There's Something About Bayes: Effective Probabilistic Programming for the Rest of Us

James Bornholt Todd Mytkowicz Kathryn S. McKinley
Microsoft Research

Bayes' rule is a fundamental and general mechanism that composes hypotheses and data, which is arguably the purpose of many computer programs, and yet few programming languages or libraries incorporate Bayes' rule as an abstraction. Bayes' rule has several benefits. (1) It provides a formalism for programs to express *composition* of data from multiple sources to improve the program's accuracy, e.g., combining GPS sensor data with road network map data. (2) It provides a technique for programs to implement *personalization*, by composing the original model with user history or preferences. (3) It expresses the potential for programs to *reconsider* decisions or *speculate* about a decision before all the data is available. This position paper explores the challenges involved in making Bayes' rule a programming language operator in general purpose languages. We focus on average developers, who do not have deep knowledge of statistics or machine learning, and on efficiency, since developers find it challenging to reason about the hard-to-predict, potentially unbounded runtime costs of existing implementations of Bayes' rule in inference algorithms.

Uncertainty in Today's Programs

Many modern applications compute with uncertain data from sensors, machine learning algorithms, and other sources. Uncertainty complicates reasoning about a program's behavior, but ignoring uncertainty can cause the program to exhibit strange, unpredictable and difficult-to-reproduce bugs. The machine learning community has evolved techniques for programming with uncertainty using *probabilistic programming languages*, in which program variables are random variables and program statements induce a probabilistic graphical model, queried with backward inference. Since machine learning experts design these languages, they are unsurprisingly highly effective for easing the workload of machine learning experts, and so include a way to express Bayesian inference queries that are equivalent to applying Bayes' rule. Outside the world of machine learning, however, developers increasingly face uncertainty in their own programs, whether from approximation for energy efficiency, deliberately created for differential privacy, or inherent in scientific models or sensor data. It is time for the machine learning gurus to share their secrets and for the programming language community to listen.

Recent work has made progress on bringing probabilistic reasoning to mainstream programming languages without significantly burdening developers. Sankaranarayanan et al. provide static analysis techniques for probabilistic programs, helping developers to reason about the extent to which uncertainty affects their computations [5]. Similarly, Sampson et al. provide probabilistic assertions and a verification tool to help developers reason about the probabilistic properties that their programs generate at runtime [4]. At the language level, Bornholt et al. introduce Uncertain<T>, an abstraction that encapsulates uncertain data and provides an intuitive and accessible semantics for developers to reason under uncertainty [1]. They show the potential for Bayes' rule to improve the accuracy of programs, but do not implement it as a general purpose mechanism.

What this recent work lacks is a general abstraction for Bayes' rule, with which developers compose data from multiple sources to produce more accurate and more compelling results. Bayes' rule tells us the probability of a hypothesis H given evidence E is $\Pr[H|E] \propto \Pr[E|H]\Pr[H]$. To incorporate evidence E into a hypothesis H, we multiply a prior $\Pr[H]$ with a likelihood model $\Pr[E|H]$. At the programming language level, this suggests a Bayes operator $\sharp$ as a multiplication of distributions, such that $\Pr[P \sharp Q] = \Pr[P]\Pr[Q]$ (note that this operation, multiplying two *probabilities*, is distinct from multiplying two *random variables*). Bayes' rule is a fundamental part of any probabilistic formalism for programs. Our position is that the Bayes operator should therefore be a central abstraction for future programming languages, as programmers increasingly face the challenge of computing with uncertain data.

The Power of the Bayes Operator

Composition. The Bayes operator encourages developers to specify their own prior understanding of different possibilities for their data through *composition*. For example, the developer of an in-car navigation application may specify a road network as a prior distribution (i.e., a hypothesis) for the user's location. Since the developer knows this system will be installed in cars, the application can exploit the knowledge that the car is likely to be on a road. The developer would use the Bayes operator $\sharp$ to combine this prior knowledge with GPS evidence to determine the user's location:

```
Uncertain<GeoCoordinate> map = GPS.GetRoadMap("Sydney");
Uncertain<GeoCoordinate> loc = GPS.GetLocation();

Uncertain<GeoCoordinate> new_loc = loc # map;
```

This formulation delivers more accurate location data than the GPS alone. Of course, the beauty of Bayes' rule is that strong GPS evidence can still override the prior knowledge, without requiring developers to resort to ad-hoc heuristics.

Personalization. The Bayes operator also enables *personalization* by providing a framework to incorporate historical data into future predictions. For example, the accuracy of smartphone GPS can be improved by noting that humans are creatures of habit. We tend to revisit places we have visited before. Exploiting this observation means learning a user's location history and applying it via the Bayes operator to future GPS fixes. We could do this learning automatically with machine learning algorithms, but a richer technique is to identify key places in a user's life (home, work, etc.):

```
void GPS.AddFavoriteLocation(GeoCoordinate place) {
  Uncertain<GeoCoordinate> map = GPS.location_history;
  // increment the probability for this place
  map.addToDistribution(place);
}
Uncertain<GeoCoordinate> GPS.LocationWithHistory() {
  Uncertain<GeoCoordinate> loc = GPS.GetLocation();
  return loc # GPS.location_history;
}
```

The system could identify the user's favorite locations from a combination of history and map data (e.g., the user frequently visits a location that the map shows is a gym), calendar data (the user has

frequent meetings in building 99), and explicit user input. Of course, because a user is also quite likely to be visiting a new location, the prior distribution does not assign zero probabilities to non-favorite locations, so GPS evidence can still outweigh historical data.

