Uncertain<*T*>: A First-Order Type for Uncertain Data

James Bornholt
Australian National University
u4842199@anu.edu.au

Todd Mytkowicz
Microsoft Research
toddm@microsoft.com

Kathryn S. McKinley
Microsoft Research
mckinley@microsoft.com

Abstract

Emerging applications increasingly use estimates such as sensor data (GPS), probabilistic models, machine learning, big data, and human data. Unfortunately, representing this *uncertain data* with discrete types (floats, integers, and booleans) encourages developers to pretend it is not probabilistic, which causes three types of *uncertainty bugs*. (1) Using estimates as facts ignores random error in estimates. (2) Computation compounds that error. (3) Boolean questions on probabilistic data induce false positives and negatives.

This paper introduces *Uncertain*$\langle T\rangle$, a new programming language abstraction for uncertain data. We implement a Bayesian network semantics for computation and conditionals that improves program correctness. The runtime uses sampling and hypothesis tests to evaluate computation and conditionals lazily and efficiently. We illustrate with sensor and machine learning applications that *Uncertain*$\langle T\rangle$ improves expressiveness and accuracy.

Whereas previous probabilistic programming languages focus on experts, *Uncertain*$\langle T\rangle$ serves a wide range of developers. Experts still identify error distributions. However, both experts and application writers compute with distributions, improve estimates with domain knowledge, and ask questions with conditionals. The *Uncertain*$\langle T\rangle$ type system and operators encourage developers to expose and reason about uncertainty explicitly, controlling false positives and false negatives. These benefits make *Uncertain*$\langle T\rangle$ a compelling programming model for modern applications facing the challenge of uncertainty.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Data types and structures

Keywords Estimates; Statistics; Probabilistic Programming

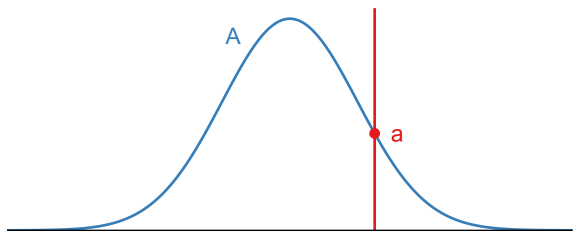


Figure 1. The probability distribution *A* quantifies potential errors. Sampling *A* produces a single point *a*, introducing uncertainty, but *a* does not necessarily represent the true value. Many programs treat *a* as the true value.

1. Introduction

Applications that sense and reason about the complexity of the world use estimates. Mobile phone applications estimate location with GPS sensors, search estimates information needs from search terms, machine learning estimates hidden parameters from data, and approximate hardware estimates precise hardware to improve energy efficiency. The difference between an estimate and its true value is *uncertainty*. Every estimate has uncertainty due to random or systematic error. Random variables model uncertainty with probability distributions, which assign a probability to each possible value. For example, each flip of a biased coin may have a 90% chance of heads and 10% chance of tails. The outcome of one flip is only a sample and not a good estimate of the true value. Figure 1 shows a sample from a Gaussian distribution which is a poor approximation for the entire distribution.

Most programming languages force developers to reason about uncertain data with discrete types (floats, integers, and booleans). Motivated application developers reason about uncertainty in ad hoc ways, but because this task is complex, many more simply ignore uncertainty. For instance, we surveyed 100 popular smartphone applications that use GPS and find only one (Pizza Hut) reasons about the error in GPS measurements. Ignoring uncertainty creates three types of *uncertainty bugs* which developers need help to avoid:

Using estimates as facts ignores random noise in data and introduces errors.

Computation compounds errors since computations on uncertain data often degrade accuracy significantly.

- ***Conditionals ask boolean questions*** of probabilistic data, leading to false positives and false negatives.

While probabilistic programming [6, 13, 15, 24, 25] and domain-specific solutions [1, 2, 11, 18, 28–30] address parts of this problem, they demand expertise far beyond what client applications require. For example in current probabilistic programming languages, domain experts create and query distributions through generative models. Current APIs for estimated data from these programs, sensors, big data, machine learning, and other sources then project the resulting distributions into discrete types. We observe that the probabilistic nature of estimated data does not stop at the API boundary. Applications using estimated data are probabilistic programs too! Existing languages do not consider the needs of applications that consume estimated data, leaving their developers to face this difficult problem unaided.

This paper introduces the *uncertain type*, $Uncertain\langle T\rangle$, a programming language abstraction for arbitrary probability distributions. The syntax and semantics emphasize simplicity for non-experts. We describe how expert developers derive and expose probability distributions for estimated data. Similar to probabilistic programming, the uncertain type defines an algebra over random variables to propagate uncertainty through calculations. We introduce a Bayesian network semantics for computations and conditional expressions. Instead of eagerly evaluating probabilistic computations as in prior languages, we lazily evaluate *evidence* for the conditions. Finally, we show how the uncertain type eases the use of prior knowledge to improve estimates.

Our novel implementation strategy performs lazy evaluation by exploiting the semantics of conditionals. The $Uncertain\langle T\rangle$ runtime creates a Bayesian network that represents computations on distributions and then samples it at conditional expressions. A sample executes the computations in the network. The runtime exploits hypothesis tests to take only as many samples as necessary for the particular conditional, rather than eagerly and exhaustively producing unnecessary precision (as in general inference over generative models). These hypothesis tests both guarantee accuracy bounds and provide high performance.

We demonstrate these claims with three case studies. (1) We show how $Uncertain\langle T\rangle$ improves accuracy and expressiveness of speed computations from GPS, a widely used hardware sensor. (2) We show how $Uncertain\langle T\rangle$ exploits prior knowledge to minimize random noise in digital sensors. (3) We show how $Uncertain\langle T\rangle$ encourages developers to explicitly reason about and improve accuracy in machine learning, using a neural network that approximates hardware [12, 26]. In concert, the syntax, semantics, and case studies illustrate that $Uncertain\langle T\rangle$ eases probabilistic reasoning, improves estimates, and helps domain experts and developers work with uncertain data.

Our contributions are (1) characterizing uncertainty bugs; (2) $Uncertain\langle T\rangle$, an abstraction and semantics for uncertain data; (3) implementation strategies that make this semantics practical; and (4) case studies that show $Uncertain\langle T\rangle$'s potential to improve expressiveness and correctness.

2. Motivation

Modern and emerging applications compute over uncertain data from mobile sensors, search, vision, medical trials, benchmarking, chemical simulations, and human surveys. Characterizing uncertainty in these data sources requires domain expertise, but non-expert developers (perhaps with other expertise) are increasingly consuming the results. This section uses Global Positioning System (GPS) data to motivate a correct and accessible abstraction for uncertain data.

On mobile devices, GPS sensors estimate location. APIs for GPS typically include a position and estimated error radius (a confidence interval for location). The Windows Phone (WP) API returns three fields:

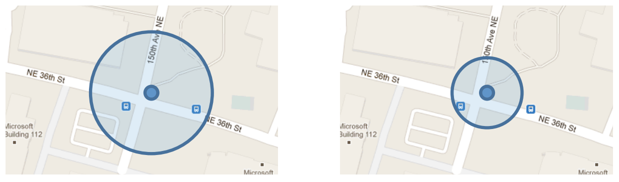
```
public double Latitude, Longitude;   // location
public double HorizontalAccuracy;    // error estimate
```

This interface encourages three types of *uncertainty bugs*.

Interpreting Estimates as Facts Our survey of the top 100 WP and 100 Android applications finds 22% of WP and 40% of Android applications use GPS for location. Only 5% of the WP applications that use GPS read the error radius and only one application (Pizza Hut) acts on it. All others treat the GPS reading as a fact. Ignoring uncertainty this way causes errors such as walking through walls or driving on water.

Current abstractions encourage this treatment by obscuring uncertainty. Consider the map applications on two different smartphone operating systems in Figure 2, which depict location with a point and horizontal accuracy as a circle. Smaller circles *should* indicate less uncertainty, but the left larger circle is a 95% confidence interval (widely used for statistical confidence), whereas the right is a 68% confidence interval (one standard deviation of a Gaussian). The smaller circle has a higher standard deviation and is less accurate! (We reverse engineered this confidence interval detail.) A single accuracy number is insufficient to characterize the underlying error distribution or to compute on it. The horizontal accuracy abstraction obscures the true uncertainty, encouraging developers to ignore it completely.

