charles river analytics

Democratizing Machine Learning and Artificial Intelligence: Probabilistic Programming with Scala

Brian Ruttenberg, PhD
Charles River Analytics
bruttenberg@cra.com

Goals of This Talk

- Introduce basic modeling concepts in Machine Learning and Artificial Intelligence
- Detail some recent approaches and limitations in using these concepts to model real world problems
- Demonstrate how the Scala language helps Charles River Analytics apply our Machine Learning and Artificial Intelligence expertise to solve these problems

Outline

- Quick introduction to probabilistic models in Artificial Intelligence and Machine Learning
- Introduction to probabilistic programming
- Introduction to Figaro
  - Features, algorithms, examples and integration with Scala
  - Goals of the language
  - Many examples
- Future work & availability

What Do I Mean By Probabilistic Model?

- Let’s say I pick a person at random here
- There is some chance that this person is student

- This person may also be a programmer

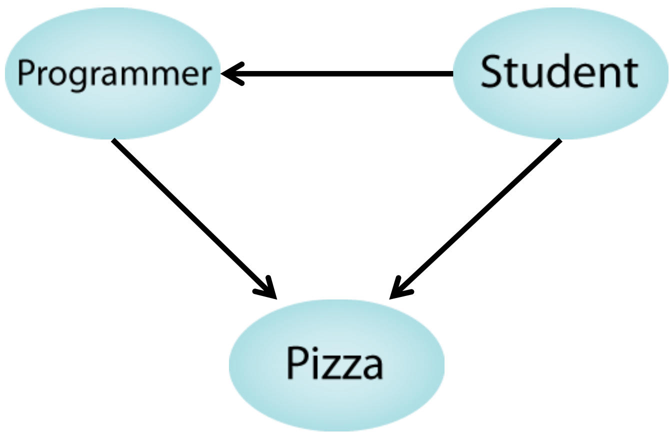
- This person may also be eating pizza

- Now what if someone asks me “is this person a student”, and I just see them eating pizza, what do I tell them?

Build a Probabilistic Model!

- We can build a model of this “world” using probability theory
- How do we do that?

- Start with Pizza
- What makes someone eat pizza?
  - If they’re a student, they probably eat pizza
  - But if they are a programmer, they probably eat pizza too
  - Represent these influences by a directed arrow
- But hold on!
  - This is a Scala meetup
  - If someone is a student, they are probably a programmer as well
  - So there is a dependency between the state of student and programmer

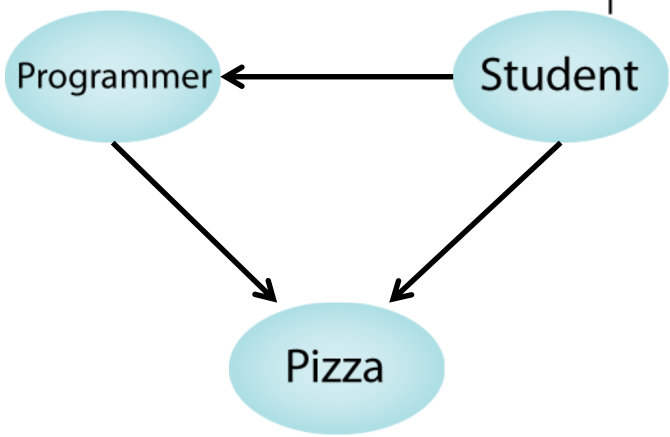


Adding Numbers

- So we've constructed a figure of the *dependencies* in our model
- But we need to add some numbers to the model in order to be useful

Student	Programmer T	Programmer F
F	0.3	0.7
T	0.8	0.2

Student T	Student F
0.4	0.6

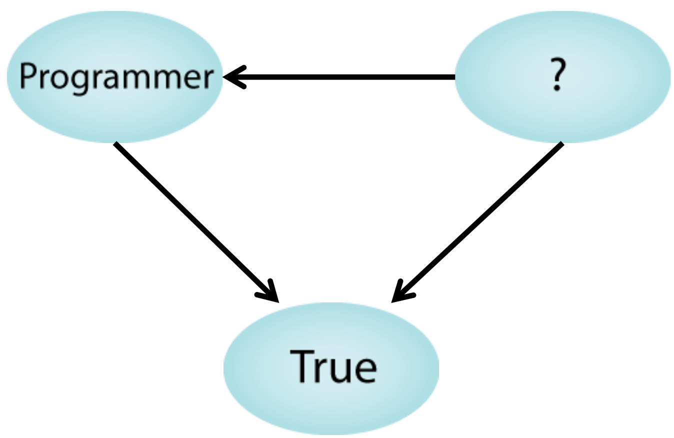


Prog	Student	Pizza T	Pizza F
F	F	0.1	0.9
F	T	0.7	0.3
T	F	0.6	0.4
T	T	0.99	0.01

- Can do this through conditional probability tables
  - Ie, what affects each variable state?
- Student depends on nothing (in our model)
- Programmer depends on student status
- Eating pizza depends on both

Answering the Question

- Someone is eating pizza, what is the probability they are a student?
- We can *infer* or *reason* about the probability of a variable (student) given some evidence (they are eating pizza)
  - “reverse” the arrows in the model
  - Compute probability using mathematics of conditional probability distributions



Answering the Question, Cont

- In theory, this is quite simple to answer
  - Encode the probabilities of each state in some programming language
  - Randomly generate states of the model by running the program
  - Record the number of times “Student” is true, divide by total states generated

Answering the Question, Cont

- How would the model look in Scala?

```
import scala.util._

def buildModel(iters: Int): Int = {
  if (iters == 0)
    0
  else {
    val prev: Int = buildModel(iters-1)
    val student: Boolean = if (Random.nextDouble() < 0.4) true else false
    val prog: Boolean = student match {
      case true => if (Random.nextDouble() < 0.8) true else false
      case false => if (Random.nextDouble() < 0.3) true else false
    }
    val pizza: Boolean = (prog, student) match {
      case (false, false) => if (Random.nextDouble() < 0.1) true else false
      case (false, true)  => if (Random.nextDouble() < 0.7) true else false
      case (true, false)  => if (Random.nextDouble() < 0.6) true else false
      case (true, true)   => if (Random.nextDouble() < 0.99) true else false
    }
    if (pizza) prev+1 else prev
  }
}

val probPizza = buildModel(100)/100
```