Speculation. Finally, the Bayes operator also helps developers write code which can automatically *reconsider* decisions in light of new evidence. By preserving a Bayesian network model of computation [1, 4], new data incorporated into a model may prompt a program to reconsider which branch it took at a conditional expression. For example, a gesture recognizer may revise its initial decision about the gesture the user is performing based on where the gesture ends. Being able to reconsider decisions also enables programs to *speculate* about decisions. For example, a program that uses InterPoll [2] to make decisions using crowd-sourced survey data can speculate about the answer to a conditional based on few samples, even if those samples are not sufficient for the desired statistical significance. The program can continue executing while further samples are gathered, reducing the latency of the program if the speculation is correct. If the speculation is incorrect, the program can reconsider its decision and make the other choice.

Concretely, we envision a speculative conditional construct `s_if` as demonstrated in the following program, which processes a user's touch input and decides which button was pressed and thus which web page to load:

```
Uncertain<bool> HitTest(Uncertain<Pixel> touch,
                        TouchTarget target) {
  return new Uncertain<bool>(
      () => target.Contains(touch.Sample()) );
}

void OnTouch(Uncertain<Pixel> touch) {
  WebPage page;
  s_if (HitTest(touch, PageOneButton)) {
    page = fetchPage(PageOneURL);
    s_wait;
  } s_elseif (HitTest(touch, PageTwoButton)) {
    page = fetchPage(PageTwoURL);
    s_wait;
  } s_else {
    return; // no target was hit
  }
  presentPage(page);
}
```

The `s_if` construct tells the runtime that it can speculate on the result of the conditional expression to decide which branch to enter. The semantics of the conditional expression itself are as provided by Uncertain<T>, so ask whether the hit test is more likely than not to be true. However, the runtime is now allowed to speculate, by observing only a few samples from the hit test and using them to choose a branch while continuing the sampling in the background. If later samples indicate the conclusion was wrong (for example, the user drags their finger off the button during the gesture), the runtime aborts the branch and tries the `s_elseif` condition. But if the speculation was correct, the program began loading the right web page before the gesture was complete, reducing the visible latency. The `s_wait` keyword tells the runtime that it must pause here until it finalizes the conclusion it speculated on, so that the program does not continue until the speculation is validated or disproven. We believe this construct could be implemented as a compiler pass, and the control flow achieved through the use of exceptions and try/catch blocks in a host language such as Java or C#.

Now Make It Fast

The Bayes operator is difficult to implement in a system that represents distributions by random sampling (which most do), because there is no obvious way to compute the probabilities $\Pr[P]$ and $\Pr[Q]$ that need to be multiplied. In particular, a completely general implementation requires expensive backward inference algorithms—those very same algorithms the machine learning community use in their probabilistic programming languages, and which require statistics expertise and acceptance of potentially-unbounded runtime costs to use. Despite this difficulty, random sampling is a desirable representation for a number of reasons surrounding expressiveness and efficiency [1, 3].

To overcome this problem, the programming language community must find ways to specialize the Bayes operator to cases that are efficient and useful in average programs. For example, applying the Bayes operator to two Bernoulli operands requires only taking the conjunction of a sample from each operand, and so imposes practically no overhead. We think many of these simpler cases will be useful in practice.

To tackle the efficiency of the Bayes operator, we suggest a set of performance optimizations that operate on a program's probabilistic semantics, rather than its concrete one. Recent work demonstrates how to lift a program from its concrete semantics to a probabilistic one and optimize it [4]. This work extends ideas such as constant folding to probabilistic models to reduce an expression subtree to a single node when the optimizer statically knows the distribution of the result. Implementing such an optimization requires the compiler to understand rules of statistics, similar to the rules compilers already understand about common patterns in programs. We suggest these same ideas will help when making the Bayes operator *efficient*.

A Call to Action

The benefits of a Bayes operator are clear. Programs written using the operator are more accurate than those without, and developers can use the operator to introduce new, more compelling features to their programs. Only recently has the programming language community been able to put elements of probabilistic programming within the reach of the non-expert developer. We will continue to develop new examples showing the power of correctly reasoning about uncertain data using these elements. The burden now lies on the community to deliver the abstractions and optimizations necessary for non-expert developers to reason usefully and efficiently in this way. If we rise to this challenge, programs that reason about probabilistic data will be easier to write, more accurate, and more effective.

References

[1] J. Bornholt, T. Mytkowicz, and K. S. McKinley. Uncertain<T>: A First-order Type for Uncertain Data. In *ASPLOS 2014*.

[2] B. Livshits and T. Mytkowicz. InterPoll: Crowd-Sourced Internet Polls (Done Right). Technical Report MSR-TR-2014-3, Microsoft Research.

[3] S. Park, F. Pfenning, and S. Thrun. A probabilistic language based on sampling functions. In *POPL 2005*.

[4] A. Sampson, P. Panchekha, T. Mytkowicz, K. S. McKinley, D. Grossman, and L. Ceze. Expressing and verifying probabilistic assertions. In *PLDI 2014*. To appear.

[5] S. Sankaranarayanan, A. Chakarov, and S. Gulwani. Static analysis for probabilistic programs: Inferring whole program properties from finitely many paths. In *PLDI 2013*.