Compounding Error Computation compounds uncertainty. To illustrate, we recorded GPS locations on WP while walk-



(a) 95% CI, $\sigma = 33$ m **(b)** 68% CI, $\sigma = 39$ m

Figure 2. GPS samples at the same location on two smartphone platforms. Although smaller circles *appear* more accurate, the WP sample in (a) is actually more accurate.

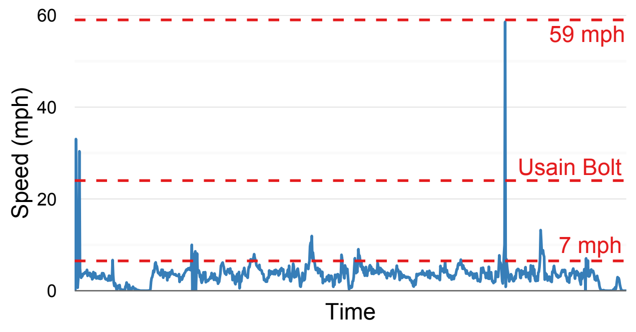


Figure 3. Speed computation on GPS data produces absurd walking speeds (59 mph, and 7 mph for 35 s, a running pace).

ing and computed speed each second. Our inspection of smartphone applications shows this computation on GPS data is very common. Figure 3 plots speed computed by the code in Figure 5(a) using the standard GPS API. Whereas Usain Bolt runs 100 m at 24 mph, the average human walks at 3 mph. This experimental data shows an average of 3.5 mph, 35 s spent above 7 mph (running speed), and absurd speeds of 30, 33, and 59 mph. These errors are significant in both magnitude and frequency. The cause is compounded error, since speed is a function of *two* uncertain locations. When the locations have a 95% confidence interval of 4 m (the best that smartphone GPS delivers), speed has a 95% confidence interval of 12.7 mph. Current abstractions do not capture the compounding of error because they do not represent the distribution nor propagate uncertainty through calculations.

Conditionals Programs eventually act on estimated data with conditionals. Consider using GPS to issue tickets for a 60 mph speed limit with the conditional $Speed > 60$. If your actual speed is 57 mph and GPS accuracy is 4 m, this conditional gives a 32% probability of a ticket due to random noise alone. Figure 4 shows this probability across speeds and GPS accuracies. Boolean questions ignore the potential for random error, leading to false positives and negatives. Applications instead should ask probabilistic questions; for example, only issuing a ticket if the probability is very high that the user is speeding.

Without an appropriate abstraction for uncertain data, propagation of errors through computations, and probabilistic semantics for conditionals, correctness is out of reach for many developers.

3. A First-Order Type for Uncertain Data

We propose a new generic data type *Uncertain*$\langle T\rangle$ and operations to capture and manipulate uncertain data as probability distributions. The operations propagate distributions through computations and conditionals with an intuitive semantics that help developers program with uncertain data. The *uncertain type* programming language abstraction is broadly applicable to many languages and we implemented prototypes in C#, C++, and Python. Unlike existing probabilistic programming approaches, which focus on statistics and machine learning experts, *Uncertain*$\langle T\rangle$ focuses on creating an accessible interface for non-expert developers, who are increasingly encountering uncertain data.

This section first overviews *Uncertain*$\langle T\rangle$'s syntax and probabilistic semantics and presents an example. It then describes the syntax and semantics for specifying distributions, computing with distributions by building a Bayesian network representation of computations, and executing conditional expressions by evaluating evidence for a conclusion. We then describe how *Uncertain*$\langle T\rangle$ makes it easier for developers to improve estimates with domain knowledge. Section 4 describes our lazy evaluation and sampling implementation strategies that make these semantics efficient.

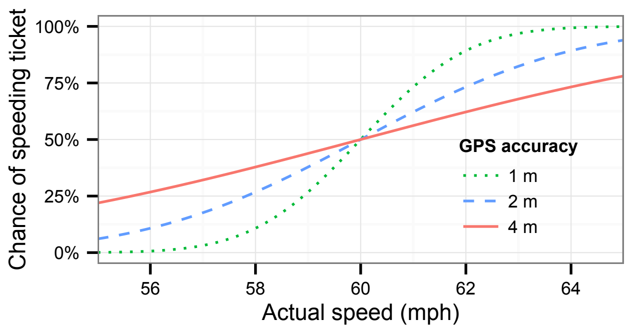
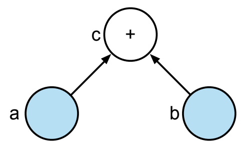


Figure 4. Probability of issuing a speeding ticket at a 60 mph speed limit. With a true speed of 57 mph and GPS accuracy of 4 m, there is a 32% chance of issuing a ticket.

Overview An object of type *Uncertain*$\langle T\rangle$ encapsulates a random variable of a numeric type T. To represent computations, the type's overloaded operators construct *Bayesian networks*, directed acyclic graphs in which nodes represent random variables and edges represent conditional dependences between variables. The leaf nodes of these Bayesian networks are known distributions defined by expert developers. Inner nodes represent the sequence of operations that compute on these leaves. For example, the following code

```
Uncertain<double> a = new Gaussian(4, 1);
Uncertain<double> b = new Gaussian(5, 1);
Uncertain<double> c = a + b;
```

results in a simple Bayesian network



with two leaf nodes (shaded) and one inner node (white) representing the computation `c = a + b`. *Uncertain*$\langle T\rangle$ evaluates this Bayesian network when it needs the distribution of `c`, which depends on the distributions of `a` and `b`. Here the distribution of `c` is more uncertain than `a` or `b`, as Figure 6 shows, since computation compounds uncertainty.

This paper describes a runtime that builds Bayesian networks dynamically and then, much like a JIT, compiles those

```
double dt = 5.0;  // seconds
GeoCoordinate L1 = GPS.GetLocation();

while (true) {
  Sleep(dt);  // wait for dt seconds
  GeoCoordinate L2 = GPS.GetLocation();
  double Distance = GPS.Distance(L2, L1);
  double Speed = Distance / dt;
  print("Speed: " + Speed);
  if (Speed > 4) GoodJob();
  else           SpeedUp();
  L1 = L2; // Last Location = Current Location;
}
```

(a) Without the uncertain type

```
double dt = 5.0;  // seconds
Uncertain<GeoCoordinate> L1 = GPS.GetLocation();

while (true) {
  Sleep(dt);  // wait for dt seconds
  Uncertain<GeoCoordinate> L2 = GPS.GetLocation();
  Uncertain<double> Distance = GPS.Distance(L2, L1);
  Uncertain<double> Speed = Distance / dt;
  print("Speed: " + Speed.E());
  if (Speed > 4)                GoodJob();
  else if ((Speed < 4).Pr(0.9)) SpeedUp();
  L1 = L2;  // Last Location = Current Location;
}
```

(b) With the uncertain type

Figure 5. A simple fitness application (GPS-Walking), encouraging users to walk faster than 4 mph, implemented with and without the uncertain type. The type `GeoCoordinate` is a pair of `double`s (latitude and longitude) and so is numeric.

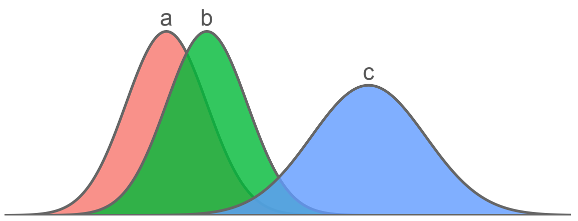


Figure 6. The sum $c = a + b$ is more uncertain than a or b.

Operators	
Math ($+ - * /$)	$op :: U\langle T\rangle \to U\langle T\rangle \to U\langle T\rangle$
Order ($< > \leq \geq$)	$op :: U\langle T\rangle \to U\langle T\rangle \to U\langle Bool\rangle$
Logical ($\wedge \vee$)	$op :: U\langle Bool\rangle \to U\langle Bool\rangle \to U\langle Bool\rangle$
Unary ($\neg$)	$op :: U\langle Bool\rangle \to U\langle Bool\rangle$
Point-mass	$Pointmass :: T \to U\langle T\rangle$
Conditionals	
Explicit	$Pr :: U\langle Bool\rangle \to [0,1] \to Bool$
Implicit	$Pr :: U\langle Bool\rangle \to Bool$
Evaluation	
Expected value	$E :: U\langle T\rangle \to T$
$U\langle T\rangle$ is shorthand for $Uncertain\langle T\rangle$.	

