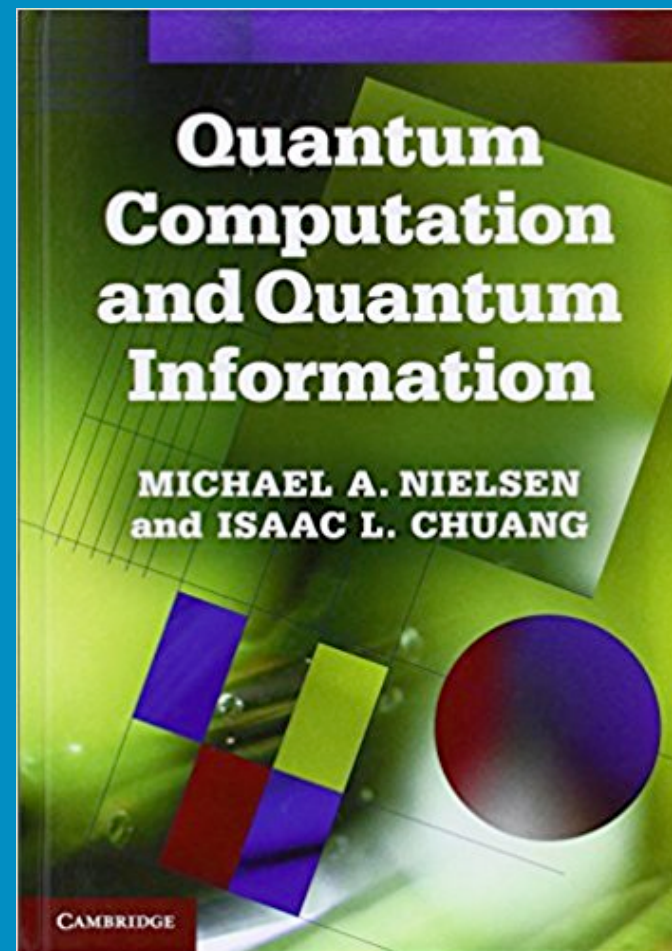




Quantum Computing Modeling in Scala

Constantin Gonciulea
Distinguished Engineer, JPMorgan Chase



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.

Quantum Postulates

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

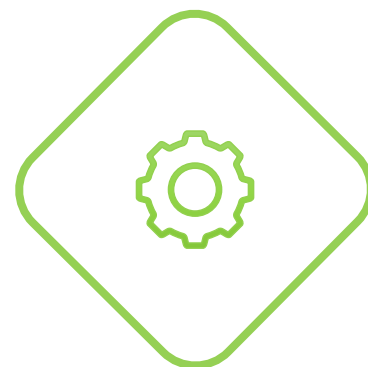
State Space

A quantum system is completely described by its state vector



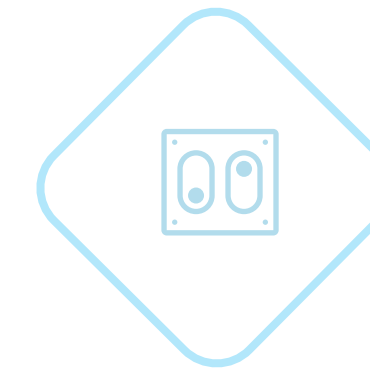
Evolution

States at two different times are related by a unitary operator



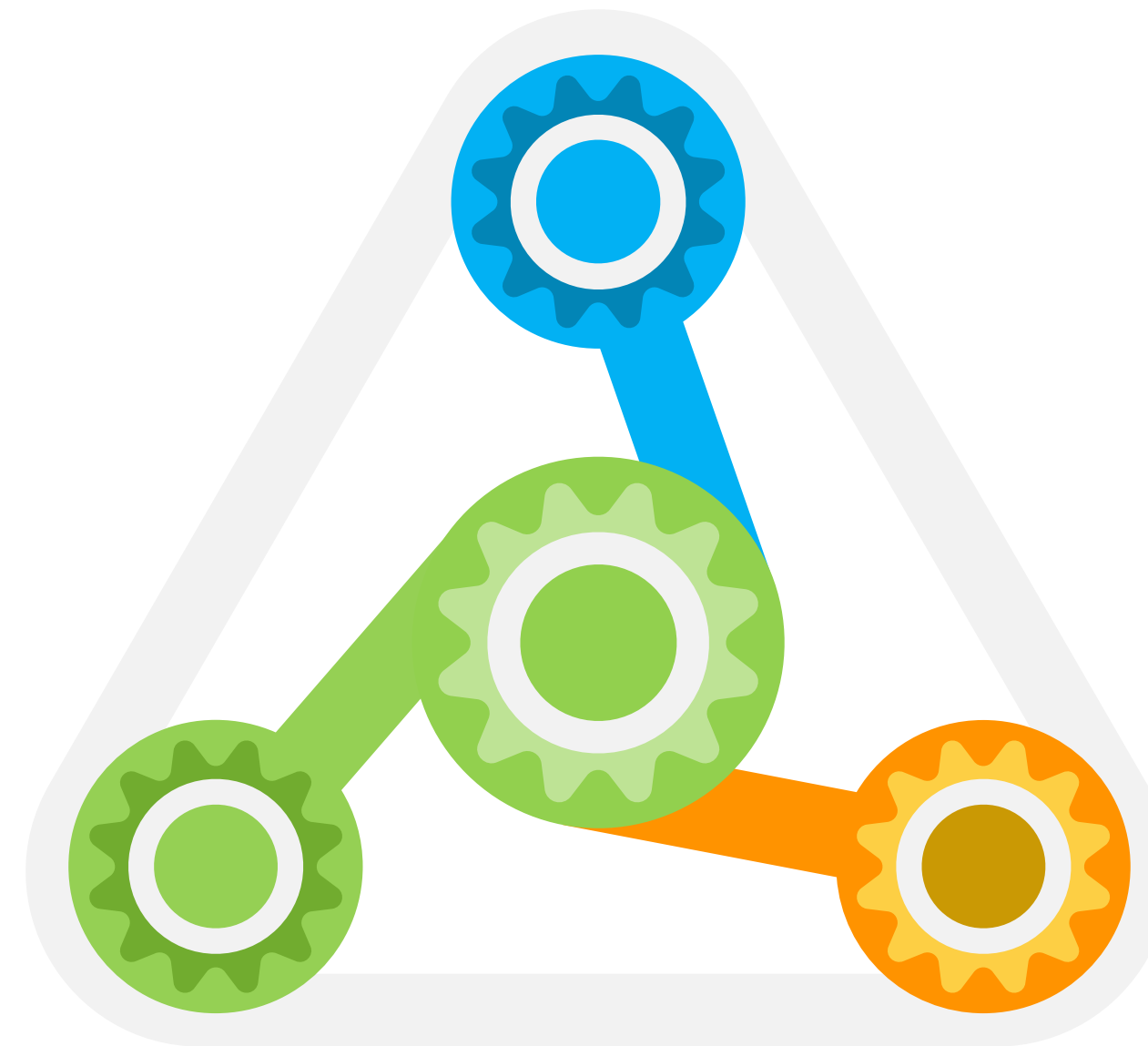
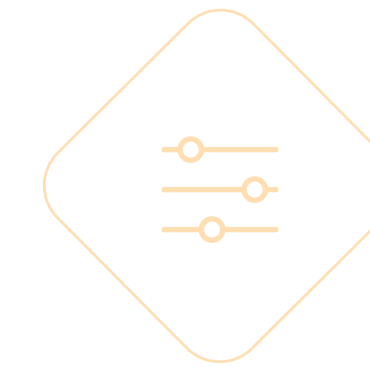
Measurement