Doesn’t Seem So Bad…

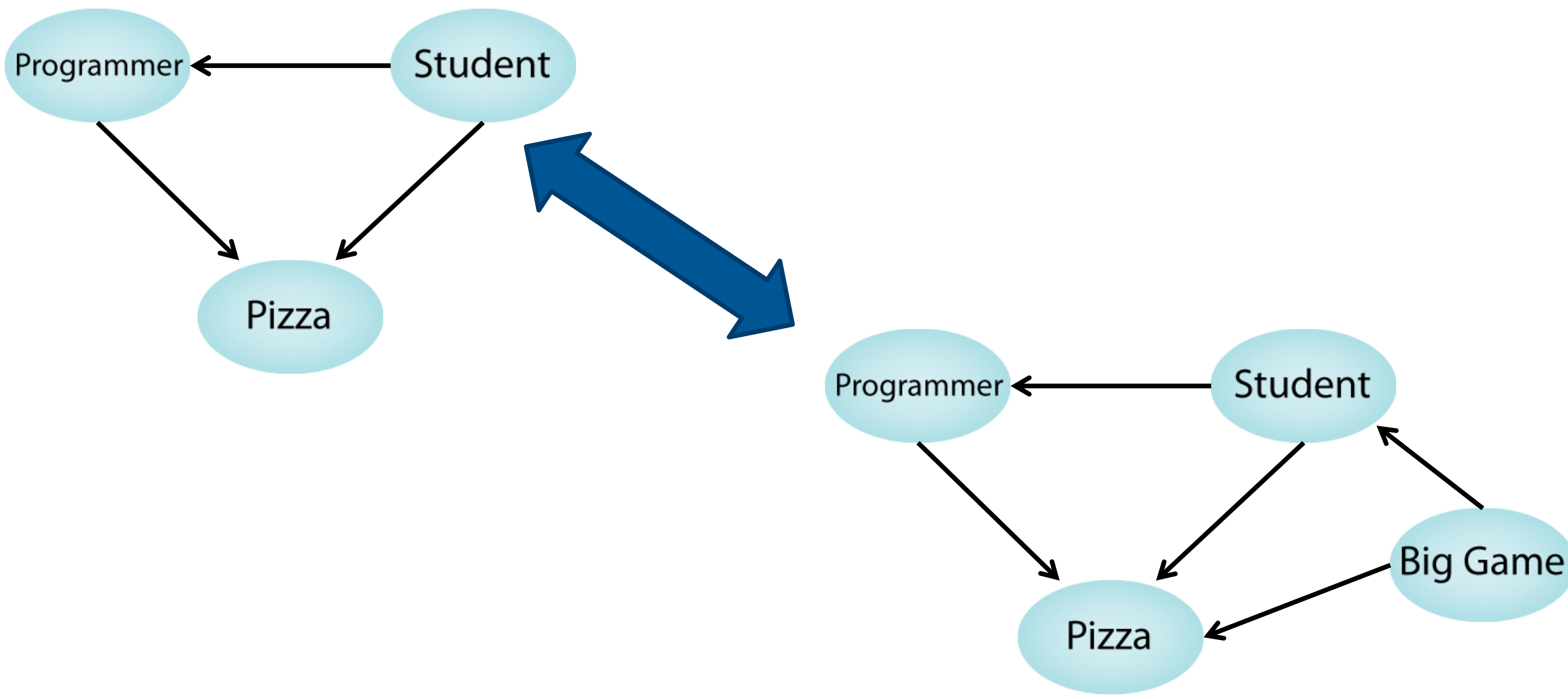
- The code isn’t that bad
  - I could set Pizza to true and run the program
  - But the model is small

- What if we had 10 variables? 100? 1000?
- What if I wanted to know the probability of programmer instead?
- What if each variable has 100 different states?
- What if each variable was continuous (like a normal distribution)?

- The major problem with probabilistic modeling:
  - *Developing a new model is a significant task*
    - Requires implementing representation, reasoning and learning algorithms for everything you want to model!

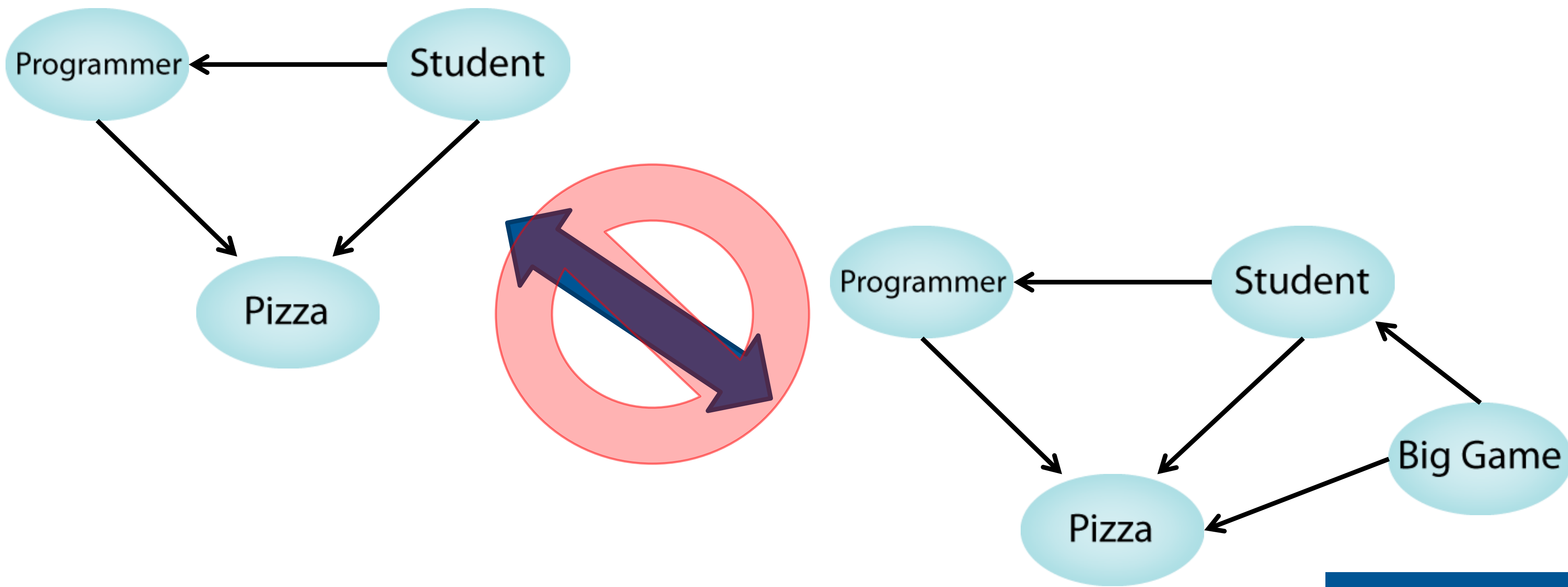
One Simple Extension

- Think of a simple extension to our model
  - What if the big Harvard-Yale game is happening this weekend?
  - Maybe that affects the number of students and pizza eaters



Extension

- These are *not* the same models
  - I have to recode what I just wrote
- Significant amount of wasted effort building models
  - Little re-use of algorithms between two models that are only slightly different
  - Adding a single variable to the model could precipitate reworking a significant amount of code



A Solution

- What if I could code up these probabilistic relationships in a simple and intuitive manner?
- My Scala code could go from this:

```
import scala.util._

def buildModel(iters: Int): Int = {
  if (iters == 0)
    0
  else {
    val prev = buildModel(iters-1)
    val student: Boolean = if (Random.nextDouble() < 0.4) true else false
    val prog: Boolean = student match {
      case true => if (Random.nextDouble() < 0.8) true else false
      case false => if (Random.nextDouble() < 0.3) true else false
    }
    val pizza: Boolean = (prog, student) match {
      case (false, false) => if (Random.nextDouble() < 0.1) true else false
      case (false, true)  => if (Random.nextDouble() < 0.7) true else false
      case (true, false)  => if (Random.nextDouble() < 0.6) true else false
      case (true, true)   => if (Random.nextDouble() < 0.99) true else false
    }
    if (pizza) prev+1 else prev
  }
}

val probPizza = buildModel(100)/100
```