Table 1. $Uncertain\langle T\rangle$ operators and methods.

expression trees to executable code at conditionals. The semantics establishes hypothesis tests at conditionals. The runtime samples by repeatedly executing the compiled code until the test is satisfied.

Example Program Figure 5 shows a simple fitness application (GPS-Walking) written in C#, both without (left) and with (right) $Uncertain\langle T\rangle$. GPS-Walking encourages users to walk faster than 4 mph. The $Uncertain\langle T\rangle$ version produces more accurate results in part because the type encourages the developer to reason about false positives and negatives. In particular, the developer chooses not to nag, admonishing users to `SpeedUp` only when it is very confident they are walking slowly. As in traditional languages, $Uncertain\langle T\rangle$ only executes one side of conditional branches. Some probabilistic languages execute both sides of conditional branches to create probabilistic models, but $Uncertain\langle T\rangle$ makes concrete decisions at conditional branches, matching the host language semantics. We demonstrate the improved accuracy of the $Uncertain\langle T\rangle$ version of GPS-Walking in Section 5.1.

This example serves as a good pedagogical tool because of its simplicity, but the original embodies a real-world uncertainty problem because *many* smartphone applications on all major platforms use the GPS API exactly this way. $Uncertain\langle T\rangle$'s simple syntax and semantics result in very few changes to the program: the developer only changes the variable types and the conditional operators.

3.1 Syntax with Operator Overloading

Table 1 shows the operators and methods of $Uncertain\langle T\rangle$. $Uncertain\langle T\rangle$ defines an algebra over random variables to propagate uncertainty through computations, overloading the usual arithmetic operators from the base numeric type T. Developers may override other types as well. Developers compute with $Uncertain\langle T\rangle$ as they would with T, and the Bayesian network the operators construct captures how error in an estimate flows through computations.

$Uncertain\langle T\rangle$ addresses two sources of random error: *domain* error and *approximation* error. Domain error motivates our work and is the difference between an estimate and its true value. Approximation error is created because $Uncertain\langle T\rangle$ must approximate distributions. Developers ultimately make concrete decisions on uncertain data through conditionals. $Uncertain\langle T\rangle$'s conditional operators enforce statistical tests at conditionals, mitigating both sources of error.

3.2 Identifying Distributions

The underlying probability distribution for uncertain data is specific to the problem domain. In many cases expert library developers already know these distributions, and sometimes they use them to produce crude error estimates such as the GPS horizontal accuracy discussed in Section 2. $Uncertain\langle T\rangle$ offers these expert developers an abstraction to expose these distributions while preserving the simplicity of their current API. The non-expert developers who consume this uncertain data program against the common $Uncertain\langle T\rangle$ API, which is very similar to the way they already program with uncertain data today, but aids them in avoiding uncertainty bugs.

The expert developer has two broad approaches for selecting the right distribution for their particular problem.

(1) Selecting a theoretical model. Many sources of uncertain data are amenable to theoretical models which library writers may adopt. For example, the error in the mean of a data set is approximately Gaussian by the Central Limit Theorem. Section 5.1 uses a theoretical approach for the GPS-Walking case study.

(2) Deriving an empirical model. Some problems do not have or are not amenable to theoretical models. For these cases, expert developers may determine an error distribution empirically by machine learning or other mechanisms. Section 5.3 uses an empirical approach for machine learning.

Representing Distributions

There are a number of ways to store probability distributions. The most accurate mechanism stores the probability density function exactly. For finite domains, a simple map can assign a probability to each possible value [30]. For continuous domains, one might reason about the density function algebraically [4]. For example, a Gaussian random variable with mean μ and variance σ^2 has density function

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left\{-\frac{(x-\mu)^2}{2\sigma^2}\right\}.$$

An instance of a Gaussian random variable need only store this formula and values of μ and σ^2 to represent the variable exactly (up to floating point error).

Exact representation has two major downsides. First, the algebra quickly becomes impractical under computation: even the sum of two distributions requires evaluating a convoluted *convolution integral*. Second, many important distributions for sensors, road maps, approximate hardware, and machine learning do not have closed-form density functions and so cannot be stored this way.

To overcome these issues, *Uncertain*$\langle T\rangle$ represents distributions through approximate *sampling functions*. Approximation can be arbitrarily accurate given sufficient space and time [31], leading to an efficiency-accuracy trade-off. Many possible approximation schemes might be appropriate for *Uncertain*$\langle T\rangle$, including fixed vectors of random samples, Chebyshev polynomials [16], or sampling functions [23]. We use sampling functions because they implement a principled solution for balancing accuracy and efficiency.

3.3 Computing with Distributions

Developers combine uncertain data by using the usual operators for arithmetic, logic, and comparison, and *Uncertain*$\langle T\rangle$ manages the resultant uncertainty. *Uncertain*$\langle T\rangle$ overloads such operators from the base type T to work over distributions. For example, the example program in Figure 5(b) calculates the user's speed by division:

```
    Uncertain<double> Speed = Distance / dt;
```

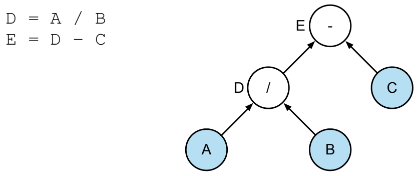


Figure 7. Bayesian network for a simple program with independent leaves.

The type *Double* has a division operator of type

$$/ :: Double \to Double \to Double.$$

The type *Uncertain*$\langle Double\rangle$ lifts this operator to work over random variables, providing a new operator

$$/ :: U\langle Double\rangle \to U\langle Double\rangle \to U\langle Double\rangle.$$

Note given x of type T, the semantics coerces x to type *Uncertain*$\langle T\rangle$ with a pointmass distribution centered at x, so the denominator `dt` is cast to type *Uncertain*$\langle Double\rangle$. The same lifting occurs on other arithmetic operators of type $T \to T \to T$, as well as comparison operators (type $T \to T \to Bool$) and logical operators (type $Bool \to Bool \to Bool$). A lifted operator may have any type. For example, we can define real division of integers as type $Int \to Int \to Double$, which *Uncertain*$\langle T\rangle$ lifts without issue.

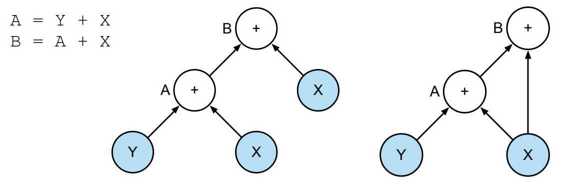
Instead of executing operations instantaneously, the lifted operators construct Bayesian network representations of the computations. A Bayesian network is a *probabilistic graphical model* and is a directed acyclic graph whose nodes represent random variables and whose edges represent conditional dependences between those variables [5]. Figure 7 shows an example of this Bayesian network representation. The shaded leaf nodes are known distributions (such as Gaussians) specified by expert developers, as previously noted. The final Bayesian network defines a joint distribution over all the variables involved in the computation:

$$\Pr[A,B,C,D,E] = \Pr[A]\Pr[B]\Pr[C]\Pr[D \mid A,B]\Pr[E \mid C,D].$$

The incoming edges to a node in the Bayesian network graph specify the other variables that the node's variable depends on. For example, `A` has no dependences while `E` depends on `C` and `D`. Since we know the distributions of the leaf nodes `A`, `B`, and `C`, we use the joint distribution to infer the marginal distributions of the variables `D` and `E`, even though their distributions are not explicitly specified by the program. The conditional distributions of inner nodes are specified by their associated operators. For example, the distribution of $\Pr[D \mid A = a, B = b]$ in Figure 7 is simply a pointmass at a/b.