Only certain outcomes may occur in an experiment



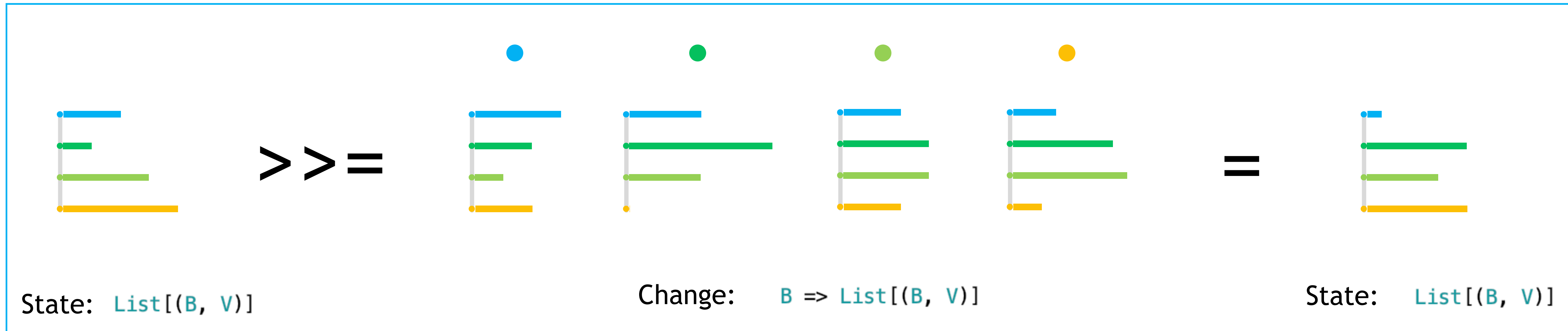
Composition

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



Monadic State Evolution

The system state is defined by labeled values (allocations)



Examples: resource allocation, accounting systems, probabilistic systems, quantum systems

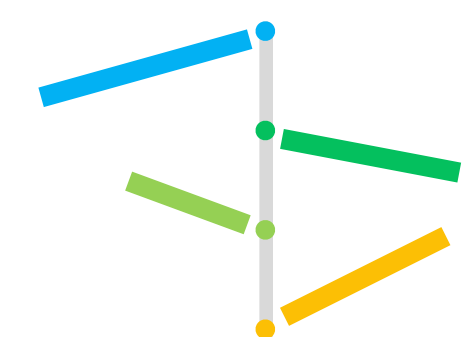
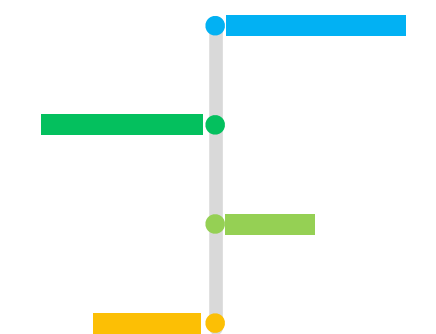
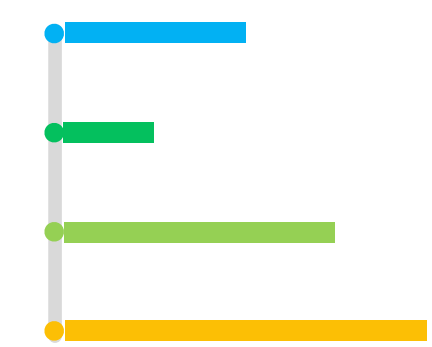
Compare with typical monads:

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

State Representation and Evolution

Unified monadic approach to classical, probabilistic and quantum state

```
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)] = { bs =>  
    bs.groupBy(_._1).toList.map {  
      case (b, vs) => (b, vs.map(_._2).foldLeft(m.empty)(m.combine))  
    }  
  }  
  
  protected val updateStateRule: ((B, V), B => 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 bv => updateStateRule(bv, f) })  
    create(normalizeStateRule(combineBinsRule(updates)))  
  }  
  
  def >>=(f: B => List[(B, V)]): This = flatMap(f)  
}
```



State: `List[(B, V)]`

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

Portfolio Balancing

Percentage based resource allocation

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

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

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

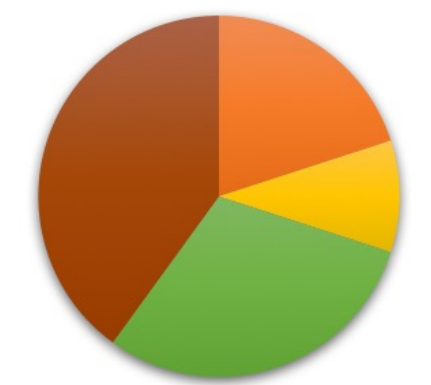
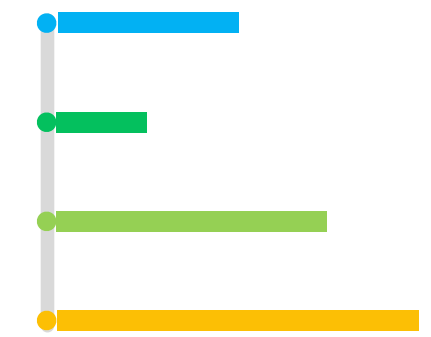
assert(state.bins.toSet == Set("a" -> .05, "b" -> .2, "c" -> .35, "d" -> .4))
```

Implementation

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

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

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



Invariant: sum = 1

Account Balances

State of a closed accounting system