A Solution

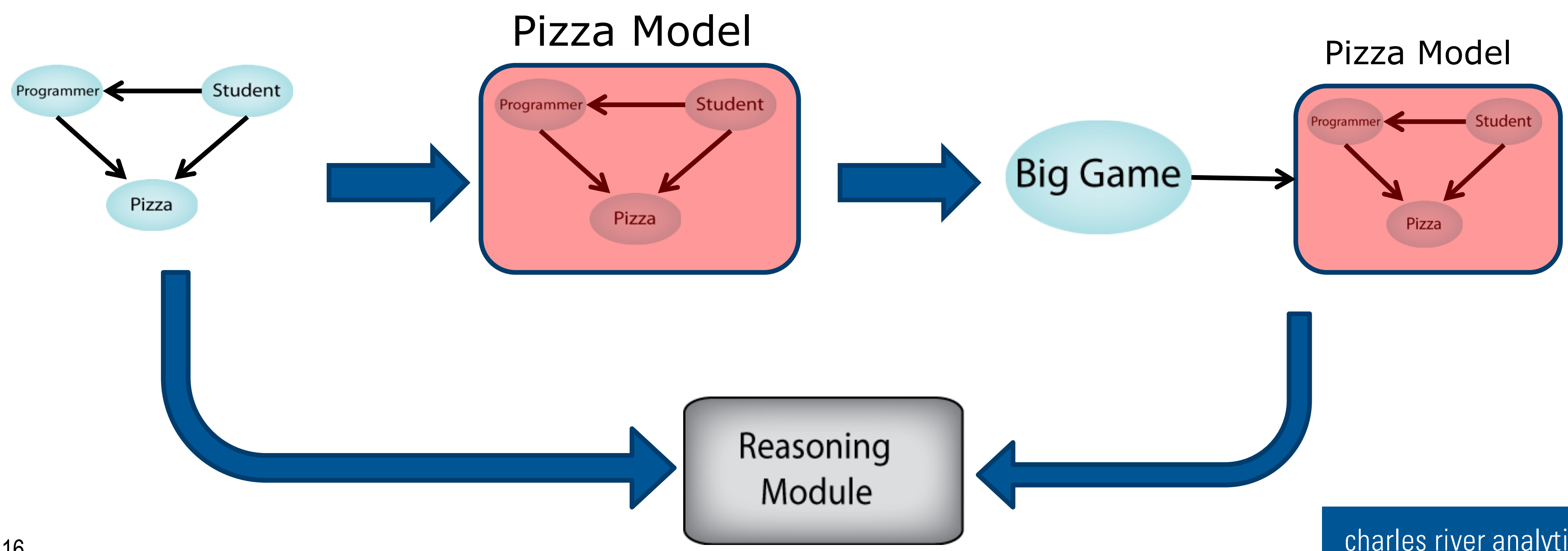
- What if I could code up these probabilistic relationships in a simple and intuitive manner?
- My Scala code could go from this:

```
import com.cra.figaro.language._
import com.cra.figaro.algorithm.Importance._

val student = Flip(0.4)
val prog = If(student, Flip(0.8), Flip(0.3)
val pizza = CPD(prog, student,
    ((false, false), Flip(0.1)), ((false, true), Flip(0.7)),
    ((true, false), Flip(0.6)), ((true, true), Flip(0.99)))
val alg = Importance(100, pizza)
val probPizza = alg.probability(pizza, true)
```

- This way of encoding models is known as *probabilistic programming* using a *probabilistic programming language*

Probabilistic Programming Languages

- Probabilistic programming languages (PPLs)
  - Represent models using the full power of programming languages
    - Data structures, control flow, abstraction, rich typing
  - Facilitate code re-use
  - Provide a suite of built-in inference and learning algorithms that can be automatically applied to new models
  - Provide a language with which to imagine *new* models and representations



Why Do We Need PPLs?

- Probabilistic models have many strengths
  - Succinctness - relationships between random variables simple
  - Powerful – can scale up to thousands of variables
  - Learnable – easily learned from data
  - Solvable – many effective algorithms to reason on these models
- They can be very rich and model a variety of situations
  - hierarchical
  - recursive
  - spatio-temporal
  - relational
  - infinite
- *The easier it is to build models, the more we can take advantage of their power*

Some Example Models

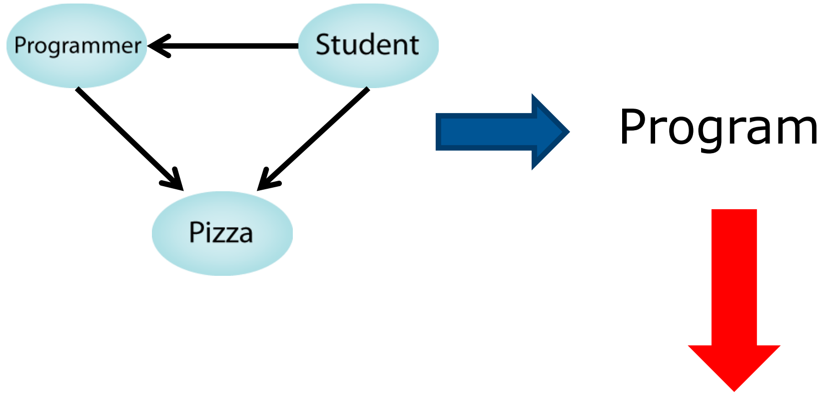
- Popular models that may (or may not) be familiar to people include:
  - Bayesian networks
  - Markov networks/random fields
  - Kalman filters
  - Probabilistic Relational Models
  - Hidden Markov Models
  - Influence Diagrams
  - Many, many more….
- These models form the basis for many everyday automation tasks
  - Spam filters
  - Speech recognition
  - Computer Vision
  - Decision making

Making Probabilistic Programming Practical

- PPLs aim to “democratize” model building
  - One should not need extensive training in ML or AI to build and code a model
- This means that a PPL should (broadly) satisfy two main goals:
  - Usability
    - Intuitive to use
    - Common design patterns easily expressed
    - Integration into other/existing applications
    - Extensible language
    - Extensible reasoning
  - Power
    - Ability to represent a wide variety of models, data, etc
    - Powerful and practical inference techniques

Basic Idea of Probabilistic Programming

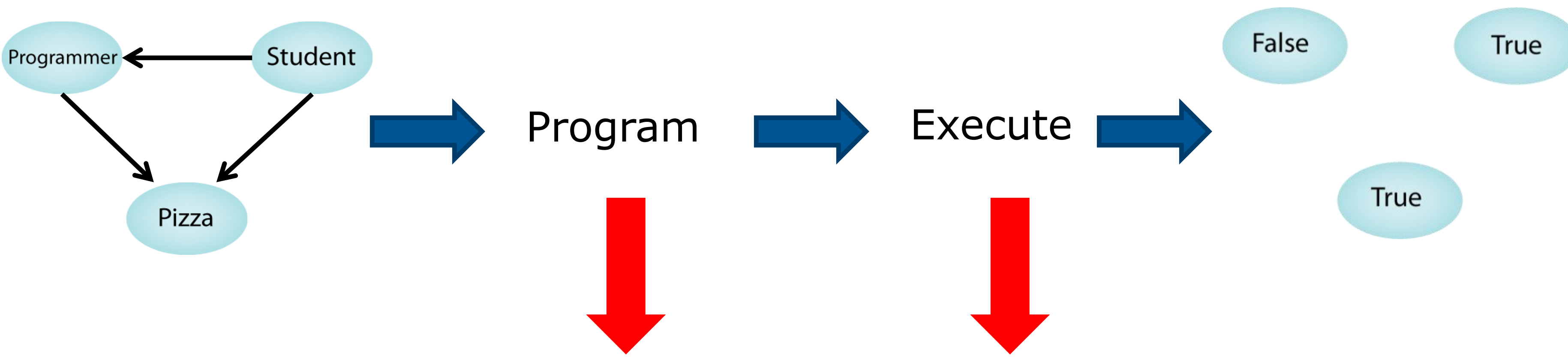
- A “world” can be any data structure
  - A single real value, array, a complete graph
- A “program” is a model of how a world is randomly generated
  - Imagine executing the program to obtain a world




```
val student = Flip(0.4)
val prog = If(student, Flip(0.8), Flip(0.3))
val pizza = CPD(prog, student,
    ((false, false), Flip(0.1)), ((false, true), Flip(0.7)),
    ((true, false), Flip(0.6)), ((true, true), Flip(0.99)))
```