Dependent Random Variables

Two random variables X and Y are independent if the value of one has no bearing on the value of the other. *Uncertain*$\langle T\rangle$'s Bayesian network representation assumes that leaf nodes are independent. This assumption is common in probabilistic programming, but expert developers can override it by specifying the joint distribution between two variables.


```
A = Y + X
B = A + X
```

(a) Wrong network **(b)** Correct network

Figure 8. Bayesian networks for a simple program with dependent leaves.

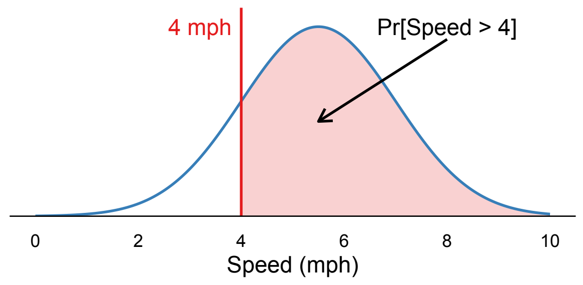


Figure 9. Uncertainty in data means there is only a *probability* that *Speed* > 4, not a concrete boolean value.

The *Uncertain*$\langle T \rangle$ semantics automatically addresses program-induced dependences. For example, Figure 8(a) shows a simple program with a naive incorrect construction of its Bayesian network. This network implies that the operands to the addition that defines `B` are independent, but in fact both operands depend on the same variable `X`. When producing a Bayesian network, *Uncertain*$\langle T \rangle$'s operators echo static single assignment by determining that the two `X` occurrences refer to the same value, and so `A` depends on the same `X` as `B`. Our analysis produces the correct Bayesian network in Figure 8(b). Because the Bayesian network is constructed dynamically and incrementally during program execution, the resulting graph remains acyclic.

3.4 Asking the Right Questions

After computing with uncertain data, programs use it in conditional expressions to make decisions. The example program in Figure 5 compares the user's speed to 4 mph. Of course, since speed is computed with uncertain estimates of location, it too is uncertain. The naive conditional *Speed* > 4 incorrectly asks a deterministic question of probabilistic data, creating uncertainty bugs described in Section 2.

Uncertain$\langle T \rangle$ defines the semantics of conditional expressions involving uncertain data by computing *evidence* for a conclusion. This semantics encourages developers to ask appropriate questions of probabilistic data. Rather than asking "is the user's speed faster than 4 mph?" *Uncertain*$\langle T \rangle$ asks "how much evidence is there that the user's speed is faster than 4 mph?" The evidence that the user's speed is faster than 4 mph is the quantity $\Pr[\mathit{Speed} > 4]$; in Figure 9, this is the area under the distribution of the variable `Speed` to the right of 4 mph. Since we are considering probability distributions, the area A satisfies $0 \leq A \leq 1$.

When the developer writes a conditional expression

```
if (Speed > 4) ...
```

the program applies the lifted version of the $>$ operator

$$> :: \mathit{Uncertain}\langle T \rangle \rightarrow \mathit{Uncertain}\langle T \rangle \rightarrow \mathit{Uncertain}\langle \mathit{Bool} \rangle.$$

This operator creates a Bernoulli distribution with parameter $p \in [0,1]$, which by definition is the probability that *Speed* > 4 (i.e., the shaded area under the curve). Unlike other probabilistic programming languages, *Uncertain*$\langle T \rangle$ executes only a single branch of the conditional to match the host language's semantics. The *Uncertain*$\langle T \rangle$ runtime must therefore convert this Bernoulli distribution to a concrete boolean value. This conditional uses the implicit conditional operator, which compares the parameter p of the Bernoulli distribution to 0.5, asking whether the Bernoulli is more likely than not to be true. This conditional therefore evaluates whether $\Pr[\mathit{Speed} > 4] > 0.5$, asking whether it is more likely that the user's speed is faster than 4 mph.

Using an explicit conditional operator, developers may specify a threshold to compare against. The second comparison in Figure 5(b) uses this explicit operator:

```
else if ((Speed < 4).Pr(0.9)) ...
```

This conditional evaluates whether $\Pr[\mathit{Speed} < 4] > 0.9$. The power of this formulation is reasoning about false positives and negatives. Even if the mean of the distribution is on one side of the conditional threshold, the distribution may be very wide, so there is still a strong likelihood that the opposite conclusion is correct. Higher thresholds for the explicit operator require stronger evidence, and produce fewer false positives (extra reports when ground truth is false) but more false negatives (missed reports when ground truth is true). In this case, the developer chooses to favor some false positives for encouraging users (`GoodJob`), and to limit false positives when admonishing users (`SpeedUp`) by demanding stronger evidence that they are walking slower than 4 mph.

Hypothesis Testing for Approximation Error

Because *Uncertain*$\langle T \rangle$ approximates distributions, we must consider approximation error in conditional expressions. We use statistical hypothesis tests, which make inferences about population statistics based on sampled data. *Uncertain*$\langle T \rangle$ establishes a hypothesis test when evaluating the implicit and explicit conditional operators. In the implicit case above, the null hypothesis is $H_0 : \Pr[\mathit{Speed} > 4] \leq 0.5$ and the alternate hypothesis $H_A : \Pr[\mathit{Speed} > 4] > 0.5$. Section 4.3 describes our sampling process for evaluating these tests in detail.

Hypothesis tests introduce a ternary logic. For example, given the code sequence

```
if (A < B) ...
else if (A >= B) ...
```

neither branch may be true because the runtime may not be able to reject the null hypothesis for either conditional at the required confidence level. This behavior is not new: just

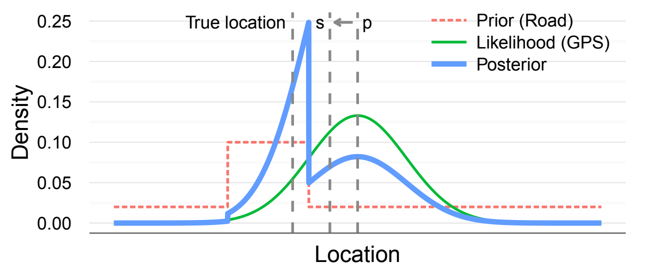


Figure 10. Domain knowledge as a prior distribution improves the quality of GPS estimates.

as programs should not compare floating point numbers for equality, neither should they compare distributions for equality. However, for problems that *require* a total order, such as sorting algorithms, *Uncertain*$\langle T\rangle$ provides the expected value operator E. This operator outputs an element of type T and so it preserves the base type's ordering properties.

3.5 Improving Estimates

Bayesian statistics represents the state of the world with degrees of belief and is a powerful mechanism for improving the quality of estimated data. Given two random variables, B representing the target variable to estimate (e.g., the user's location), and E the estimation process (e.g., the GPS sensor output), Bayes' theorem says that

$$\Pr[B=b|E=e] = \frac{\Pr[E=e|B=b]\cdot\Pr[B=b]}{\Pr[E=e]}.$$

Bayes' theorem combines evidence from estimation processes (the value e of E) with hypotheses about the true value b of the target variable B. We call $\Pr[B=b]$ the *prior* distribution, our belief about B before observing any evidence, and $\Pr[B=b|E=e]$ the *posterior* distribution, our belief about B after observing evidence.

Uncertain$\langle T\rangle$ unlocks Bayesian statistics by encapsulating entire data distributions. Abstractions that capture only single point estimates are insufficient for this purpose and force developers to resort to ad-hoc heuristics to improve estimates.

Incorporating Knowledge with Priors

Bayesian inference is powerful because developers can encode their domain knowledge as prior distributions, and use that knowledge to improve the quality of estimates. For example, a developer working with GPS can provide a prior distribution that assigns high probabilities to roads and lower probabilities elsewhere. This prior distribution achieves a "road-snapping" behavior [21], fixing the user's location to nearby roads unless GPS evidence to the contrary is very strong. Figure 10 illustrates this example – the mean shifts from p to s, closer to the road the user is actually on.