```
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)

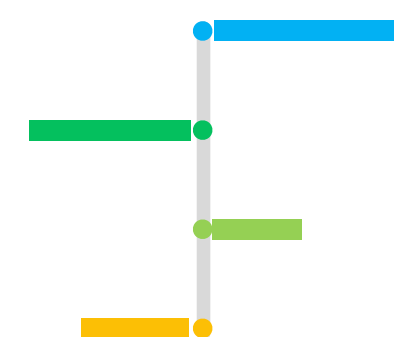
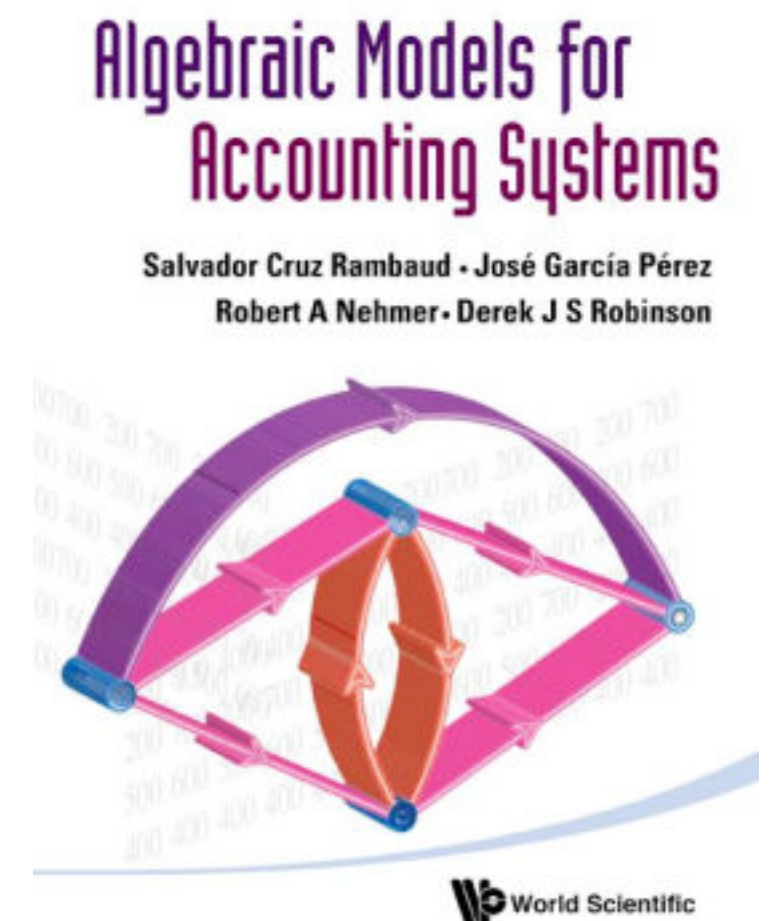
val state = z >=> Map("a" -> changeA, "b" -> changeB, "c" -> Nil, "d" -> Nil, "e" -> Nil)

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

Implementation

```
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 updateStateRule: ((B, Double), B => List[(B, Double)]) => List[(B, Double)] = {
    case ((b, v), f) => List((b -> v)) ++ f(b)
  }

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








Invariant: sum = 0

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:

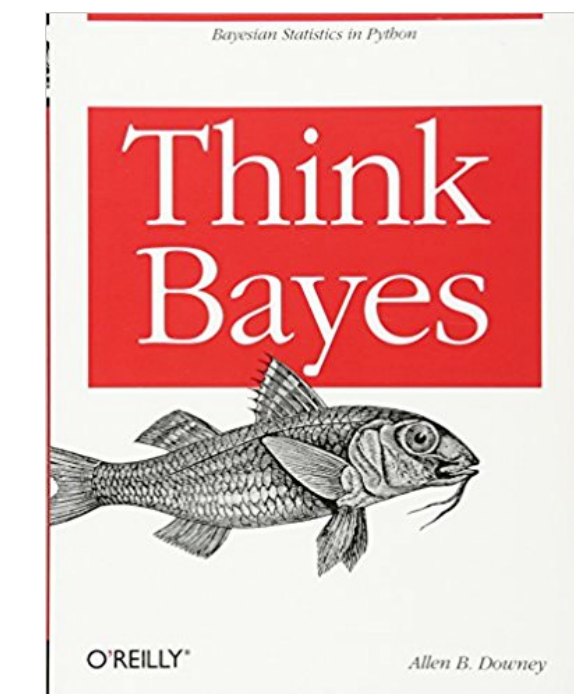
	4	0.2	#####
	6	0.2	#####
	8	0.2	#####
	12	0.2	#####
	20	0.2	#####

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:

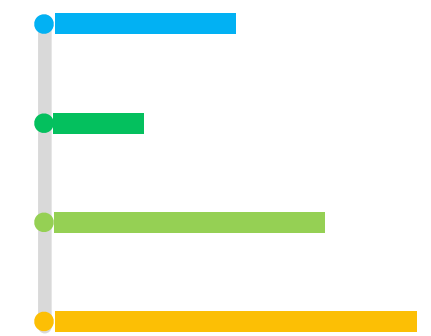
	4	0.0	
	6	0.0	
	8	0.9432	#####
	12	0.0552	##
	20	0.0015	



Probabilistic State

Bayes Theorem

