plan

Dhruv Rajan

June 16, 2017

Contents

1 Primitive Constructs and Types **1**
- 1.1 Primitive Types (Figaro) . . . 1
- 1.2 Observing Variables (Figaro) . . . 2
- 1.3 Creating Compound Elements (Figaro) . . . 2
- 1.4 **TODO** Representing Distributions . . . 2
- 1.5 **TODO** Representing Atomic Elements . . . 3
- 1.6 **TODO** Creating Compound Elements . . . 3
- 1.7 **TODO** Simple Inference . . . 3

2 Modeling Dependencies Between Variables **3**
- 2.1 Types of Dependencies (Figaro) . . . 3
- 2.2 Modeling Bayesian Networks . . . 4
  - 2.2.1 **TODO** Structure of Conditional Probability Table . . . 4
  - 2.2.2 **TODO** Representing the Network . . . 4
  - 2.2.3 **TODO** Supporting a Network with Both Discrete and Continuous RVs . . . 5
- 2.3 **TODO** Modeling Markov Networks . . . 5
- 2.4 **TODO** Programming with External State . . . 5
- 2.5 **TODO** Structure of Inference Algorithms . . . 5

1 Primitive Constructs and Types

1.1 Primitive Types (Figaro)

In Figaro, every data structure is an *element*. Every element holds a *value*, with some associated *value type*. Thus, all elements are of the form `Element[type]`. Distinctions are made between four categories of elements:

1. *atomic* self contained, does not depend on another element (`Normal`)
2. *compound* build out of other elements (`If (...), Apply (...)`)
3. *discrete* element whose value type is discrete (`Poisson`)
4. *continuous* element whose value type is continuous (`Gamma`)

1.2 Observing Variables (Figaro)

Elements are defined by distributions of their possible values. A simple `Normal` element can only hold values on the interval [-1, 1], with corresponding probabilities. Thus, each element may be considered a random variable, on its given distribution. Figaro provides methodology for symbolic manipulation of these random variables, allowing dependent variables (compound elements) to be utilized.

No element is given an immediate value until it is observed. The `observe()` method alters only the element on which it is called, so that when an inference algorithm (such as `VariableElimination.probability`) is run on that element, or any variable in the same dependency network, it can calculate the relevant conditional probabilities.

1.3 Creating Compound Elements (Figaro)

Atomic elements can be combined to form compound elements. The simplest method for combining them is the `If` construct, which allows for slightly complex conditioning on distributions. There are also the `Apply` and `Chain`

1.4 TODO Representing Distributions

I've tried to take the core ideas in from the Figaro language, and express them in Haskell. This includes, (1) Representation of Elements / Random Variables, (2) Ability to observe Random Variables (3) Ability to use inference to infer probability distributions, given observations.

I've tried to separate the complexity of combining random variables from the representation of distributions. Thus, a distribution is just a function, mapping some values of some type a to probabilities (Double). I have yet to integrate the standard library of distributions from the statistics library.

```
type Distribution a = a -> Double
```

1.5 TODO Representing Atomic Elements

There is an Element type for representing the structure which correspond to Figaro elements. Atomic elements have two attributes—one for the distribution on which their values are drawn, and one to keep track of the element's observed values.

```
data Element a = Atomic {distribution ::  Distribution a,
                         observed ::  Maybe a}
```

Thus, observations can be performed on atomic elements simply by changing this attribute.

1.6 TODO Creating Compound Elements

The `Element` datatype has separate recursive type constructors for creating compound elements from other elements. `If`, for example, creates an Element from three existing elements. `Apply` and `Chain` have yet to be implemented.

1.7 TODO Simple Inference

The `probability` function contains simple inference logic. Since compound elements are allowed, every element can be considered part of a dependency tree. This function crawls down the tree recursively, calculating the probabilities at each required step. The next steps will include looking into more sophisticated inference algorithms.

2 Modeling Dependencies Between Variables

2.1 Types of Dependencies (Figaro)

Directed and *Undirected* dependencies exist between variables. Directed dependencies imply a clear direction; or, cause-and-effect relationship between two variables. Undirected dependencies represent correlations or observed conditions. This can occur, for example, between two variables that are influenced by the same "confounding" variable.

Bayesian Networks can be used to model directed dependencies, using conditional probability distributions. Markov Networks can be used to model undirected dependencies.

2.2 Modeling Bayesian Networks

A Bayesian Network is a DAG where each node is annotated with probability information, under the following specification.

1. A set of random variables makes up tohe nodes of the network. Variables may be discrete or continuous.
2. A set of directed links or arrows connects pairs of nodes. If there is an Arrow from node X to node Y, X is said to be a parent of Y
3. Each node X_i has a conditional probability distribution $P(X_i|Parents(X_i))$ which quantifies the effect of the parents on the node
4. The graph has no cycles (it is a DAG)

Such a network provides a concise representation of a full joint distribution. Each node is conditionally dependent on its parents, and thus stores some form of a CDT (Conditional Probability Table)

2.2.1 TODO Structure of Conditional Probability Table

This will be a new Haskell datatype, perhaps just something like `[Entry]` Where an entry has multiple fields.

```
data Entry = -- To be defined data CDT = [Entry]
```

Russel & Norvig's book explains how to use noisy logical relationships (noisy-OR) to reduce the space complexity of the tables from $O(2^k)$ to $O(k)$.

2.2.2 TODO Representing the Network

The network is a graph, so it should be a collection of Nodes (call it `[Node]` for now, but much smarter can be done). Each node will contain a `CDT` and two lists of nodes: one for parents, and one for children.

```
data Network = [Node] data Node = Node {getCDT ::  CDT,
    getParents ::  [Node], getChildren ::  [Node]}
```

This structure will *definitely* change as I look into how to build the table by successively adding nodes, since at each step many of the `CDT` entries have to be changed.

Alternatively, could look into using a graph library like FGL, since support for topological algorithms and info will help a lot. For example, determining whether two variables `x` and `y` are conditionally independent of `z`, or finding all variables which are conditionally independent of `z`, etc.

2.2.3 TODO Supporting a Network with Both Discrete and Continuous RVs

Techniques for this are explained in Russel & Norvig

2.3 TODO Modeling Markov Networks

Look at Pfeiffer's book, do more research.

2.4 TODO Programming with External State

In Figaro, the bayesian network is modeled as global state, which is automatically changed when new variables "elements" are created, and queried when inference algorithms are run. This concept is more genral than just Figaro's representation: it begs the question of how to program in Haskell with random variables while updating the external network state properly. It seems that this is a good fit for the State and ST monads.

The current `Element` datatype can become an instance of Monad, since the combinator function will just create a new conditionally dependent distribution (similar to Figaro's Chain function). Then, all computations with elements can be performed within the State monad, and the Bayesian Network's state can be modified accordingly.

2.5 TODO Structure of Inference Algorithms

Any inference algorithm should be just a series of (perhaps complex) computations inside the state monad.