Basic Idea of Probabilistic Programming

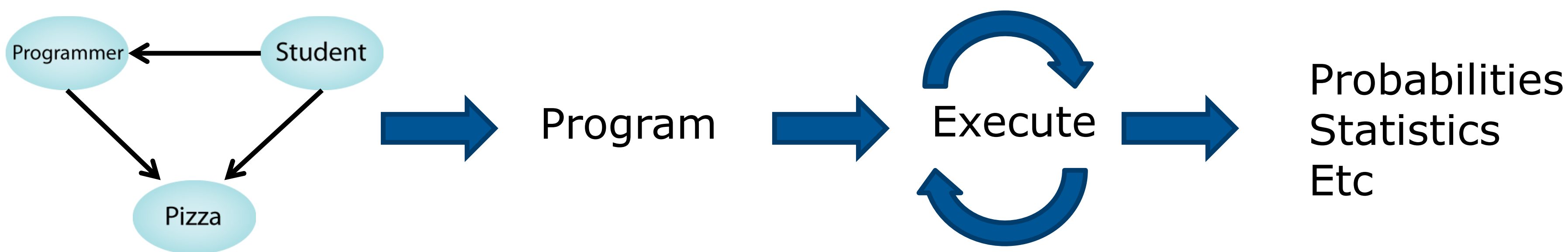
- A “world” can be any data structure
  - A single real value, array, a complete graph
- A “program” is a model of how a world is randomly generated
  - Imagine executing the program to obtain a world




```
student.generate ()
prog.generate ()
pizza.generate ()
```

Basic Idea of Probabilistic Programming

- But programs are not intended to be executed but to be analyzed
  - Not really interested in a single “run” of this program
  - Want to know the behavior of the “program” over many worlds, or analyze a single world
    - Compute a probability distribution over a single world, given observations
    - Compute a distribution over all possible worlds generated from the program



Introducing Figaro

- Figaro is an object-functional PPL
  - Developed by Dr. Avi Pfeffer at Harvard and Charles River Analytics
- An "object-functional" programming language combines functional and object-oriented styles
  - E.g. Scala
- Functional programming provides
  - Powerful representational constructs
  - Reasoning building blocks
- Object-orientation provides
  - Easy representation of common designs
  - Extensibility
- Figaro is currently implemented as a library in…
  - Scala!

Goals of the Figaro Language

- Implement a PPL in a widely-used language
  - Scala is widely-used
  - Scala interoperability with Java also gives Figaro access to an even larger library
- Provide a language to describe models with interacting components
  - Object-oriented
- Provide a means to expressed directed and undirected models with general constraints
  - Functional
- Extensibility and reuse of inference algorithms
  - Object-oriented, traits

- Using Scala helps achieve all of these goals!

Goal 1: Implement a PPL in a widely-used language

Design as a Library

- Figaro is a library in Scala
  - com.cra.figaro.language -> Figaro internals
  - com.cra.figaro.library -> Library of existing distributions/models
  - com.cra.figaro.algorithm -> Library of inference algorithms

- We've seen building a model in Figaro is easy

```
val student = Flip(0.4)
val prog = If(student, Flip(0.8), Flip(0.3)
val pizza = CPD(prog, student,
    ((false, false), Flip(0.1)), ((false, true), Flip(0.7)),
    ((true, false), Flip(0.6)), ((true, true), Flip(0.99)))
```

- But what does all of this mean?

Figaro Internals

- `Flip, If, CPD` are Figaro *elements*
- This is a core concept in Figaro
  - Represented by class `Element[T]`
  - An element represents a process that produces a value of type T
  - Can be stochastic or non-stochastic
  - An element can also use other elements as arguments to produce an output value
- All Figaro library elements are subclasses of `Element[T]`

```
abstract class Element[T] {
  var value: T = _
  type Randomness
  def generateRandomness(): Randomness
  def generateValue(r: Randomness): T
}
```

The `Element[T]` Class

- Two functions need to be defined in an instantiation of an element
  - `generateRandomness`: function that randomly generates a value *r* according to some probability distribution
  - `generateValue`: function that *deterministically* computes the value of the element given *r* and the values of its arguments

  - Example: Normal distribution

```
class AtomicNormal(val mean: Double, val standardDeviation: Double)
        extends Element[Double] {
  Randomness = Double
  def generateRandomness(): Double = {
    val u1 = random.nextDouble()
    val u2 = random.nextDouble()
    val w = sqrt(-2.0 * log(u1))
    sin(2.0 * Pi * u2) * w
  }
  def generateValue(r: Double) = r * standardDeviation + mean
```

The `Element[T]` Class

- To generate a value from an element
  - Call generate() which does

```
def generate(): Unit = {
  r = generateRandomness()
  value = generateValue(r)
}
```

- Elements are parameterized by the types of values they produce
  - E.g. `Element[Boolean]` is the type of elements that produce Boolean values
  - Parameterization one of the major strengths of Figaro over other PPLs
    - Can create elements over basic types, other classes, entire processes, etc
    - Graphs or DNA sequences

Element Classes

- Figaro comes with many elements for common processes

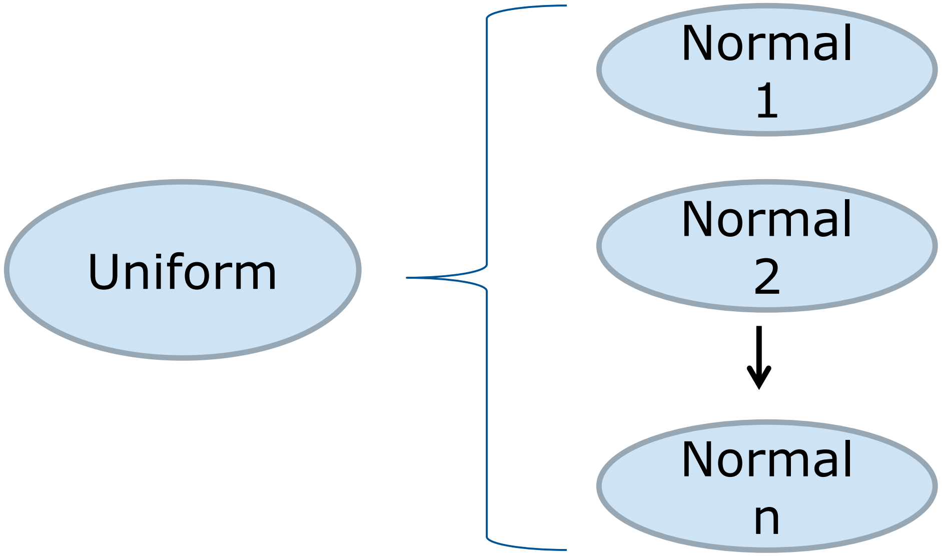
- Simple Elements
  - `Constant(x)` – a distribution that always returns the value x
  - `Flip(p)` – a Bernoulli trial, i.e., return true with probability p, false otherwise
  - `Select(clauses*)` – select a value at random from a list according to given probabilities
  - `If(testElement, thenClause, elseClause)` – Not the scala "if"; choose between thenClause and elseClause depending on the current value of testElement, which is an `Element[Boolean]`