```
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 updateStateRule: ((B, Double), B => List[(B, Double)]) => List[(B, Double)] = {  
    case ((b, v), f) => f(b).map { case (c, u) => (c, u * v) }  
    //case ((b, v), f) => List((b -> v)) ++ f(b) // both work  
  }  
  
  override val normalizeStateRule = { bs: List[(B, Double)] =>  
    val sum = bs.map(_._2).foldLeft(0.0)(_ + _)  
    if (sum == 1.0) bins else bs.map {  
      case (b, v) => (b, v / sum)  
    }  
  }  
  
  override def create(bins: List[(B, Double)]) = PState(bins)  
}
```



Invariant: normalized sum = 1

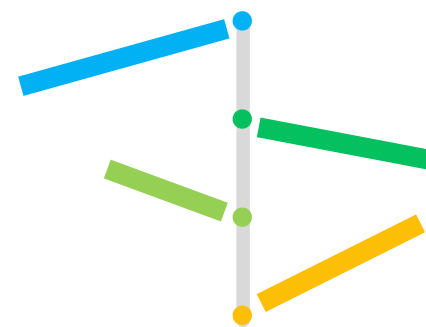
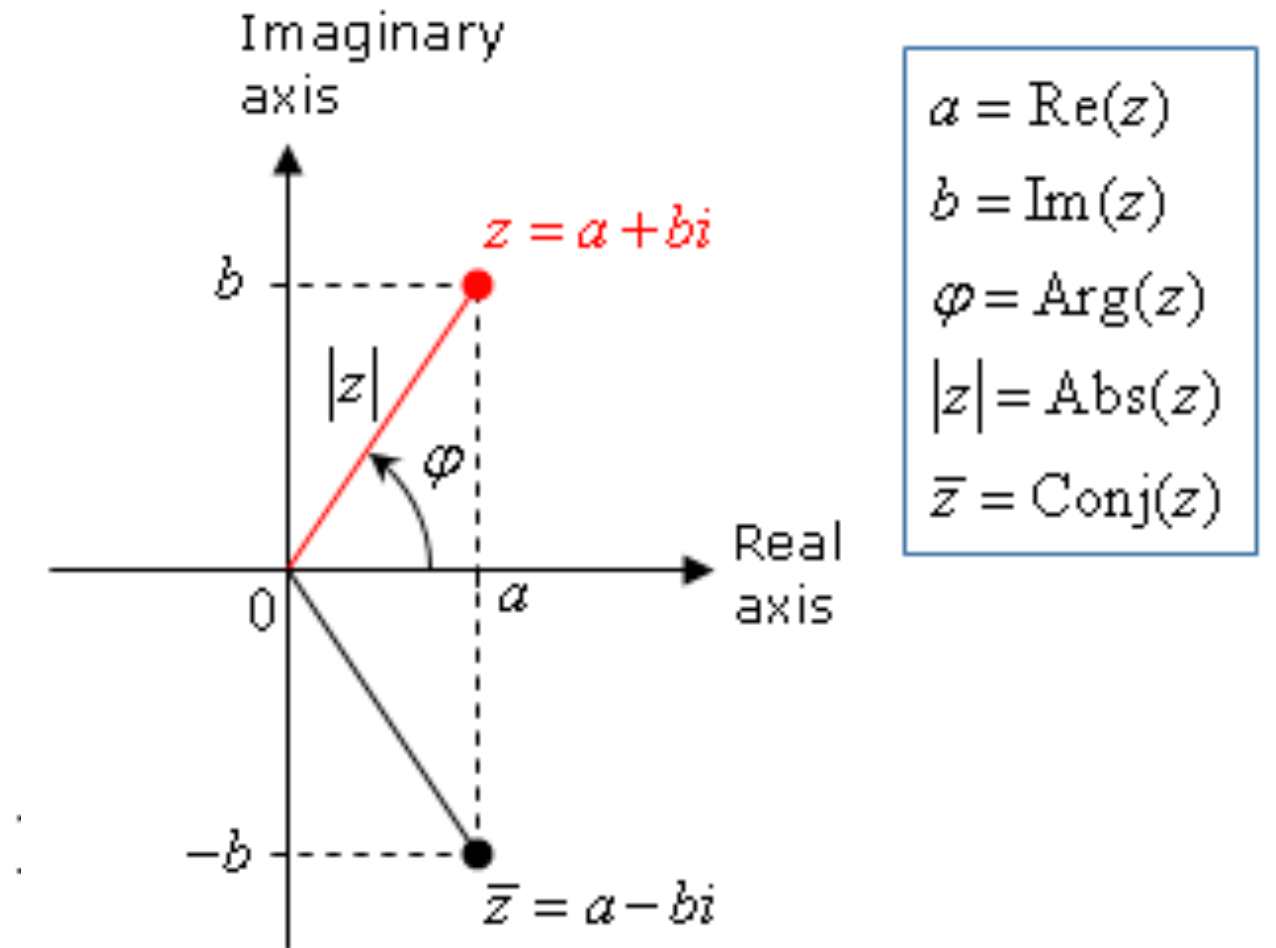
Change: data point likelihoods

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

Quantum State

Complex numbers (2 -dimensional vectors) as values

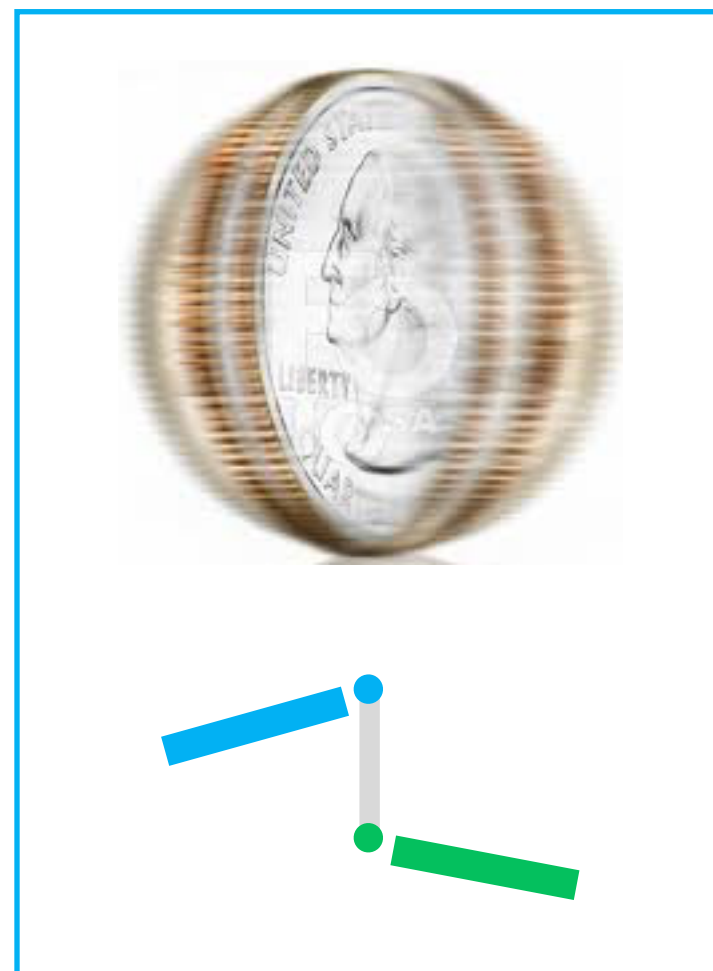
```
case class QState[B](bins: List[(B, Complex)]) extends UState[QState[B], B, Complex] {  
  val m = new Monoid[Complex] {  
    override val empty: Complex = Complex.zero  
    override val combine: (Complex, Complex) => Complex = Complex.plus  
  }  
  
