The Semantics of Figaro

Avi Pfeffer

Charles River Analytics

POPL Workshop on Probabilistic Programming Semantics, January 2016

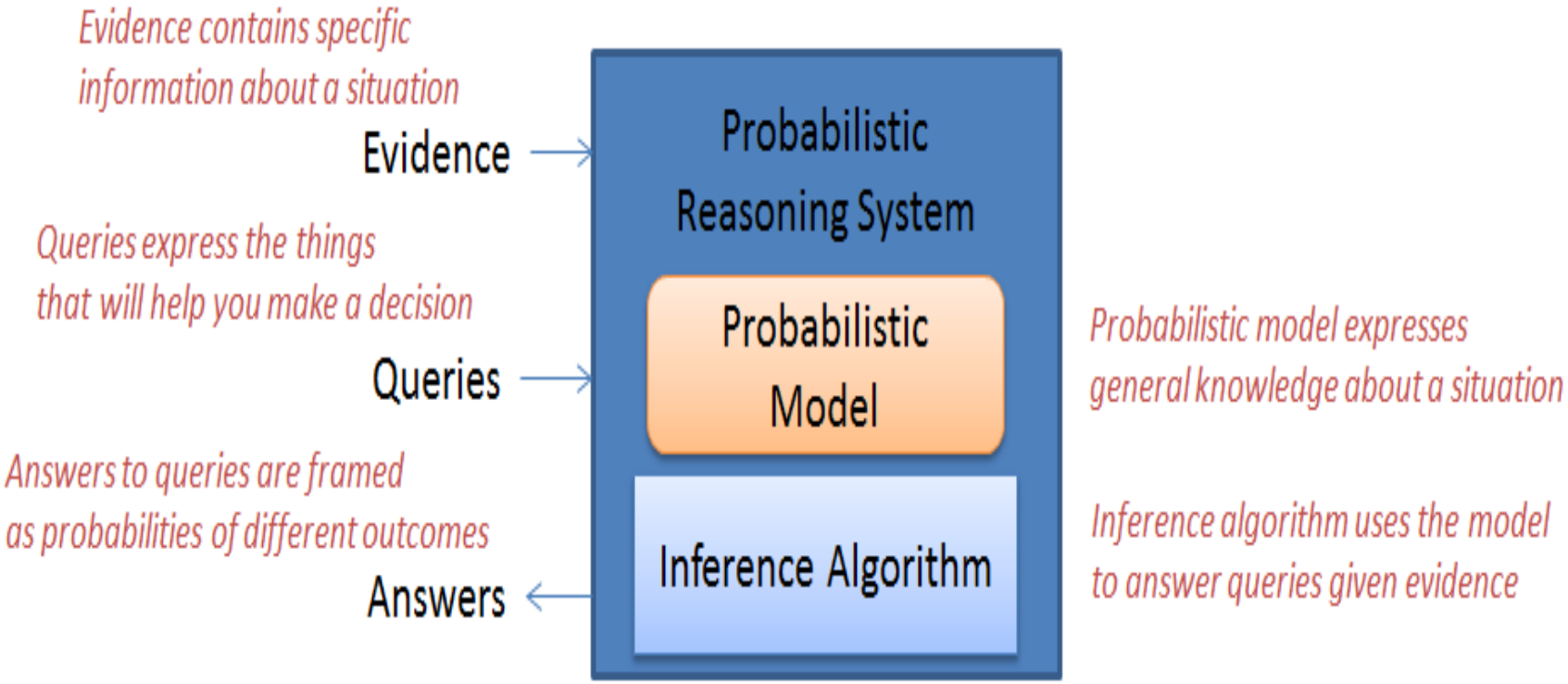
Overview

- Introduction to Probabilistic Programming
- Figaro Language Design
- Semantics

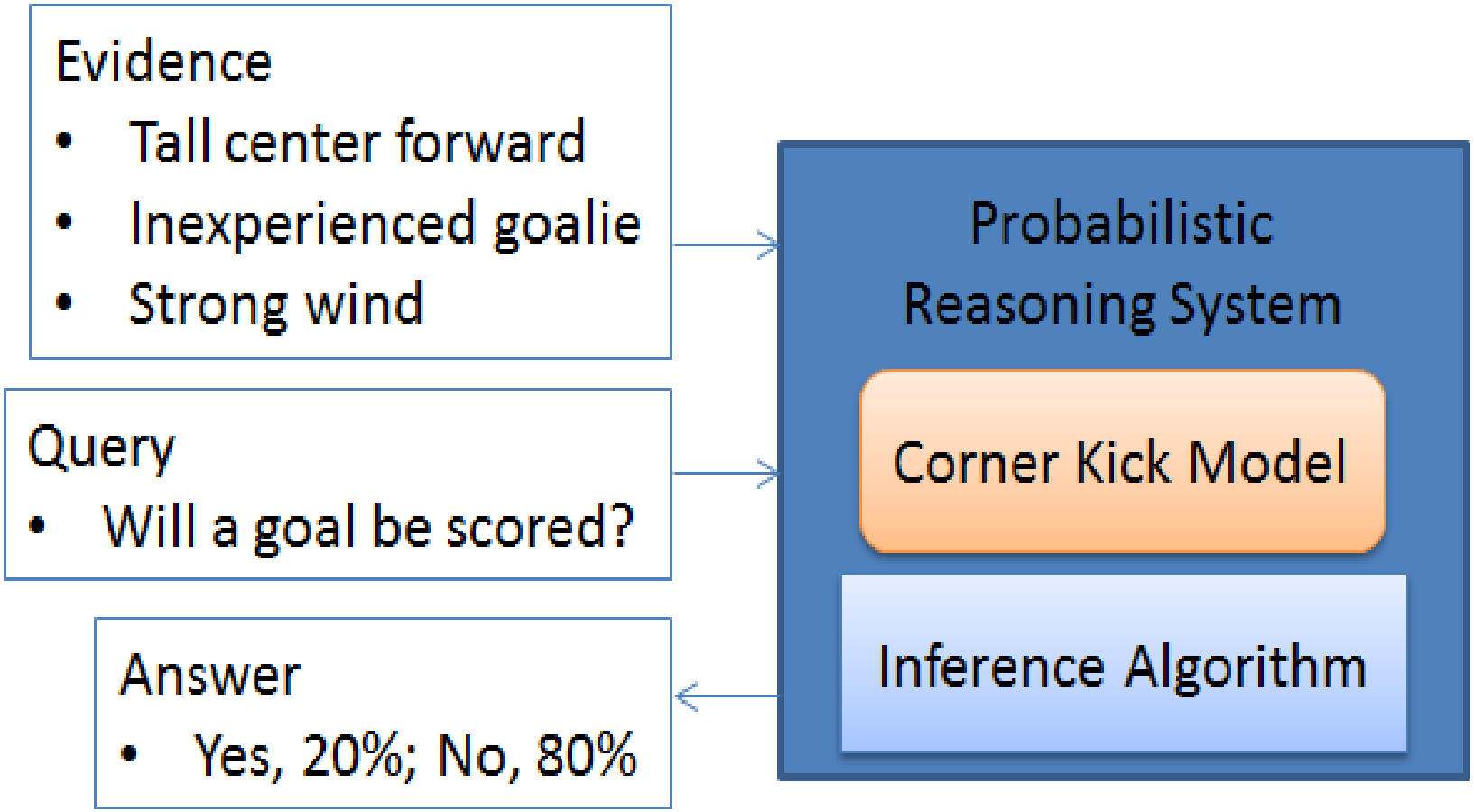
What is Probabilistic Reasoning?

- Much modern machine learning is probabilistic
- Define a probabilistic model over your domain
- Learn the parameters or structure of the model from data
- Use your model to reason about your domain
  - Predicting future events
  - Inferring causes of observed events
  - Using current observations to predict the future