Specifying priors, however, requires a knowledge of statistics beyond the scope of most developers. In our current implementation, applications specify domain knowledge with *constraint abstractions*. Expert developers add preset prior distributions to their libraries for common cases. For example, GPS libraries would include priors for driving (e.g., roads and driving speeds), walking (walking speeds), and being on land. Developers combine priors through flags that select constraints specific to their application. The library applies the selected priors, improving the quality of estimates without much burden on developers. This approach is not very satisfying because it is not compositional, so the application cannot easily mix and match priors from different sources (e.g., maps, calendars, and physics for GPS). We anticipate future work creating an accessible and compositional abstraction for prior distributions, perhaps with the Bayes operator $P\sharp Q$ for sampled distributions by Park et al. [23].

4. Implementation

This section describes our practical and efficient C# implementation of the *Uncertain*$\langle T\rangle$ abstraction. We also implemented prototypes of *Uncertain*$\langle T\rangle$ in C++ and Python and believe most high-level languages could easily implement it.

4.1 Identifying Distributions

Uncertain$\langle T\rangle$ approximates distributions with *sampling functions*, rather than storing them exactly, to achieve expressiveness and efficiency. A sampling function has no arguments and returns a new random sample, drawn from the distribution, on each invocation [23]. For example, a pseudo-random number generator is a sampling function for the uniform distribution, and the Box-Mueller transform [8] is a sampling function for the Gaussian distribution.

In the GPS-Walking application in Figure 5(b), the variables `L1` and `L2` are distributions obtained from the GPS library. The expert developer who implements the GPS library derives the correct distribution and provides it to *Uncertain*$\langle T\rangle$ as a sampling function. Bornholt [7] shows in detail the derivation for the error distribution of GPS data. The resulting model says that the posterior distribution for a GPS estimate is

$$\begin{aligned}&\Pr[\mathbf{Location}=p\,|\,\mathbf{GPS}=\mathit{Sample}]\\&\quad=\text{Rayleigh}(\|\mathit{Sample}-p\|\,;\varepsilon/\sqrt{\ln 400})\end{aligned}$$

where *Sample* is the raw GPS sample from the sensor, ε is the sensor's estimate of the 95% confidence interval for the location (i.e., the horizontal accuracy from Section 2), and the Rayleigh distribution [22] is a continuous non-negative single-parameter distribution with density function

$$\text{Rayleigh}(x;\rho) = \frac{x}{\rho^2}\exp\left\{-\frac{x^2}{2\rho^2}\right\},\quad x\geq 0.$$

Figure 11 shows the posterior distribution given a particular value of ε. Most existing GPS libraries return a coordinate (the center of the distribution) and present ε as a confidence parameter most developers ignore. Figure 11 shows that the true location is *unlikely* to be in the center of the distribution and more likely to be some fixed radius from the center.

We built a GPS library that captures error in its estimate with *Uncertain*$\langle T\rangle$ using this distribution. Figure 12 shows our library function `GPS.GetLocation`, which returns an instance of *Uncertain*$\langle$*GeoCoordinate*$\rangle$ by implementing a

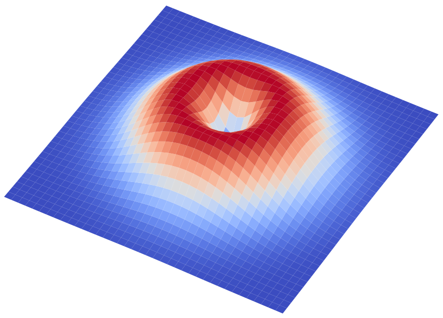


Figure 11. The posterior distribution for GPS is a distribution over the Earth's surface.

```
Uncertain<GeoCoordinate> GetLocation() {
    // Get the estimates from the hardware
    GeoCoordinate Point = GPS.GetHardwareLocation();
    double Accuracy = GPS.GetHardwareAccuracy();
    // Compute epsilon
    double epsilon = Accuracy / Math.Sqrt(Math.Log(400));
    // Define the sampling function
    Func<GeoCoordinate> SamplingFunction = () => {
        double radius, angle, x, y;
        // Samples the distribution in Figure 11 using
        // polar coordinates
        radius = Math.RandomRayleigh(epsilon);
        angle  = Math.RandomUniform(0, 2*Math.PI);
        // Convert to x,y coordinates in degrees
        x = Point.Longitude;
        x += radius*Math.Cos(angle)*DEGREES_PER_METER;
        y = Point.Latitude;
        y += radius*Math.Sin(angle)*DEGREES_PER_METER;
        // Return the GeoCoordinate
        return new GeoCoordinate(x, y);
    }
    // Return the instance of Uncertain<T>
    return new Uncertain<GeoCoordinate>(SamplingFunction);
}
```

Figure 12. The *Uncertain*$\langle T\rangle$ version of `GPS.GetLocation` returns an instance of *Uncertain*$\langle$*GeoCoordinate*$\rangle$.

sampling function to draw samples from the posterior distribution. The sampling function captures the values of `Point` and `Accuracy`. Although the sampling function is later invoked many times to draw samples from the distribution, the GPS hardware functions `GetHardwareLocation` and `GetHardwareAccuracy` are only invoked once for each call to `GetLocation`.

4.2 Computing with Distributions

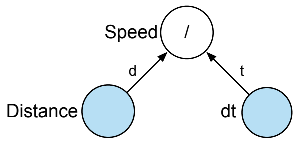
Uncertain$\langle T\rangle$ propagates error through computations by overloading operators from the base type T. These lifted operators dynamically construct Bayesian network representations of the computations they represent as the program executes. However, an alternate implementation could statically build a Bayesian network and only dynamically perform hypothesis tests at conditionals. We use the Bayesian network to define the sampling function for a computed variable in terms of the sampling functions of its operands. We only evaluate the sampling function at conditionals and so the root node of a network being sampled is always a comparison operator.

The computed sampling function uses the standard *ancestral sampling* technique for graphical models [5]. Because the Bayesian network is directed and acyclic, its nodes can be topologically ordered. We draw samples in this topological order, starting with each leaf node, for which sampling functions are explicitly defined. These samples then propagate up the Bayesian network. Each inner node is associated with an operator from the base type, and the node's children have already been sampled due to the topological order, so we simply apply the base operator to the operand samples to generate a sample from an inner node. This process continues up the network until reaching the root node of the network, generating a sample from the computed variable. The topological order guarantees that each node is visited exactly once in this process, and so the sampling process terminates.

In the GPS-Walking application in Figure 5(b), the user's speed is calculated by the line

```
    Uncertain<double> Speed = Distance / dt;
```

The lifted division operator constructs a Bayesian network for this computation:



Here shaded nodes indicate leaf distributions, for which sampling functions are defined explicitly.

To draw a sample from the variable `Speed`, we draw a sample from each leaf: a sample d from `Distance` (defined by the GPS library) and t from `dt` (a pointmass distribution, so all samples are equal). These samples propagate up the network to the `Speed` node. Because this node is a division operator, the resulting sample from `Speed` is simply d/t.

4.3 Asking Questions with Hypothesis Tests

Uncertain$\langle T\rangle$'s conditional and evaluation operators address the domain error that motivates our work. These operators require concrete decisions under uncertainty. The conditional operators must select one branch target to execute. In the GPS-Walking application in Figure 5(b), a conditional operator

```
    if (Speed > 4)...
```

must decide whether or not to enter this branch. Section 3.4 describes how *Uncertain*$\langle T\rangle$ executes this conditional by comparing the probability $\Pr[Speed > 4]$ to a default threshold 0.5, asking whether it is more likely than not that $Speed > 4$. To control approximation error, this comparison is performed by a hypothesis test, with null hypothesis $H_0 : \Pr[Speed > 4] \leq 0.5$ and alternate hypothesis $H_A : \Pr[Speed > 4] > 0.5$.

Sampling functions in combination with this hypothesis test control the efficiency-accuracy trade-off that approximation introduces. A higher confidence level for the hypothesis test leads to fewer approximation errors but requires more samples to evaluate. We perform the hypothesis test using Wald's *sequential probability ratio test* (SPRT) [32] to dynamically choose the right sample size for a particular conditional, only taking as many samples as necessary to obtain a statistically significant result.

