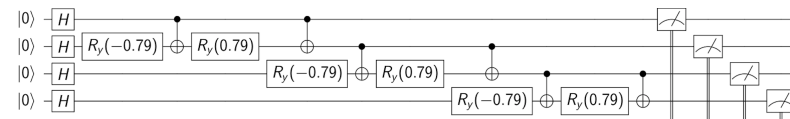
Calculating Fibonacci Numbers

Counting binary words with no consecutive ones

```
def fib(n: Int): QState = {
  var state = pure(Word.fromInt(0, n))
  for (i <- 0 until n) state = state >>= wire(i, H)
  for (i <- 0 until n - 1)  state = state >>= controlled(i, i + 1, ZERO)
  state
}
```

Circuit implementation:

```
for (i <- 0 until n - 1)  state = state >>= wire(i + 1, Ry(-math.Pi/4)) >>=
  controlled(i, i + 1, X) >>= wire(i + 1, Ry(math.Pi/4)) >>= controlled(i, i + 1, X)
```


F(1) = 2	
F(2) = 3	
F(3) = 5	
F(4) = 8	
F(5) = 13	
F(6) = 21	
F(7) = 34	
F(8) = 55	
F(9) = 89	
F(10) = 144	
F(11) = 233	
F(12) = 377	
F(13) = 610	
F(14) = 987	
F(15) = 1597	

Controlled transformations are applied only to states whose labels are satisfying a condition. The most important class of controlled transformations are the ones where the condition checks that the outcome contains 1 in one of the positions.

Let's build a quantum system that has only outcomes that are binary words of length n without consecutive ones. We first put every component quantum bit in equal superposition, so both 0 and 1 outcomes are possible and have equal probability of being measured. Then we add the constrain that if an outcome has 1 in a position, then it has to be followed by 0.

Running the system multiple times and counting the number of different possible outcomes allows to calculate the number of binary words of length n without consecutive ones. This is a Fibonacci number.

Implementing the computation in a real computer though requires breaking down state transformations into elementary operations that are possible on the physical device.

Thank You

Code

https://github.com/logicalguess/quantum-scale

Credits
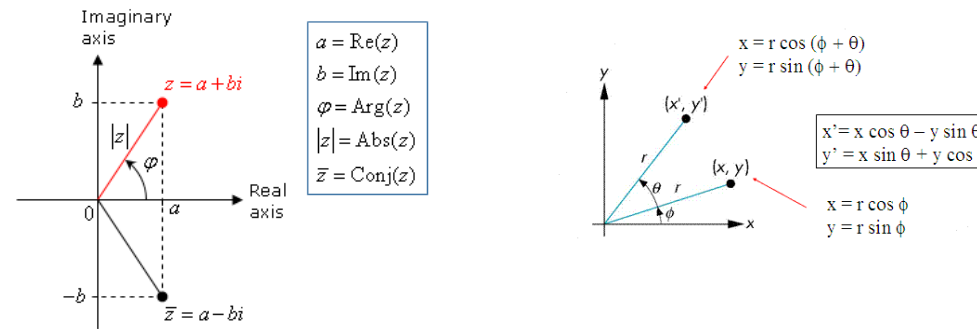
https://github.com/jliszka/quantum-probability-monad
https://sigfpe.wordpress.com/2007/03/04/monads-vector-spaces-and-quantum-mechanics-pt-ii/

Appendix

Complex Numbers

Complex numbers (2 -dimensional vectors) as values

http://www.thefouriertransform.com/math/complexmath.php

I added this slide in order to have a quick refresher on complex numbers, the type of values used in quantum systems. I will not go into all the details, but we can refer back to this slide when needed. A complex number is a pair of real numbers, that can be viewed as a two dimensional vector, or as an arrow (Richard Feynman's favorite term). The magnitude or length of a complex number is derived from the Pythagorean Theorem as the square root of the sum of the squares of its components.

Vectors can be added, multiplied and rotated.

Elementary Gates

```scala
"X" should "swap the amplitudes of |0> and |1>" in forAll { s: QState =>
  val t: QState = X(s)

  assert(t(S0) == s(S1))
  assert(t(S1) == s(S0))
}

"Y" should "swap the amplitudes of |0> and |1>, multiply each amplitude by i, and negate the amplitude of |1>" in forAll { s: QState =>
  val t: QState = Y(s)

  assert(t(S0) == - s(S1) * Complex.i)
  assert(t(S1) == s(S0) * Complex.i)
}

"Z" should "negate the amplitude of |1>, leaving the amplitude of |0> unchanged" in forAll { s: QState =>
  val t: QState = Z(s)

  assert(t(S0) == s(S0))
  assert(t(S1) == -s(S1))
}
```

The circuit computing model is a popular implementation of quantum computations, which happens to be used by IBM. In a real quantum computer, only a small number of transformations (called gates) is available, but any transformation can be build from them.

This is the equivalent of the system that outputs binary strings without consecutive ones. It uses a number of standard gates (H, Z, X, Ry).

Show and run code (Scala and Qiskit)

Rotations

Rotations around the Y axis

```scala
"Ry(theta)" should "mix the amplitudes of |0> and |1> (like vector rotation)" in
forAll { ts: (Double, QState) =>
  val theta = ts._1
  val state = ts._2

  val y: QState = Ry(theta)(state)

  // same formula as 2-dimensional vector rotation (but with half angle)
  val t0 = state(S0) * math.cos(theta/2) - state(S1) * math.sin(theta/2)
  val t1 = state(S0) * math.sin(theta/2) + state(S1) * math.cos(theta/2)

  assert(y(S0) == t0)
  assert(y(S1) == t1)
}
```

```scala
"Ry(pi/2)" should "equal HZ" in forAll { state: QState =>

  val y: QState = state >>= Ry(math.Pi/2)
  val h: QState = state >>= Z >>= H

  assert(y(S0).toString == h(S0).toString)
  assert(y(S1).toString == h(S1).toString)
}
```
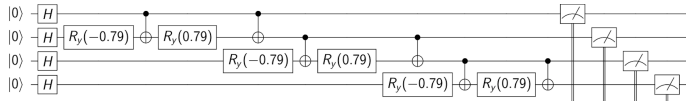
```scala
"Ry(theta)" should "be a weighted average of I and Ry(pi)" in
forAll { ts: (Double, QState) =>
  val theta = ts._1
  val state = ts._2

  val A: Gate = I * math.cos(theta / 2) + Ry(math.Pi) *
math.sin(theta / 2)

  val z: QState = Ry(theta)(state)
  val a: QState = A(state)

  assert(z(S0).toString == a(S0).toString)
  assert(z(S1).toString == a(S1).toString)
}
```


The circuit computing model is a popular implementation of quantum computations, which happens to be used by IBM. In a real quantum computer, only a small number of transformations (called gates) is available, but any transformation can be build from them.

This is the equivalent of the system that outputs binary strings without consecutive ones. It uses a number of standard gates (H, Z, X, Ry).

Show and run code (Scala and Qiskit)