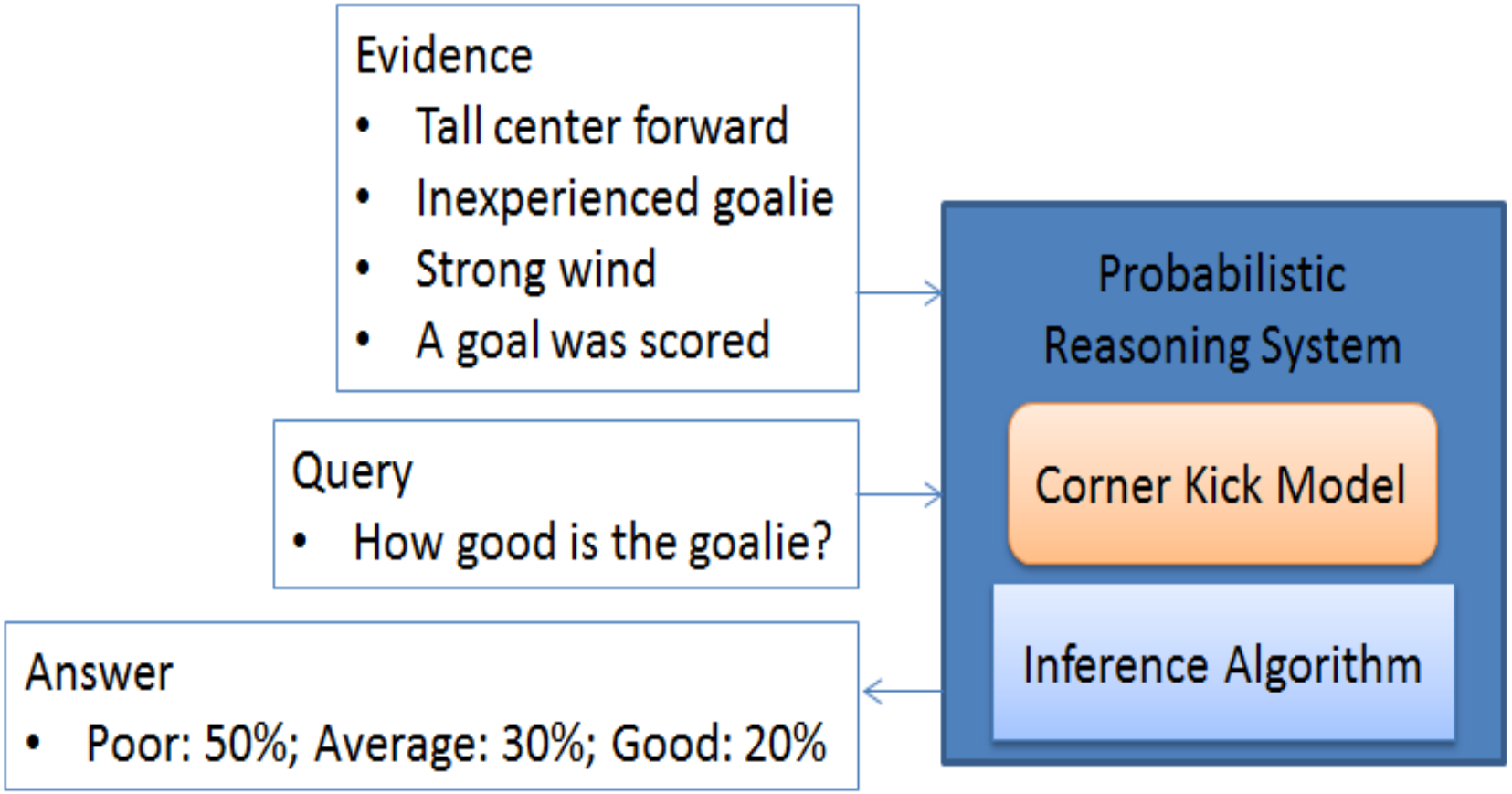
Probabilistic Reasoning: The Gist