We specify a step size, say $k = 10$, and start by drawing $n = k$ samples from the Bernoulli distribution $Speed > 4$. We then apply the SPRT to these samples to decide if the parameter p of the distribution (i.e., the probability $\Pr[Speed > 4]$) is significantly different from 0.5. If so, we can terminate immediately and take (or not take) the branch, depending on in which direction the significance lies. If the result is not significant, we draw another batch of k samples, and repeat the process with the now $n = 2k$ collection of samples. We repeat this process until either a significant result is achieved or a maximum sample size is reached to ensure termination. The SPRT ensures that this repeated sampling and testing process still achieves overall bounds on the probabilities of false positives (significance level) and false negatives (power).

Sampling functions and a Bayesian network representation may draw as many samples as we wish from any given variable. We may therefore exploit sequential methods, such as the SPRT, which do not fix the sample size for a hypothesis test in advance. Sequential methods are a principled solution to the efficiency-accuracy trade-off. They ensure we draw the minimum necessary number of samples for a sufficiently accurate result for each specific conditional. This goal-oriented sampling approach is a significant advance over previous random sampling approaches, which compute with a fixed pool of samples. These approaches do not efficiently control the effect of approximation error.

Wald's SPRT is optimal in terms of average sample size, but is potentially unbounded in any particular instance, so termination is not guaranteed. The artificial maximum sample size we introduce to guarantee termination has a small effect on the actual significance level and power of the test. We anticipate adapting the considerable body of work on group sequential methods [17], widely used in medical clinical trials, which provide "closed" sequential hypothesis tests with guaranteed upper bounds on the sample size.

We cannot apply the same sequential testing approach to the evaluation operator E, since there are no alternatives to compare against (i.e., no goal to achieve). Currently for this operator we simply draw a fixed number of samples and return their mean. We believe a more intelligent adaptive sampling process, sampling until the mean converges, may improve the performance of this approach.

5. Case Studies

We use three case studies to explore the expressiveness and correctness of $Uncertain\langle T\rangle$ on uncertain data. (1) We show how $Uncertain\langle T\rangle$ improves accuracy and expressiveness of speed computations from GPS, a widely used hardware sensor. (2) We show how $Uncertain\langle T\rangle$ exploits prior knowledge to minimize random noise in digital sensors. (3) We show how $Uncertain\langle T\rangle$ encourages developers to explicitly reason about and improve accuracy in machine learning, using a neural network that approximates hardware [12, 26].

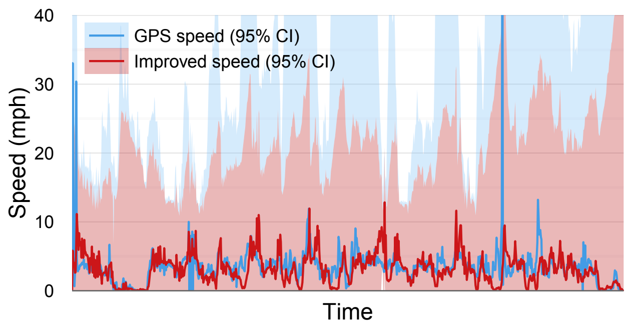


Figure 13. Data from the GPS-Walking application. Developers improve accuracy with priors and remove absurd values.

5.1 Uncertain<T> on Smartphones: GPS-Walking

The modern smartphone contains a plethora of hardware sensors. Thousands of applications use the GPS sensor, and many compute distance and speed from GPS readings. Our walking speed case study serves double duty, showing how uncertainty bugs occur in practice as well as a clear pedagogical demonstration of how $Uncertain\langle T\rangle$ improves correctness and expressiveness. We wrap the Windows Phone (WP) GPS location services API with $Uncertain\langle T\rangle$, exposing the error distribution. The GPS-Walking application estimates the user's speed by taking two samples from the GPS and computing the distance and time between them. Figure 5 compares the code for GPS-Walking with and without the uncertain type.

Defining the Distributions Our $Uncertain\langle T\rangle$ GPS library provides the function

```
Uncertain<GeoCoordinate> GPS.GetLocation();
```

which returns a distribution over the user's possible locations. Section 4.1 overviews how to derive the GPS distribution, and Figure 12 shows the implementation.

Computing with Distributions GPS-Walking uses locations from the GPS library to calculate the user's speed, since $Speed = \Delta Distance / \Delta Time$. Of course, since the locations are estimates, so too is speed. The developer must change the line

```
print("Speed: " + Speed);
```

from the original program in Figure 5(a), since `Speed` now has type $Uncertain\langle Double\rangle$. It now prints the expected value `Speed.E()` of the speed distribution.

We tested GPS-Walking by walking outside for 15 minutes. Figure 13 shows the expected values `Speed.E()` measured each second by the application (*GPS speed*). The uncertainty of the speed calculation, with extremely wide confidence intervals, explains the absurd values.

Conditionals GPS-Walking encourages users to walk faster than 4 mph with messages triggered by conditionals. The original implementation in Figure 5(a) uses naive conditionals, which are susceptible to random error. For example, on our test data, such conditionals would report the user to be walking faster than 7 mph (a running pace) for 30 seconds.

The *Uncertain*$\langle T\rangle$ version of GPS-Walking in Figure 5(b) evaluates evidence to execute conditionals. The conditional

```
if (Speed > 4)              GoodJob();
```

asks if it is *more likely than not* that the user is walking fast. The second conditional

```
else if ((Speed < 4).Pr(0.9)) SpeedUp();
```

asks if there is at least a 90% chance the user is walking slowly. This requirement is stricter than the first conditional because we do not want to unfairly admonish the user (i.e., we want to avoid false positives). Even the simpler *more likely than not* conditional improves the accuracy of GPS-Walking: such a conditional would only report the user as walking faster than 7 mph for 4 seconds.

Improving GPS Estimates with Priors Because GPS-Walking uses *Uncertain*$\langle T\rangle$, we may incorporate prior knowledge to improve the quality of its estimates. Assume for simplicity that users only invoke GPS-Walking when walking. Since humans are incredibly unlikely to walk at 60 mph or even 10 mph, we specify a prior distribution over likely walking speeds. Figure 13 shows the improved results (*Improved speed*). The confidence interval for the improved data is much tighter, and the prior knowledge removes the absurd results, such as walking at 59 mph.

Summary Developers need only make minimal changes to the original GPS-Walking application and are rewarded with improved correctness. They improve accuracy by reasoning correctly about uncertainty and eliminating absurd data with domain knowledge. This complex logic is difficult to implement without the uncertain type because a developer must know the error distribution for GPS, how to propagate it through calculations, and how to incorporate domain knowledge to improve the results. The uncertain type abstraction hides this complexity, improving programming productivity and application correctness.

5.2 Uncertain<T> with Sensor Error: SensorLife

This case study emulates a binary sensor with Gaussian noise to explore accuracy when ground truth is available. For many problems using uncertain data, ground truth is difficult and costly to obtain, but it is readily available in this problem formulation. This case study shows how *Uncertain*$\langle T\rangle$ makes it easier for non-expert developers to work with noisy sensors. Furthermore, it shows how expert developers can simply and succinctly use domain knowledge (i.e., the fact that the noise is Gaussian with known variance) to improve these estimates.

We use Conway's Game of Life, a cellular automaton that operates on a two-dimensional grid of cells that are each dead or alive. The game is broken up into generations. During each generation, the program updates each cell by (i) sensing the state of the cell's 8 neighbors, (ii) summing the binary value (dead or alive) of the 8 neighbors, and (iii) applying the following rules to the sum:

1. A live cell with 2 or 3 live neighbors lives.
2. A live cell with less than 2 live neighbors dies.
3. A live cell with more than 3 live neighbors dies.
4. A dead cell with exactly 3 live neighbors becomes live.

These rules simulate survival, underpopulation, overcrowding, and reproduction, respectively. Despite simple rules, the Game of Life provides complex and interesting dynamics (e.g., it is Turing complete [3]). We focus on the accuracy of sensing if the neighbors are dead or alive.

Defining the Distributions The original Game of Life's discrete perfect sensors define our ground truth. We view each cell as being equipped with up to eight sensors, one for each of its neighbors. Cells on corners and edges of the grid have fewer sensors. Each perfect sensor returns a binary value $s \in \{0,1\}$ indicating if the associated neighbor is alive.