  override val updateStateRule: ((B, Complex), B => List[(B, Complex)]) => List[(B, Complex)]  
    case ((b, v), f) => f(b).map { case (c, u) => (c, u * v) }  
  
  override def create(bins: List[(B, Complex)]) = QState(bins)  
}
```



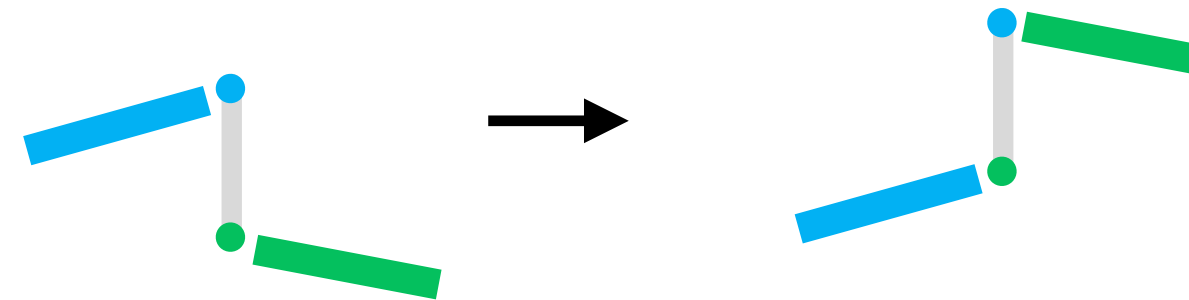
Invariant:
sum of squared magnitudes
= 1

Quantum State Transformations

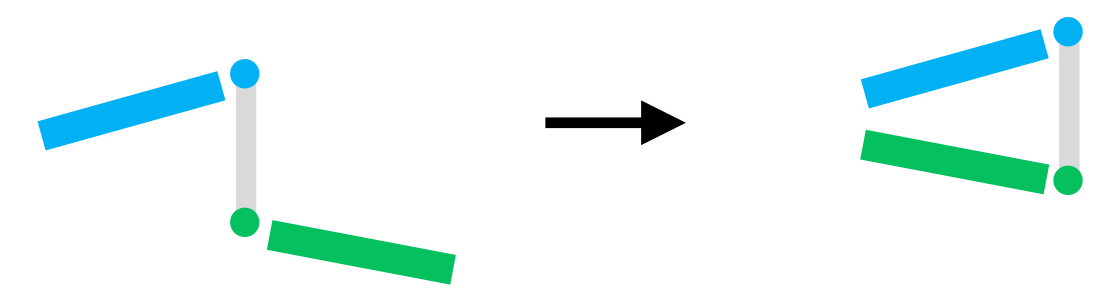
Standard single qubit transformations



X transformation

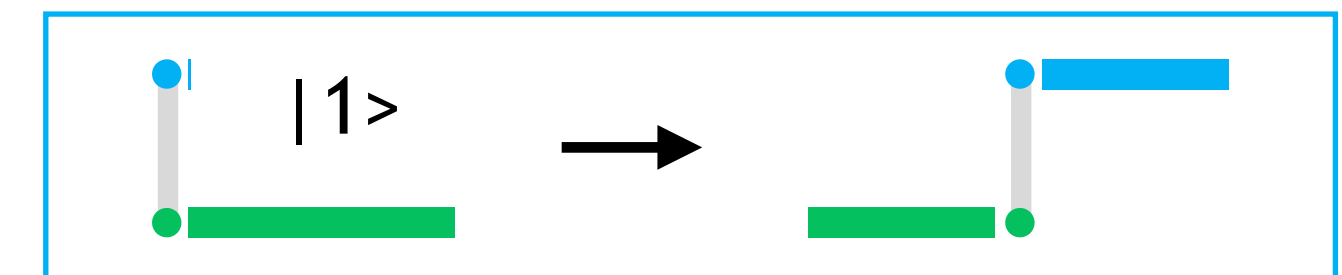
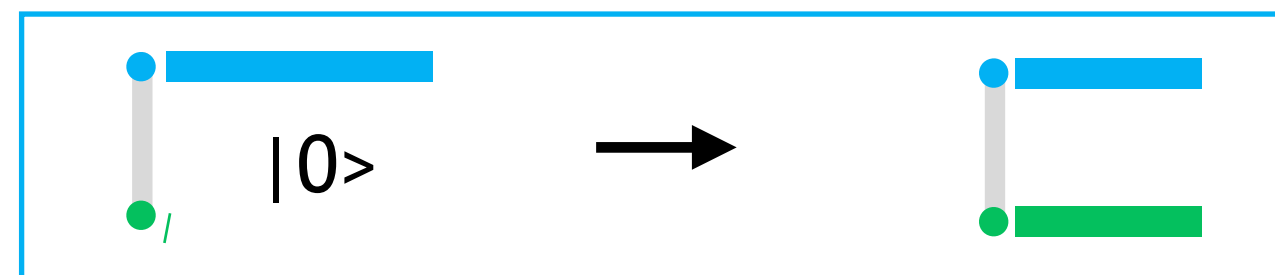


Z transformation



Hadamard transformation

