Plan for Language Specification

Dhruv Rajan

June 9, 2017

1 Figaro's Representation

1.1 Primitive Types

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

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

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

1.3.1 If

The type signature for `If` is as follows:

```
If (test:  Element[Boolean],
    then:  Element[T],
    else:  Element[T])
   => cond:  Element[T]
```

This constructor creates an element which follows the corresponding distribution.

1.3.2 Apply

The type signature for `Apply` is as follows:

```
Apply (e1:  Element[T],
       fn:  T -> U)
```

1.3.3 Chain

The type signature for `Chain` is as follows:

```
Chain ((e1:  Element[T],
 fn:  T -> Element[U]))
```

2 Haskell Implementation (src/Probability.hs)

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.

2.1 Representing Distributions

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
```

2.2 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.

2.3 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.

2.4 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.