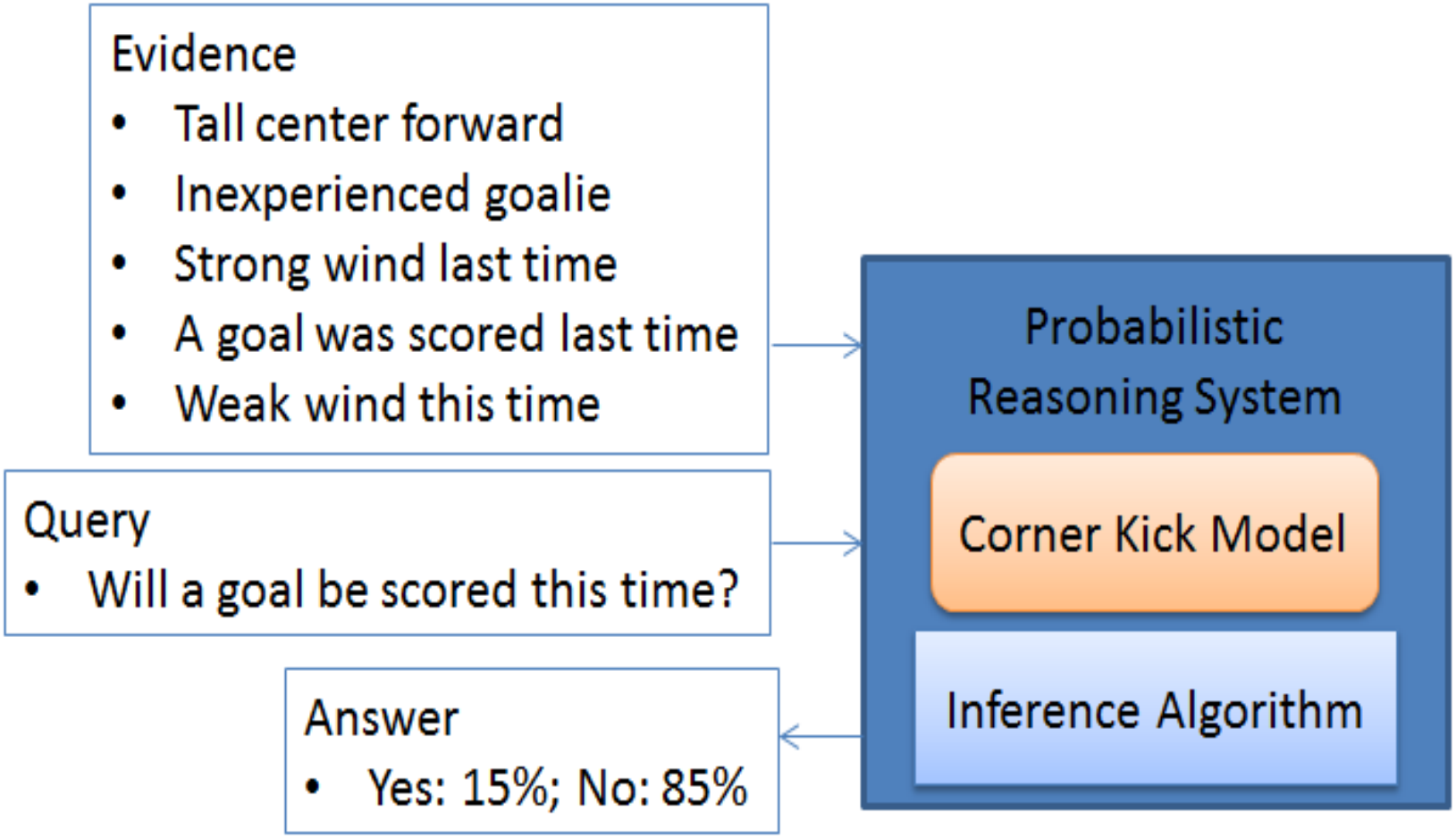
Probabilistic Reasoning: Predicting the Future