```
val sq = toComplex(1 / math.sqrt(2))
val H = Map(
  "|0>" -> List("|0>" -> sq, "|1>" -> sq),
  "|1>" -> List("|0>" -> sq, "|1>" -> -sq)
)
```



Quantum Postulates

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

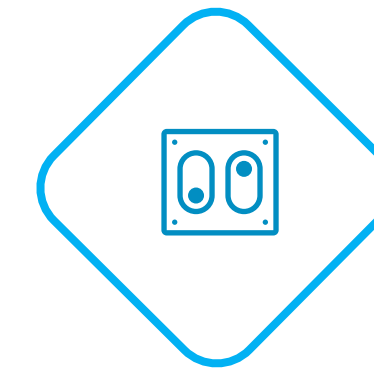
State Space

A quantum system is completely described by its state vector



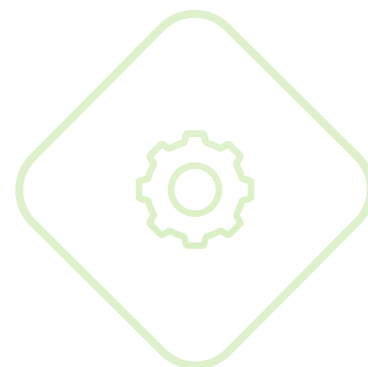
Measurement

Only certain outcomes may occur in an experiment



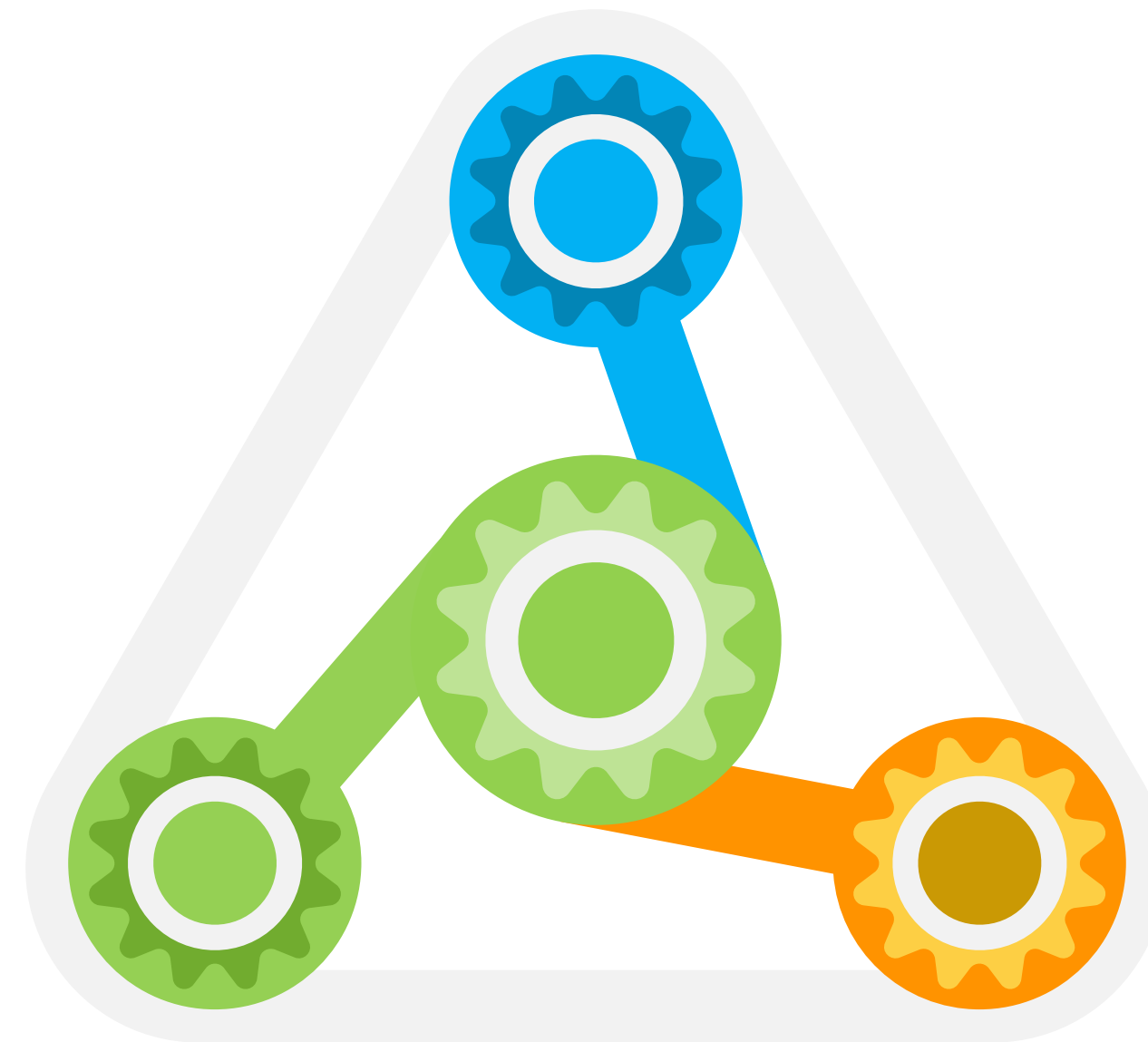
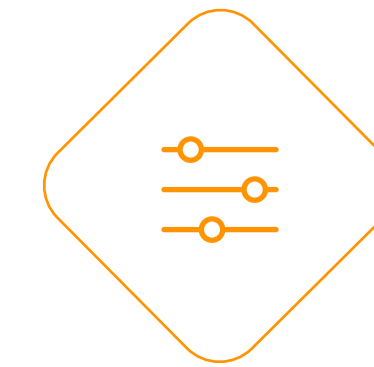
Evolution

States at two different times are related by a unitary operator




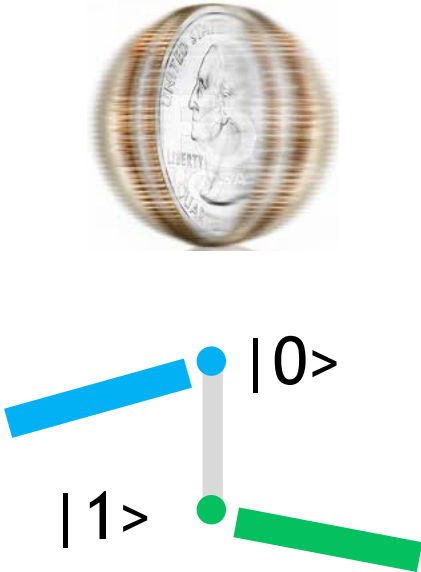



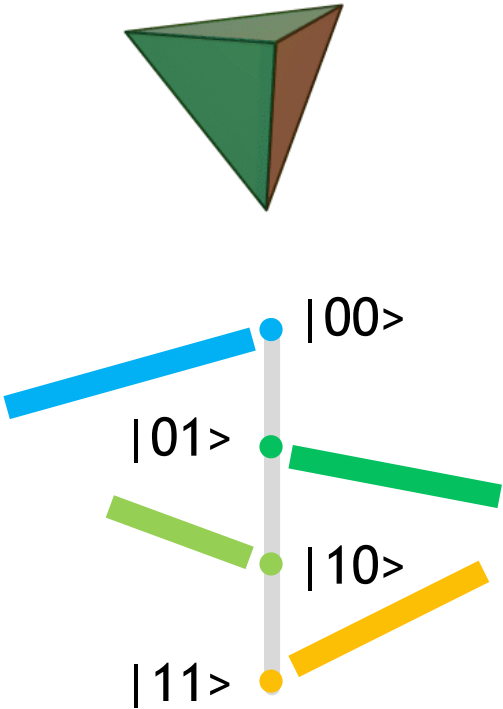








Composition

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



Composition and Measurement

Qubits, superposition, entanglement

Qubits	Quantum State	Measurement Outcomes
	One amplitude for each possible outcome	The probability of an outcome is the squared magnitude of its associated amplitude
		<div><div>• $0\rangle$</div><div></div><div>• $1\rangle$</div><div></div></div>
		<div><div>• $00\rangle$</div><div></div><div>• $01\rangle$</div><div></div><div>• $10\rangle$</div><div></div><div>• $11\rangle$</div><div></div></div>

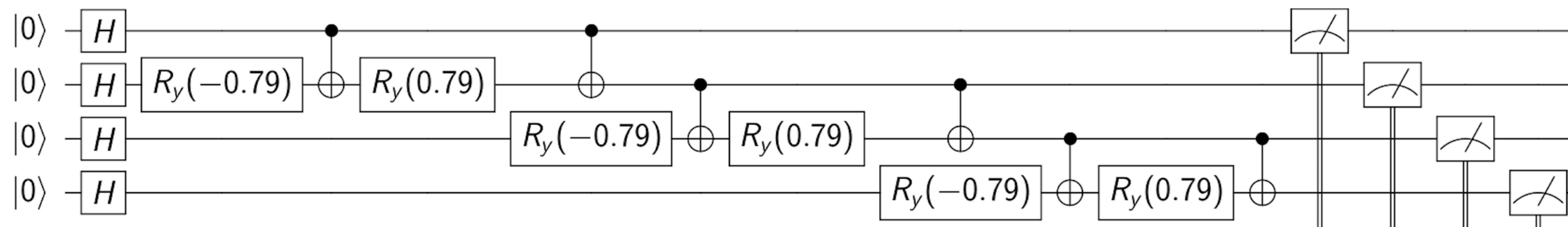
Calculating Fibonacci Numbers

Counting binary words with no consecutive ones