- Many discrete and continuous probability distributions
  - Uniform
  - Normal
  - Poisson
  - Gamma
  - Binomial
  - Many more…

Element Classes, Cont

- Many of these distributions come in two flavors
  - Atomic – their parameters are fixed values, e.g., mean and std dev of normal distribution is fixed at instantiation time
  - Compound– their parameters are themselves other elements.
    - Normal(meanElement, stddev) represents a normal distributions whose mean depends on the current value of meanElement

```
val mean = Uniform(0, 10)
val norm = Normal(mean, 1.0)
```



Adding New Elements

- Most of the internal Figaro workings are defined in the `Element[T]` class
- Creating new elements very easy
- Let's say we want model the distribution of the maximum value of X draws from zero to an upper bound
  - Eg, pull 10 random integers from 0 to 100, return the largest value

```
class MaxValue(val numTries: Int, val UpperBound: Int)
      extends Element[Int] {
  type Randomness = List[Int]
  def generateRandomness(): List[Int] = {
    List.tabulate(numTries)(i => Random.nextInt(UpperBound))
  }
  def generateValue(r: List[Int]) = r.max
```

Adding New Elements

- We just added an *atomic* element
- What about compound elements?

```
val mean = Uniform(0.0, 10.0)
val norm = Normal(mean, 1.0)
```

- To do this, we borrow from functional programming

Function Programming ⟷ Monad

Probabilistic Programming ⟷ Probability Monad

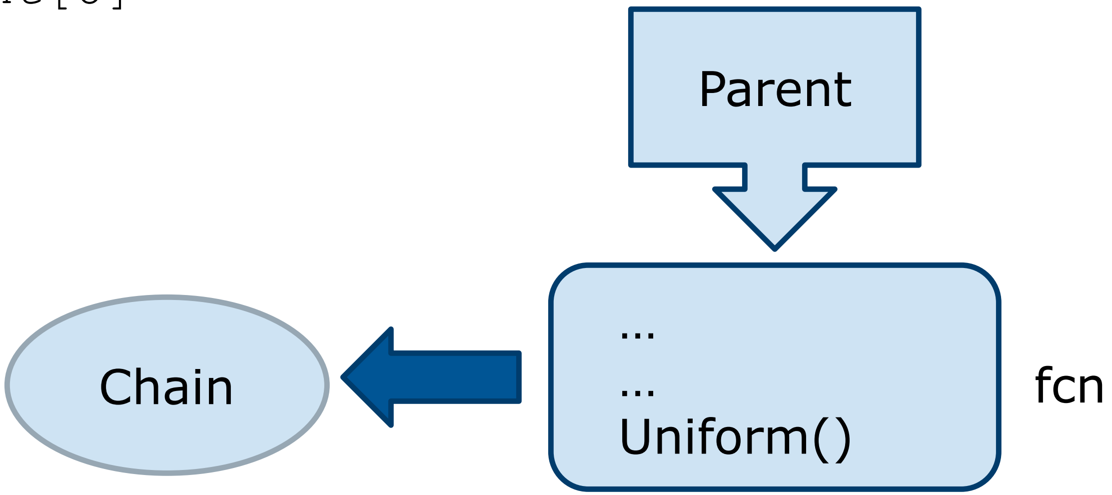
The Probability Monad

- Figaro makes extensive use of *probability monads*
  - Monads: lift computation from space of values to space of concepts over values
  - Probability monad: lifts computation from values to probabilistic models over values
- Figaro implements three monadic operations in three different elements:
  - Monadic unit -> Constant
  - Monadic bind -> Chain
  - Monadic fmap -> Apply
- Many Figaro elements are implemented through a combination of these three element classes

- `Constant(x)`: lifts the value x to the probability model that returns x with probability 1

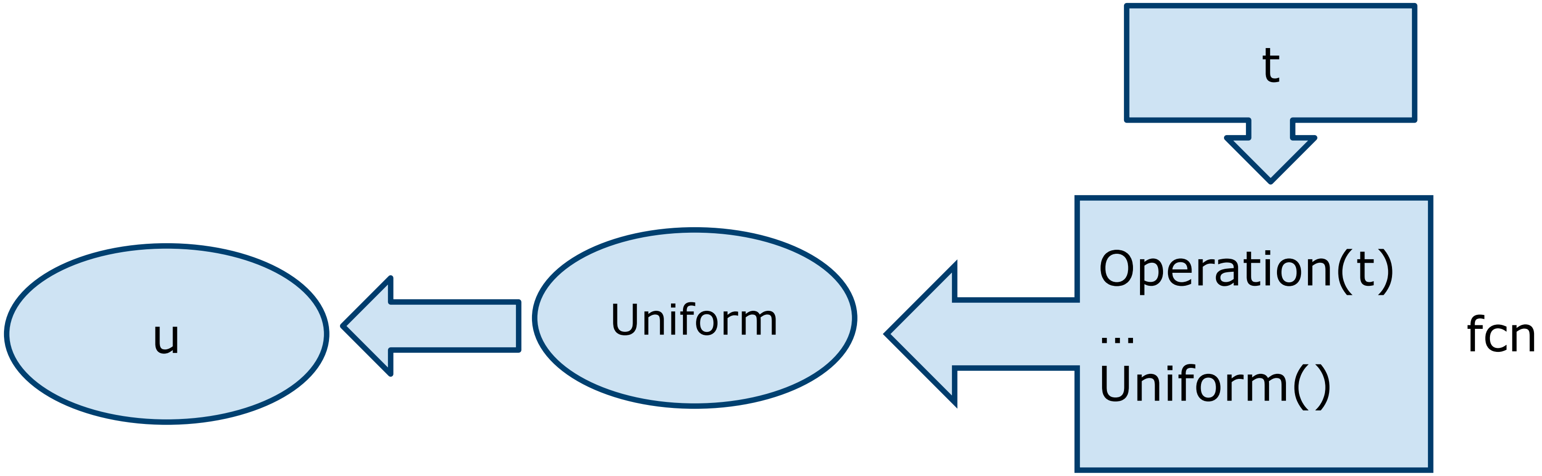
Chain

- `Chain[T,U]` represents a computation from an `Element[T]` to an `Element[U]`
  - It literally chains together probabilistic computations
- Takes two arguments:
  - *parent*, a `Element[T]`
  - *fcn*, a function that takes a value of type `T` and returns an `Element[U]`



Chain

- Calling `generateValue` on a Chain is a three step process
  1. Retrieve a value t from parent
  2. Generate an `Element[U]` by calling *fcn*(*t*)
  3. Return a value u from the returned `Element[U]`




```
class Chain[T,U](val parent: Element[T], fcn: T => Element[U])
    extends Element[U] {
 def generateValue() = {
    fcn(parent.value).value }
}
```

Chain Examples

- Many model classes are implemented using Chain, e.g.