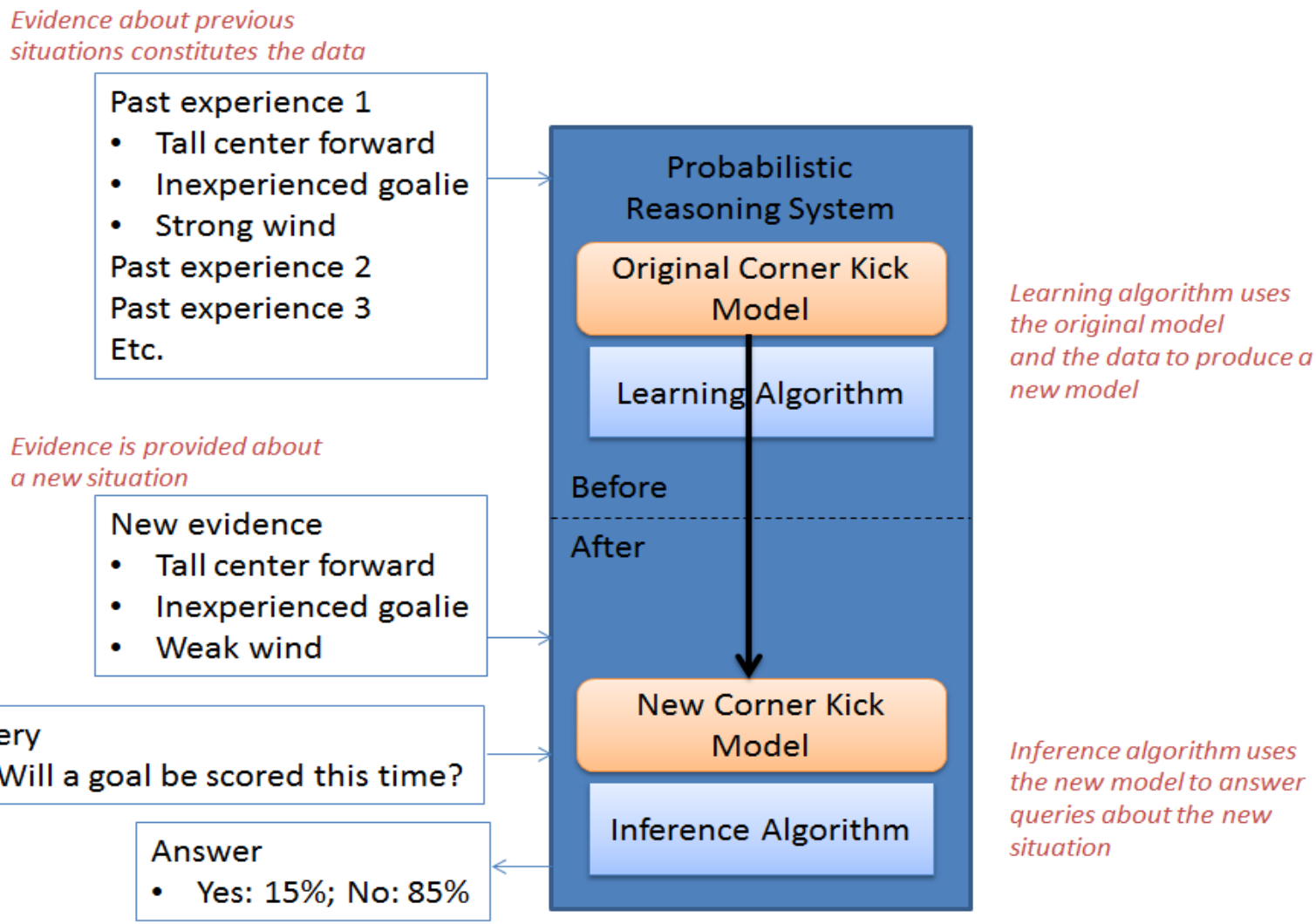
Probabilistic Reasoning: Inferring Factors that Caused Observations



Probabilistic Reasoning: Using the Past to Predict the Future



Probabilistic Reasoning: Learning from the Past



Probabilistic Modeling in the Last Few Years

- Models ever growing in richness and variety
  - hierarchical
  - recursive
  - spatio-temporal
  - relational
  - infinite
- Using a probabilistic model requires a lot of work
  - Representation, inference, and learning
- Probabilistic programming provides a general-purpose solution to building all sorts of probabilistic models
  - Key insight: algorithms for graphical models generalize to programs

Overview

- Introduction to Probabilistic Programming
- Figaro Language Design
- Semantics

Figaro goals

A probabilistic programming system that is:

- Easy to interact with data
- Easy to integrate with applications
- General representation to capture common programming patterns
  - Functional programming
  - Object-oriented programming
  - Constraints for undirected models
- An extensible library of inference algorithms
  - Sampling algorithms
  - Factored algorithms
  - Applying multiple algorithms to a model

Figaro as a Scala Library

- Figaro provides data structures to represent probabilistic programs
- Scala programs construct the Figaro models
- Inference algorithms implemented in Scala operate on these models

Scala Embedding: Pros and Cons

- Pros:
  - Easy interaction with data and integration with applications
  - Can embed general-purpose code in probabilistic programs
  - Can construct models programmatically
  - Figaro inherits functional and object-oriented features of Scala
  - Can use Scala functions to specify constraints
  - Scala supports extensible library of inference algorithms