```
def fib(n: Int): QState[Word] = {  
  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

Thank You

Credits

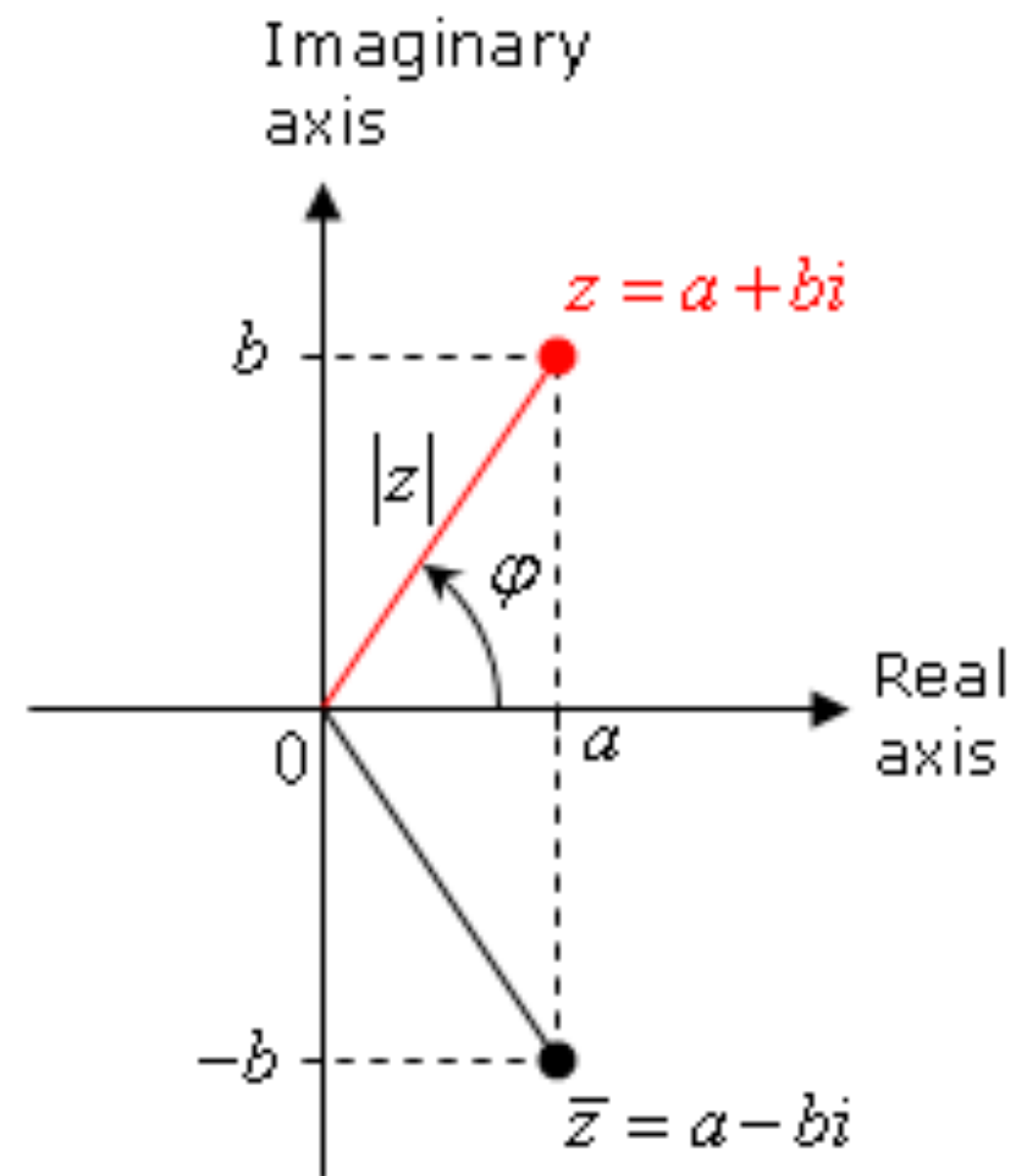
<https://github.com/jliszka/quantum-probability-monad>