- If

```
class If[T](testElement: Element[Boolean],
    thenClause: Element[T], elseClause: Element[T])
  extends Chain[Boolean,T](
    testElement,
    (b: Boolean) => if (b) thenClause; else elseClause)
```

Chain Examples

- Normals where the mean changes

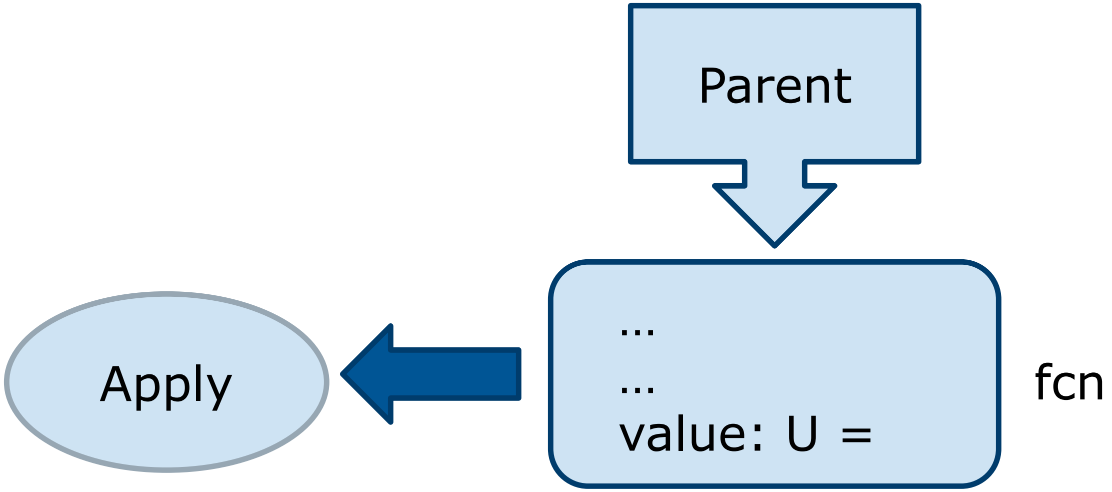
```
class NormalCompoundMean(val mean: Element[Double],
    val stdDev: Double)
  extends Chain(mean,
          (m: Double) => new AtomicNormal(m, stdDev))
```

- Normals where the mean *and* std dev changes
  - Chain of Chain

```
class NormalCompound(val mean: Element[Double],
    val stdDev: Element[Double])
  extends Chain(mean,
          (m: Double) => new Chain(stddev,
          (s: Double) => new AtomicNormal(m, s))
```

Apply

- Represents the monadic *fmap*
- Apply is a element class that allows Scala functions to be integrated into Figaro models
- It can be thought of as "lifting" a function from the space of values to the space of elements
- Like Chain, it takes two arguments
  - *parent*, a `Element[T]`
  - *fcn,* a function that takes a value of type `T` and **value** of type `U`



Apply

- Example: Distribution over the sum of two normals

```
val norm1 = Normal(0.0, 1.0)
val norm2 = Normal(-1.0, 2.0)
val sum = Apply(norm1, norm2, (x: Double, y: Double) => x + y)
```

Note: Scala's type inference is handy here, since we don't need to explicitly declare all the parameterization

Some other handy examples

- Use Apply to convert tuples of elements into elements of tuples

```
val e1: Element[Int] = …
val e2: Element[Double] = …
val tupleElement: Element[(Int, Double)] = ^^(e1, e2)
```

Where `^^ = Apply(arg1, arg2, (t1: T1, t2: T2) => (t1, t2))`

- Use Chain to create conditional probability distributions (CPDs)

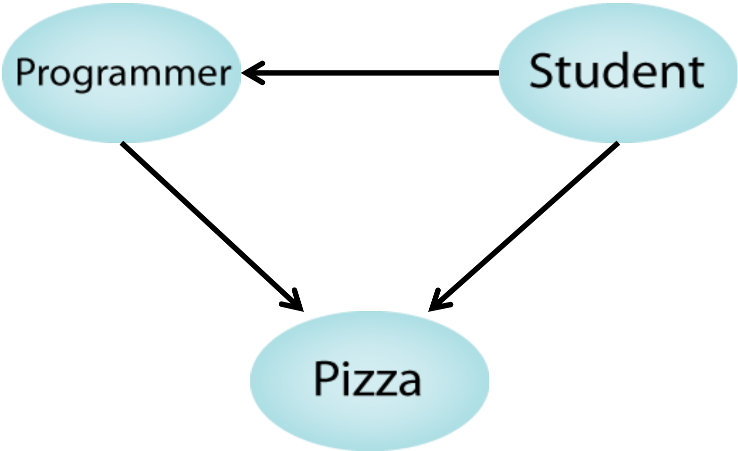
```
class CPD[T,U](arg1: Element[T], clauses: Seq[(T, Element[U])])
  extends Chain[T,U](arg1, (t: T) => getMatch(clauses, t))
```

Where `getMatch` is just a function that matches the value of `arg1` to the clause values

Note that multi-argument versions of Chain and Apply are available

Does Figaro Meet the Two PPL Goals?

- Usability?
  - Writing models in Figaro easily accomplished by stitching together elements
- Earlier Example:




```
val student = Flip(0.4)
val prog = If(student, Flip(0.8), Flip(0.3)
val pizza = CPD(prog, student,
    ((false, false), Flip(0.1)), ((false, true), Flip(0.7)),
    ((true, false), Flip(0.6)), ((true, true), Flip(0.99)))
```

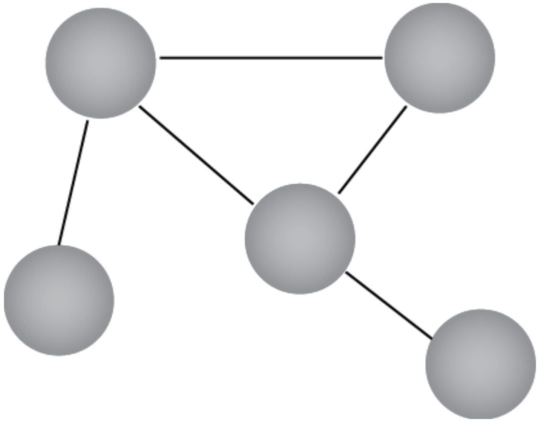
Goals, cont

- Power?
  - Absolutely: Chain + recursion = Huge potential

Example: PageRank

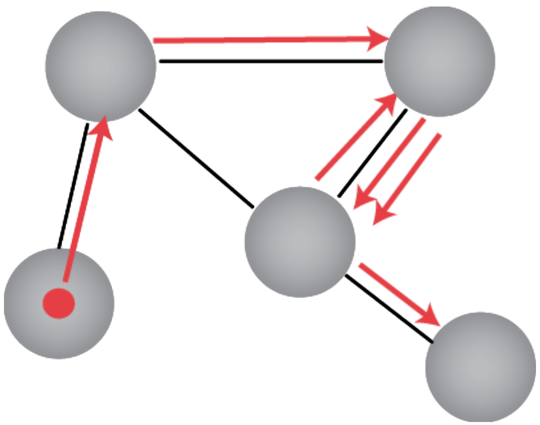
- Google's PageRank is a model of a probabilistic process on a graph
- We can model this process in Figaro
  - Do a slightly modified version for simplicity

- Each webpage on the internet is a node in a graph
- Draw an edge between each node if the webpages link to each other
  - Real PageRank uses directed edges