We artificially induce zero-mean Gaussian noise $N(0,\sigma)$ on each of these sensors, where σ is the amplitude of the noise. Each sensor now returns a real number, not a binary value. We define three versions of this noisy Game of Life:

NaiveLife reads a single sample from each noisy sensor and sums the results directly to count the live neighbors.

SensorLife wraps each sensor with *Uncertain*$\langle T\rangle$. The sum uses the overloaded addition operator and each sensor may be sampled multiple times in a single generation.

BayesLife uses domain knowledge to improve SensorLife, as we describe below.

Our construction results in some negative sensor readings, but choosing a non-negative noise distribution, such as the Beta distribution, does not appreciably change our results.

Computing with Distributions Errors in each sensor are independent, so the function that counts a cell's live neighbors is almost unchanged:

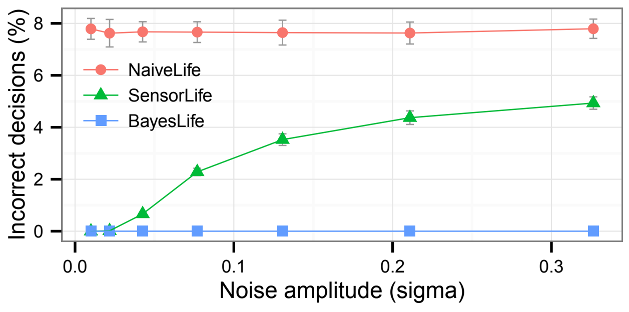
```
Uncertain<double> CountLiveNeighbors(Cell me) {
  Uncertain<double> sum = new Uncertain<double>(0.0);
  foreach (Cell neighbor in me.GetNeighbors())
    sum = sum + SenseNeighbor(me, neighbor);
  return sum;  }
```

Because each sensor now returns a real number rather than a binary value, the count of live neighbors is now a distribution over real numbers rather than an integer. Operator overloading means that no further changes are necessary, as the addition operator will automatically propagate uncertainty into the resulting sum.

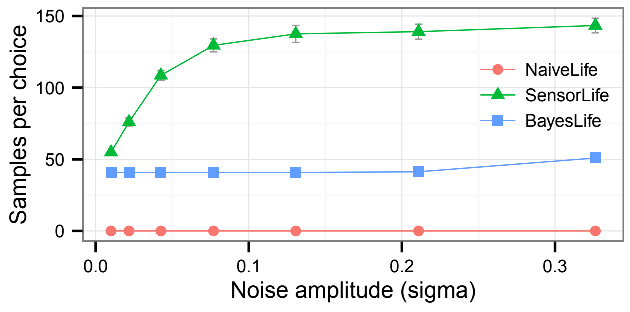
Conditionals The Game of Life applies its rules with four conditionals to the output of `CountLiveNeighbors`:

```
bool IsAlive = IsCellAlive(me);
bool WillBeAlive = IsAlive;
Uncertain<double> NumLive = CountLiveNeighbors(me);
if (IsAlive && NumLive < 2)
    WillBeAlive = false;
else if (IsAlive && 2 <= NumLive && NumLive <= 3)
    WillBeAlive = true;
else if (IsAlive && NumLive > 3)
    WillBeAlive = false;
else if (!IsAlive && NumLive == 3)
    WillBeAlive = true;
```

Each comparison involving `NumLive` implicitly performs a hypothesis test.



(a) Rate of incorrect decisions



(b) Samples required for each cell

Figure 14. SensorLife uses *Uncertain*$\langle T\rangle$ to significantly decrease the rate of incorrect decisions compared to a naive version, but requires more samples to make decisions. BayesLife incorporates domain knowledge into SensorLife for even better results.

Evaluation We compare the noisy versions of the Game of Life to the precise version. We perform this comparison across a range of noise amplitude values σ. Each execution randomly initializes a 20×20 cell board and performs 25 generations, evaluating a total of 10000 cell updates. For each noise level σ, we execute each Game of Life 50 times. We report means and 95% confidence intervals.

Figure 14(a) shows the rate of incorrect decisions (y-axis) made by each noisy Game of Life at various noise levels (x-axis). NaiveLife has a consistent error rate of 8%, as it takes only a small amount of noise to cross the integer thresholds in the rules of the Game of Life. SensorLife's errors scale with noise, as it considers multiple samples and so can correct smaller amounts of noise. At all noise levels, SensorLife is considerably more accurate than NaiveLife.

Mitigating noise has a cost, as *Uncertain*$\langle T\rangle$ must sample each sensor multiple times to evaluate each conditional. Figure 14(b) shows the number of samples drawn for each cell update (y-axis) by each noisy Game of Life at various noise levels (x-axis). Clearly NaiveLife only draws one sample per conditional. The number of samples for SensorLife increases as the noise level increases, because noisier sensors require more computation to reach a conclusion at the conditional.

Improving Estimates SensorLife achieves better results than NaiveLife despite not demonstrating any knowledge of the fact that the underlying true state of a sensor must be either 0 or 1. To improve SensorLife, an expert can exploit knowledge of the distribution and variance of the sensor noise. We call this improved version BayesLife.

Let v be the raw, noisy sensor reading, and s the underlying true state (either 0 or 1). Then $v = s + N(0,\sigma)$ for some σ we know. Since s is binary, we have two *hypotheses* for s: H_0 says $s = 0$ and H_1 says $s = 1$. The raw sensor reading v is evidence, and Bayes' theorem calculates a *posterior probability*

$$\Pr[H_0|v] = \Pr[v|H_0]\Pr[H_0]/\Pr[v]$$

for H_0 given the evidence, and similarly for H_1. To improve an estimate, we calculate which of H_0 and H_1 is more likely under the posterior probability, and fix the sensor reading to be 0 or 1 accordingly.

This formulation requires (i) the prior likelihoods $\Pr[H_0]$ and $\Pr[H_1]$; and (ii) a likelihood function to calculate $\Pr[v|H_0]$ and $\Pr[v|H_1]$. We assume no prior knowledge, so both H_0 and H_1 are equally likely: $\Pr[H_0] = \Pr[H_1] = 0.5$. Because we know the noise is Gaussian, we know the likelihood function is just the likelihood that $N(0,\sigma) = s - v$ for each of $s = 0$ and 1, which we calculate trivially using the Gaussian density function. We need not calculate $\Pr[v]$ since it is a common denominator of the two probabilities we compare.

To implement BayesLife we wrap each sensor with a new function `SenseNeighborFixed`. Since the two likelihoods $\Pr[v|H_0]$ and $\Pr[v|H_1]$ have the same variance and shape and are symmetric around 0 and 1, respectively, and the priors are equal, the hypothesis with higher posterior probability is simply the closer of 0 or 1 to v. The implementation is therefore trivial:

```
Uncertain<double>
SenseNeighborFixed(Cell me, Cell neighbor) {
  Func<double> samplingFunction = () => {
    Uncertain<double> Raw = SenseNeighbor(me, neighbor);
    double Sample = Raw.Sample();
    return Sample > 0.5 ? 1.0 : 0.0;
  };
  return new Uncertain<double>(samplingFunction);  }
```

Evaluation of BayesLife Figure 14(a) shows that BayesLife makes no mistakes at all at these noise levels. At noise levels higher than $\sigma = 0.4$, considering individual samples in isolation breaks down as the values become almost completely random. A better implementation could calculate joint likelihoods with multiple samples, since each sample is drawn from the same underlying distribution. Figure 14(b) shows that BayesLife requires fewer samples than SensorLife, but of course still requires more samples than NaiveLife.

5.3 Uncertain<T> for Machine Learning: Parakeet

This section explores using *Uncertain*$\langle T\rangle$ for machine learning, inspired by *Parrot* [12], which trains neural networks for approximate hardware. We study the Sobel operator from the Parrot evaluation, which calculates the gradient of image intensity at a pixel. Parrot creates a neural network that approximates the Sobel operator.

Machine learning algorithms estimate the true value of a function. One source of uncertainty in their estimates is generalization error, where predictions are good on training data but poor on unseen data. To combat this error, Bayesian machine learning considers *distributions* of estimates, reflecting how different estimators would answer an unseen input. Figure 15 shows for one such input a distribution of neural network outputs created by the Monte Carlo method described below, the output from the single naive neural network Parrot trains, and the true output. The one Parrot value is significantly different from the correct value. Using a distribution helps mitigate generalization error by recognizing other possible predictions.