- Cons:
  - Hard to reason about models at source level, since arbitrary Scala code may be embedded in model
  - Syntax is not as elegant as self-contained languages
  - Semantics is harder

Basic Figaro Concepts

- Element[T] is class of probabilistic models over type T

- Atomic elements

Constant[T], Flip, Uniform, Geometric

- Compound elements built out of other elements

If(Flip(0.8), Constant(0.5), Uniform(0,1))

- Example program and solution

```
val x = If(Flip(0.8), Constant(0.5), Uniform(0,1))
val algorithm = VariableElimination(x)
algorithm.start()
println(algorithm.probability(x, 0.5))
```

Chaining Elements Together

- Chain[T,U] takes two arguments:
  - The *parent*, which is an Element[T]
  - The *chain function*, which is a function from a value of type T to an Element[U]

- Example: Chain(Uniform(0,1), (d: Double) => Normal(d, 0.5))

- Chain is the bind operation for the probability monad

Chain Generative Process

Chain(Uniform(0,1), (d: Double) => Normal(d, 0.5))

1. Generate a value from the parent Uniform(0, 1), say 0.6
2. Apply the function to this value to get Normal(0.6, 0.5)
3. Generate a value from Normal(0.6, 0.5), say 0.62

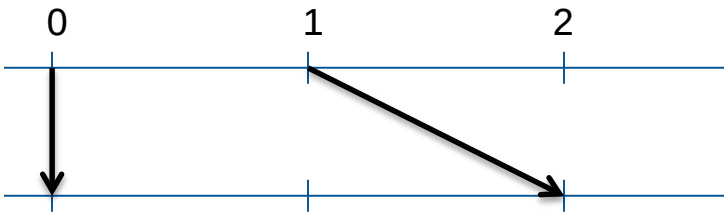
- This chain represents a normal distribution whose mean is uniformly distributed between 0 and 1

Integrating Scala Functions

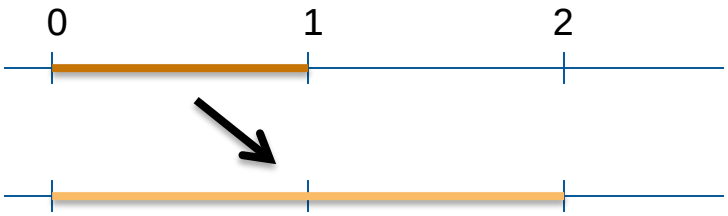
- Apply lifts Scala functions to elements
  - Monadic fmap

Apply(Uniform(0,1), (d: Double) => d * 2)

(d: Double) => d * 2



Apply(Uniform(0,1), (d: Double) => d * 2)



Conditions and Constraints

- Any Element[T] can have conditions and constraints
- Condition: function from T to Boolean
  - Specifies a property that must be satisfied for a value to have positive probability
- Constraint: function from T to Double
  - Weights probability of value
- Two purposes
  - Asserting evidence
  - Specifying undirected models
- We assume that constraints are bounded by [0, 1]
- Since constraints generalize conditions, we need only consider them going forward

Overview

- Introduction to Probabilistic Programming
- Figaro Language Design
- Semantics
  - Abstracting away Scala functions
  - Semantics of abstracted programs
  - Adding constraints

Abstracting Away Scala Functions

- Scala functions are black boxes
  - Required to terminate and have no side effects
- Figaro with Scala functions abstracted is called SimplePPL
- A SimplePPL program consists of a sequence of definitions of random variables (RVs)
  - An RV r has type T_r
- Each program has a set of *free* RVs
- An RV is *available* in a definition if it is either free in the program or is defined previously in the program

Defining an RV in SimplePPL

An RV r can be defined by one of the following:

- Primitive – defines a probability distribution over values

- $A(r_1, \ldots, r_n, g)$, where each r_i is an available RV and g is a function from $T_{r_1} \times \cdots \times T_{r_n} \rightarrow T_r$

- $C(r_1, g)$, where r_1 is an available RV and g is a function $T_{r_1} \rightarrow Q$, where Q is the space of programs such that
  - The free variables in the program are available variables for r
  - The final RV in the program has type T_r

Converting a Figaro program to SimplePPL

Define a (cached) converter H as follows:

- For a Figaro atomic element, use a SimplePPL primitive

- For a Figaro element $\texttt{Apply}(a_1, \ldots, a_n, f)$
  - For each a_i, create the SimplePPL program $H(a_i)$ with final RV r_i
  - For each $(x_1, \ldots, x_n) \in T_{r_1} \ldots \times T_{r_n}$, evaluate the Scala function $f(x_1, \ldots, x_n)$ to obtain a value x; define $g(x_1, \ldots, x_n) = x$.
  - Add the definition $r = A(r_1, \ldots, r_n, g)$

- For a Figaro element $\texttt{Chain}(a_1, f)$
  - Create the SimplePPL program $H(a_1)$ with final RV r_1
  - For each $x_1 \in T_{r_1}$, evaluate the Scala function $f(x_1)$ to obtain an element e; define $g(x_1) = H(e)$
  - Add the definition $r = C(r_1, g)$

SimplePPL Semantics

- For finite programs, semantics is straightforward
- What about (potentially infinite) recursive programs?
  - We assign a probability to partial assignments of values to prefixes of a program
- Partial expansion includes variables defined up to bounded depth
  - Depth of expansion defined by C form

Partial Assignment

- Intuitively,
  - A partial assignment assigns values to RVs in a partial expansion
  - The special value * is used to indicate “undetermined”A partial assignment must be consistent

- Formally:
- For each RV r in a partial expansion, a partial assignment X assigns a value in T_r or the special value $*$, such that:
  - If r is a primitive, $X(r) \neq *$
  - If r is $A(r_1, \ldots, r_n, g)$ and any $X(r_i) = *$, $X(r) = *$
  - If r is $A(r_1, \ldots, r_n, g)$ and no $X(r_i) = *$, $X(r) = g(X(r_1), \ldots, X(r_n))$
  - If r is $C(r_1, g)$ and $X(r_1) = *$, $X(r) = *$
  - If r is $C(r_1, g)$ and $X(r_1) = v_1$, and $g(v_1)$ is unexpanded, $X(r) = *$
  - If r is $C(r_1, g)$ and $X(r_1) = v_1$, and $g(v_1)$ is expanded, $X(r) = X(r')$, where r' is the final RV in $g(v_1)$

Defining the Probabilities

- Probability of a partial assignment = product of probabilities of assignments to primitives
- Partial assignments correspond to sets of complete assignments
- This process defines a probability measure over space of complete assignments under the σ-algebra generated by partial assignments

Handling Constraints

- For a Figaro constraint f on an element e represented by SimplePPL RV r, we define a SimplePPL constraint g_r by evaluating f on every value in T_r
- For finite programs, we multiply the probability of an assignment by the values of all constraints, then normalize
- For infinite programs, we must work with partial assignments

Probability Bounds

- We will define lower and upper bounds for unnormalized probability
  - For any constraint g_r and value v of r, we define $L_g(v) = U_g(v) = g_r(v)$
  - We define $L_g(*) = 0, U_g(*) = 1$
  - We define the lower probability $L(X)$ of a partial assignment X to be the product of probabilities of primitive assignments times $L_g(X(r))$ for each constraint
  - Similarly for upper probability $U(X)$

- Let $X_1, \ldots, X_n$ be a mutually exclusive and exhaustive set of partial assignments
  - $\frac{L(X_i)}{\sum_{j=1}^{n} U(X_j)}$ is a lower bound on the probability of X_i
  - $\frac{U(X_i)}{\sum_{j=1}^{n} L(X_j)}$ is an upper bound on the probability of X_i

Some Questions

- What is the structure of the space defined by the partial assignments?
  - For pathological programs, the space seems to be trivial
  - In other words, although the probability measure is well-defined, it doesn't say anything interesting
- With constraints, when do the lower and upper probability bounds converge?
- What does a program mean when they don't converge?

More Information

- Figaro is open source
  - Contributions welcome!
  - Releases can be downloaded from www.cra.com/figaro
  - Version 4.0 to be released shortly
- Figaro source is on GitHub at www.github.com/p2t2
- I'm writing a book on "Practical Probabilistic Programming"
  - Publication in the next few weeks
  - www.manning.com/pfeffer
- If you have any questions, feel free to contact me at apfeffer@cra.com