PageRank

- The probabilistic process is known as a *random walk*



- Start at some node on the graph
- Randomly move to one of the node's neighbors
- Repeat the process for some time steps
- Record all the nodes visited
- The more times a node is visited, the higher its PageRank

Random Walk in Figaro

- Start by defining some data structures for a webpage graph

```
class Edge(from: Int, to: Int)

class Node(ID: int, edges: Set[Edge])

class Graph(nodes: Set[Nodes]) {
  def get(id: Int) = // return Node with ID == id
}

// some function that builds a graph given some params
def graphGenProcess(params*): Graph
```

Random Walk

- Define some parameters of the random walk

```
val numSteps: Int = 10
val startNode: Int = 0
Val inputGraph: Graph = graphGenProcess(…)
```

- Now that we have these parameters, we have to "lift" them into the space of elements

```
val inputGraphElem: Element[Graph] = Constant(inputGraph)
val numStepsElem: Element[Int] = Constant(numSteps)
val startNodeElem: Element[Int] = Constant(startNode)
```

- If I choose, can also model the parameters to the random walk as a probabilistic process

Random Walk in Figaro

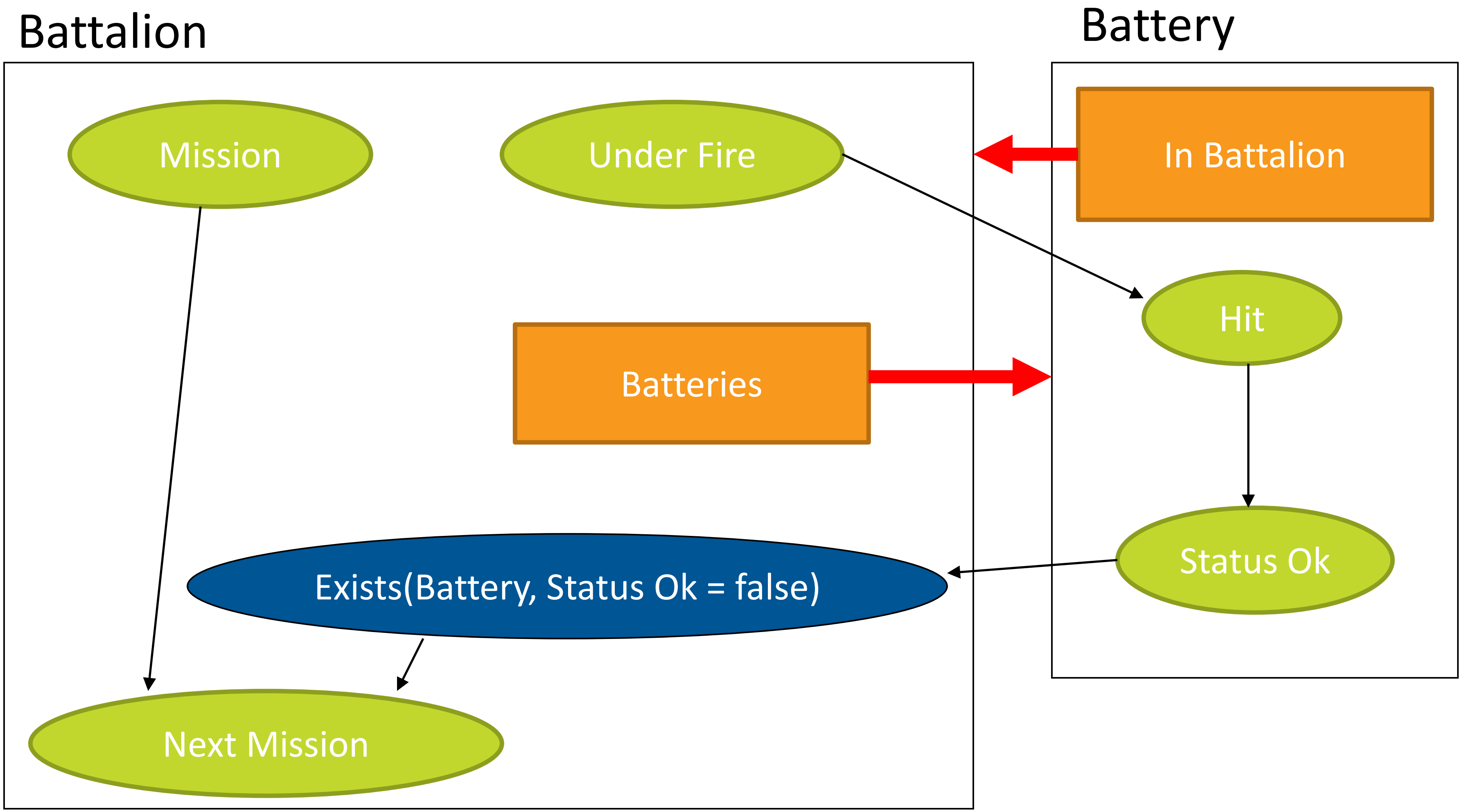
```
val rWalk = Chain(inputGraphElem, numStepsElem, startNodeElem,
  rFcn)

def rFcn(g: Graph, remain: Int, n: Int): Element[List[Int]] = {
  if (remainSteps == 1)
    val curr = step(Constant(List(n)), g)
    Apply(curr, (i: Int) => List(i))
  else {
    val prev = rFcn(g, remain-1, n)
    val curr = step(prev, g)
    Apply(curr, prev, (i: Int, l: List[Int]) => List(i)::l)
  }
}
def step(hist: Element[List[Int]], g: Graph): Element[Int] = {
  Chain(hist, (i: List[Int]) =>
   lastNode = g.get(i.head)
   Select(lastNode.edges.map(e =>(e.to, 1/lastNode.edges.size)))
}
```

Goal 2: Interacting Objects

- Since Scala is OO, can create complex Class-Element relationships
  - Classes containing elements
  - Elements of classes
  - Highly reusable, flexible and scalable

- Figaro and Scala are natural means to build Probabilistic Relational Models (PRMs)
  - Describe world in terms of objects and relationships
    - Graphical model representation of relational database
  - Probability models associated with classes
    - small and self-contained
    - apply to many situations and instances
  - PRMs difficult to represent in other PPLs
    - No encapsulation

Example PRM



- For time purposes, will not delve into this
  - Good examples of PRMs in code with release

Goal 3: Directed and Undirected Models with Constraints

Conditions and Constraints

- Functional nature of Figaro lets us define conditions and constraints on our models
- A *condition* is a function *f* from a value to a Boolean
  - Think of this as an observation of some variable
  - But can be any arbitrary function that returns a boolean from a value of the element
- A (soft) *constraint* is a function *f* from a value to a real number
  - *f* can be any programmable function
  - Essentially saying "some value of element e is x times more likely than another value"

Undirected Models

- Constraints and conditions are particularly useful on *undirected* models
  - Undirected can model some dependencies that directed models cannot
  - Also known as Markov networks, Markov random fields
- Example
  - Smokers and Friendship
  - People who smoke tend to have friends that smoke (and vice-versa)

Smokers Model

```
class Person {
  val smokes = Flip(0.6)
}
val alice, bob, clara = new Person
val friends = List((alice,bob), (bob,clara))

def smokeInfluence(pair: (Boolean, Boolean)) =
  if (pair._1 == pair._2) 3.0 else 1.0
for {(p1, p2) <- friend} {
  ^^(p1.smokes, p2.smokes).constraint(smokeInfluence)
  // creates an element tuple for each frienship and
 constrains its value
}
clara.smokes.condition((b: Boolean) => b == true)
// run inference
```