<https://sigfpe.wordpress.com/2007/03/04/monads-vector-spaces-and-quantum-mechanics-pt-ii/>

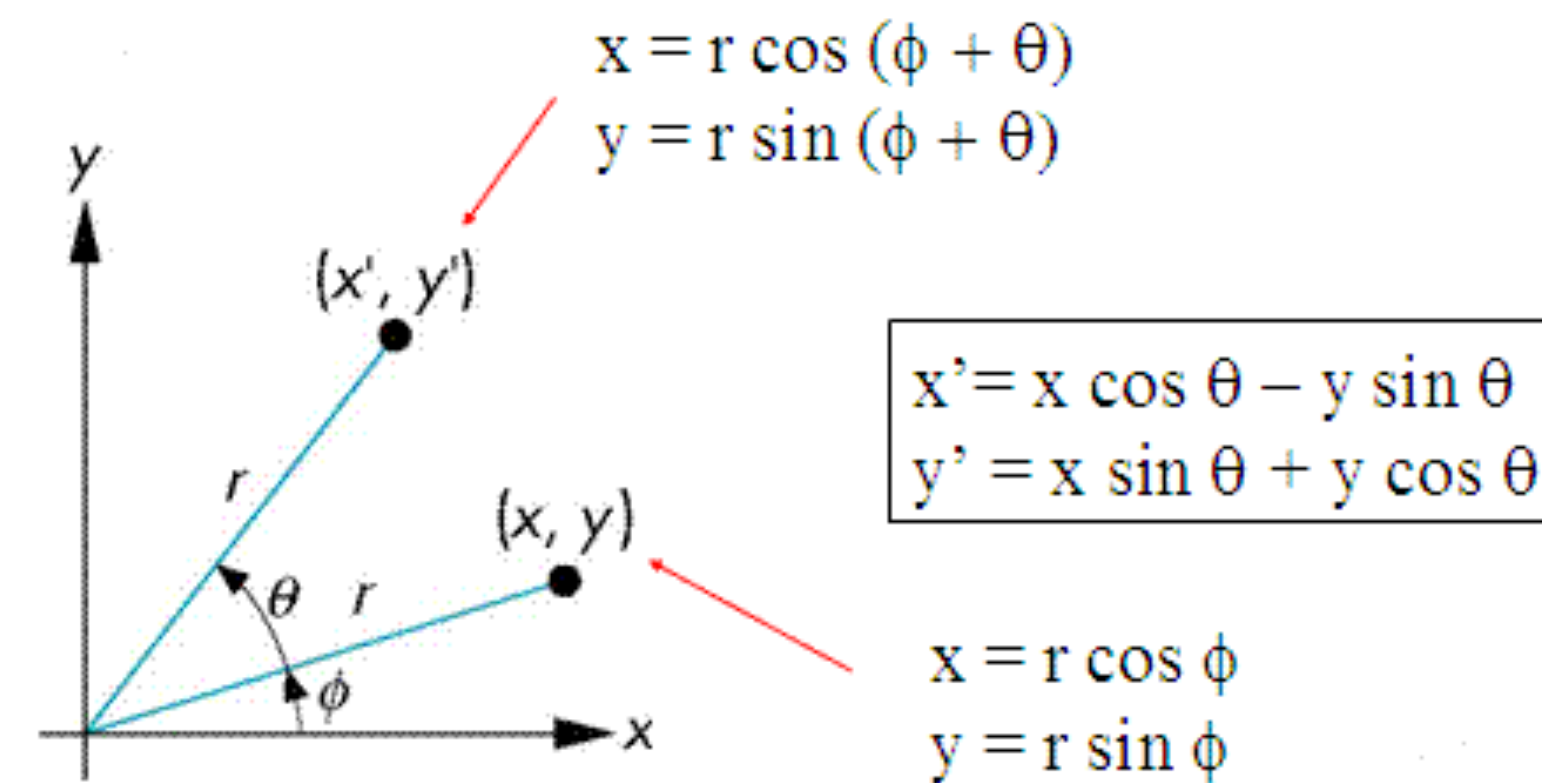
Appendix

Complex Numbers

Complex numbers (2 -dimensional vectors) as values



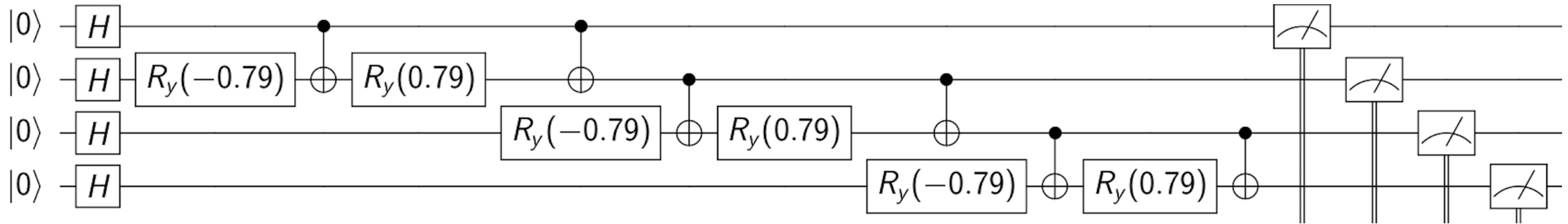
$$\begin{aligned} a &= \operatorname{Re}(z) \\ b &= \operatorname{Im}(z) \\ \varphi &= \operatorname{Arg}(z) \\ |z| &= \operatorname{Abs}(z) \\ \bar{z} &= \operatorname{Conj}(z) \end{aligned}$$



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

Circuits and Gates

Composing and measuring qbits



"**Ry(pi/2)Z**" should **"equal H"** in `forall { state: QState[Std] =>`

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

assert(y(S0).toString == h(S0).toString)
assert(y(S1).toString == h(S1).toString)
}
```

"**Ry(theta)**" should **"mix the amplitudes of $|0\rangle$ and $|1\rangle$ (like vector rotation)"**

```
val theta = ts._1
val state = ts._2

val y: QState[Std] = 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)
```