Goal 4: Extensibility and Reuse of Inference Algorithms

Inference

- So far we have just talked about building models
- But most people want to *do* something with the models they build
- Generally want to infer or reason with the model, for example
  - The distribution over some variable in the model, given some evidence
  - Some statistics about the model – mean, variance, etc

- This is where many of the benefits of PPLs are realized
  - Most algorithms work “out of the box” for any model that a user creates!
  - Very extensible algorithm library using traits and inheritance

Main Ideas of Figaro Algorithms

- New algorithms are constantly being developed
- Different algorithms are good for different problems
  - $\Rightarrow$ Anyone should be able to implement new algorithms
  - $\Rightarrow$ Algorithms should be implemented as a service
  - $\Rightarrow$ Algorithms should specify declaratively when they work

- Several completely implemented inference algorithms included in Figaro
  - Variable Elimination
  - Importance and Forward Sampling
  - Metropolis-Hastings
  - Particle Filtering

Extensibility

- These algorithms built on a framework of classes and traits
  - trait Algorithm
  - traits OneTime and AnyTime define how the algorithm is run
  - ProbQueryAlgorithm and ProbEvidenceAlgorithm are two bases classes define the information the algorithm is computing
  - Sampler trait that defines interface for sampling algorithms…
  - Many more…
- *General idea is that creating a new algorithm should be done through existing traits and by subclassing*

How the Algorithm is Run

- Figaro breaks algorithms into two runnable types
- OneTime
  - Run the algorithm once, produce answer
- Anytime
  - Run the algorithm continuously
  - At any time the algorithm is interrupted, produce the *best* answer achieved so far
  - User can continue the algorithm where it left off

Running Algorithms

- Recall the smoking example

```
class Person { val smokes = Flip(0.6) }
val alice, bob, clara = new Person
val friends = List((alice,bob), (bob,clara))
// constraints…
clara.smokes.condition((b: Boolean) => b == true)
```

Running Algorithms

- Want to infer the probability that alice smokes:

Target

```
val alg = Importance(10000,alice.smokes)
alg.start()
alg.probability(alice.smokes, true)
```

Run once for 10,000 iterations

```
val alg = Importance(alice.smokes)
alg.start()
Thread.sleep(1000)
alg.stop()
alg.probability(alice.smokes, true)
```

Run continuously, stop after 1 second

What’s Next?

- We are constantly updating and improving Figaro
- Major improvements we are working include:
  - Better debugging tools
  - Distributed models
  - Parameter learning
  - Intelligent Metropolis-Hastings
    - Automatic proposal distributions

Lessons Learned

- Some reflections on my experience with Figaro and Scala

- First: I am new to Scala – learned Scala when I learned Figaro
  - Came from a heavy C/C++/Matlab background
  - The verdict: Scala is great!
  - While those languages have their use, I’m pretty much a Scala convert

Lessons Learned

- I don’t think something like Figaro could be written in another language
  - Object-oriented is essentially *required* to build some models
  - Could we do this with Java? Maybe
  - But functional aspects of Scala make creating Figaro much easier

Lessons Learned

- In Figaro, implicits are our friends
  - We make heavy use of implicit arguments and conversions
  - Want to make Figaro as easy as possible for the everyday user, but allow power for the experienced user
- However, sometimes we can't hide everything from the user

```
class DecisionPolicyExact[T, U](policy: Map[T, (U, Double)])
  extends DecisionPolicy

class DecisionPolicyApprox[T <% Distance[T], U](policy: Map[T,
  (U, Double)]) extends DecisionPolicy


trait PolicyMaker[T,U] {

  def makePolicy(policyMap: Map[(T,U), DecisionSample]):
  DecisionPolicy[T,U]

}
```

Lessons Learned

- We'd prefer to not have users doing the instantiation of Exact or Approx classes
- So we make a trait to do the instantiation

```
trait ExactPolicyMaker[T,U] extends PolicyMaker[T,U] {
  def makePolicy(policyMap: Map[(T,U), DecisionSample]) =
  DecisionPolicyExact(policyMap)
}
trait ApproxPolicyMaker[T,U] extends PolicyMaker[T,U] {
  def makePolicy(policyMap: Map[(T,U), DecisionSample]) =
  DecisionPolicyApprox(policyMap)
}
```

- Doesn't work because the view bounds on the Approx must be defined at instantiation time
- So the user has to do a little more work in their code

More Lessons Learned

- Chain is powerful, but problematic
- Mainly:

- Scala scope of objects $\neq$ - Figaro scope of objects

- Just because element is created or used in a chain, does not mean it goes out of model scope when the chain function call is complete

```
val f1 = Flip(0.1)
val f2 = Flip(0.2)
val c = Chain(Flip(0.3), (b: Boolean) => if (b) f1 else f2)
```

More Lessons Learned

- More problematic (but a valid program):

```
val c = Chain(Flip(0.3), (b: Boolean) =>
           if (b) Flip(0.2)("true") else Flip(0.3)("false"))
val t = Universe.getElement("true")
```

- Once an element created, we must ensure that it always remains referenced since it may be used later!
  - Especially in inference algorithms
- Requires us to use lots of data structures to keep track of elements
  - Ie, *we* are taking a more active control of memory management
- Leads to some memory leaks in Figaro!

More Lessons Learned

- Finally with Chain, consider:

```
val c = Chain(Normal(0, 1), (d: Double) => Constant(d))
```

- A normal distribution is a continuous value
- Repeated sampling of c will constantly create new elements
- Object creation imparts some overhead from Scala, *as well* as our element management
    - This becomes a significant bottleneck in sampling algorithms
    - So we implement limited caching – on this example, not every useful, but for discrete values it is
    - We still have not solved the problem of speeding up Chain execution

- *Vast majority of Figaro bugs are found in element memory management and Chain caching!*

Implied Goal 5: Get People to Use Figaro!

- Many more features of Figaro that I haven’t touched upon
  - Element references – naming elements, collections of elements, aggregation of elements with the same name
  - Universes – where an element “lives”, running algorithms between universes
  - List elements – lists of random length and value
  - Decision-making *New!*
    - Library added to reason about structured decision problems
    - Bayesian networks with decisions known as Influence Diagrams
    - Can compute optimal decisions over complicated data structures like graphs or DNA sequences (examples included in the code)

Availability

- Figaro is open source
- Version available now has most of the features I talked about (except decision-making)
- New release very soon, hopefully within a month or two
  - Lots of bug fixes, decision-making, Scala 2.10 support
- Request a copy by going to
  - www.cra.com/figaro and filling out the form
  - Email me: bruttenberg@cra.com
  - Contains a short tutorial
  - For more information also see Avi's paper "Creating and Manipulating Probabilistic Programs with Figaro" in *UAI Workshop on Statistical Relational Artificial Intelligence 2012*
- We are discussing setting up a GitHub project for Figaro
  - Not finalized, so may not happen
- We welcome feedback and improvements!

Conclusion

- Figaro is open source, Scala library where one can create probabilistic models with little AI and ML experience
- Can we say that "practical" probabilistic programming has been reached?
  - Probably not, but certainly Figaro is a huge step in that direction
- Figaro is a language and platform with which one can explore new types, paradigms and ways of building probabilistic models
  - Can build models in Figaro that model other Figaro models!?!?
  - Many more things possible that we haven't even thought of yet
- We hope other people can find it useful as well

Thank You and Questions