Computation amplifies generalization error. For example, edge-detection algorithms use the Sobel operator to report an edge if the gradient is large (e.g., $s(p) > 0.1$). Though Parrot approximates the Sobel operator well, with an average root-mean-square error of 3.4%, using its output in such a conditional is a computation. This computation amplifies the error and results in a 36% false positive rate. In Figure 15, by considering the entire distribution, the evidence for the condition $s(p) > 0.1$ is only 70%. To accurately consume estimates, developers must consider the effect of uncertainty not just on direct output but on computations that consume it.

We introduce Parakeet, which approximates code using Bayesian neural networks, and encapsulates the distribution with $Uncertain\langle T\rangle$. This abstraction encourages developers to consider uncertainty in the machine learning prediction. We evaluate Parakeet by approximating the Sobel operator $s(p)$ and computing the edge detection conditional $s(p) > 0.1$.

Identifying the Distribution We seek a *posterior predictive distribution* (PPD), which tells us for a given input how likely each possible output is to be correct, based on the training data. Intuitively, this distribution reflects predictions from *other* neural networks that also explain the training data well.

Formally, a neural network is a function $y(\mathbf{x};\mathbf{w})$ that approximates the output of a target function $f(\mathbf{x})$. The weight vector $\mathbf{w}$ describes how each neuron in the network relates to the others. Traditional training uses example inputs and outputs to learn a single weight vector $\mathbf{w}$ for prediction. In a Bayesian formulation, we learn the PPD $p(t|\mathbf{x},\mathcal{D})$ (the distribution in Figure 15), a distribution of predictions of $f(\mathbf{x})$ given the training data $\mathcal{D}$.

We adopt the *hybrid Monte Carlo* algorithm to create samples from the PPD [20]. Each sample describes a neural network. Intuitively, we create multiple neural networks by perturbing the search. Formally, we sample the posterior distribution $p(\mathbf{w}|\mathcal{D})$ to approximate the PPD $p(t|\mathbf{x},\mathcal{D})$ using Monte Carlo integration, since

$$p(t|\mathbf{x},\mathcal{D}) = \int p(t|\mathbf{x},\mathbf{w})p(\mathbf{w}|\mathcal{D})\,\mathrm{d}\mathbf{w}.$$

The instance of $Uncertain\langle T\rangle$ that Parakeet returns draws samples from this PPD. Each sample of $p(\mathbf{w}|\mathcal{D})$ from hybrid Monte Carlo is a vector $\mathbf{w}$ of weights for a neural network.

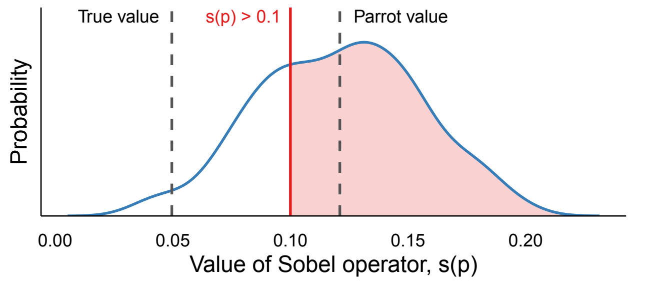


Figure 15. Error distribution for Sobel approximated by neural networks. Parrot experiences a false positive on this test, which Parakeet eliminates by evaluating the *evidence* that $s(p) > 0.1$ (the shaded area).

To evaluate a sample from the PPD $p(t|\mathbf{x},\mathcal{D})$, we execute a neural network using the weights $\mathbf{w}$ and input $\mathbf{x}$. The resulting output is a sample of the PPD.

We execute hybrid Monte Carlo offline and capture a fixed number of samples in a training phase. We use these samples at runtime as a fixed pool for the sampling function. If the sample size is sufficiently large, this approach approximates true sampling well. As with other Markov chain Monte Carlo algorithms, the next sample in hybrid Monte Carlo depends on the current sample, which improves on pure random walk behavior that scales poorly in high dimensions. To compensate for this dependence we discard most samples and only retain every M^{th} sample for some large M.

Hybrid Monte Carlo has two downsides. First, we must execute multiple neural networks (one for each sample of the PPD). Second, it often requires hand tuning to achieve practical rejection rates. Other PPD approximations strike different trade-offs. For example, a Gaussian approximation [5] to the PPD would mitigate all these downsides, but may be an inappropriate approximation in some cases. Since the Sobel operator's posterior is approximately Gaussian (Figure 15), a Gaussian approximation may be appropriate.

Evaluation We approximate the Sobel operator with Parakeet, using 5000 examples for training and a separate 500 examples for evaluation. For each evaluation example we compute the ground truth $s(p) > 0.1$ and then evaluate this conditional using $Uncertain\langle T\rangle$, which asks whether $\Pr[s(p) > 0.1] > \alpha$ for varying thresholds α.

Figure 16 shows the results of our evaluation. The x-axis plots a range of conditional thresholds α and the y-axis plots precision and recall for the evaluation data. Precision is the probability that a detected edge is actually an edge, and so describes false positives. Recall is the probability that an actual edge is detected, and so describes false negatives. Because Parrot does not consider uncertainty, it locks developers into a particular balance of precision and recall. In this example, Parrot provides 100% recall, detecting all actual edges, but only 64% precision, so 36% of reported edges are false positives. With a conditional threshold, developers select their own balance. For example, a threshold of $\alpha = 0.8$ (i.e., eval-

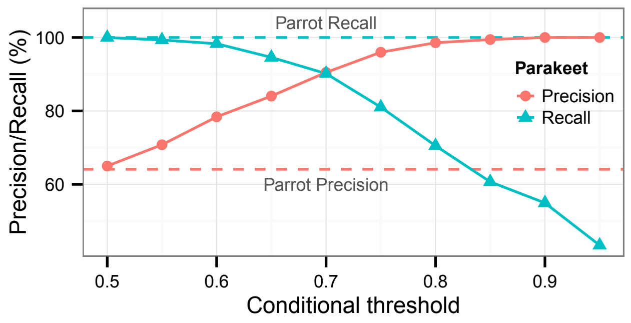


Figure 16. Developers choose a balance between false positives and negatives with Parakeet using *Uncertain*$\langle T \rangle$.

uating $\Pr[s(p) > 0.1] > 0.8$) results in 71% recall and 99% precision, meaning more false negatives (missed edges) but fewer false positives. The ideal balance of false positives and negatives depends on the particular application and is now in the hands of the developer.

Summary Machine learning approximates functions and so introduces uncertainty. Failing to consider uncertainty causes uncertainty bugs; even though Parrot approximates the Sobel operator with 3.4% average error, using this approximation in a conditional results in a 36% false positive rate. Parakeet uses *Uncertain*$\langle T \rangle$, which encourages the developer to consider the effect of uncertainty on machine learning predictions. The results on Parakeet show that developers may easily control the balance between false positives and false negatives, as appropriate for the application, using *Uncertain*$\langle T \rangle$. Of course, *Uncertain*$\langle T \rangle$ is not a panacea for machine learning. Our formulation leverages a probabilistic interpretation of neural networks, which not all machine learning techniques have, and only addresses generalization error.

These case studies highlight the need for a programming language solution for uncertainty. Many systems view error in a local scope, measuring its effect only on direct output, but as all the case studies clearly illustrate, the effect of uncertainty on programs is inevitably global. We cannot view estimation processes such as GPS and machine learning in isolation. *Uncertain*$\langle T \rangle$ encourages developers to reason about how local errors impact the entirety of their programs.

6. Related Work

This section relates our work to programming languages with statistical semantics or that support solving statistical problems. Because we use, rather than add to, the statistics literature, prior sections cite this literature as appropriate.

Probabilistic Programming Prior work has focused on the needs of experts. Probabilistic programming creates generative models of problems involving uncertainty by introducing random variables into the syntax and semantics of a programming language. Experts encode generative models which the program queries through inference techniques. The foundation of probabilistic programming is the monadic structure of probability distributions [14, 25], an elegant representation of joint probability distributions (i.e., generative models) in a functional language. Researchers have evolved this idea into a variety of languages such as BUGS [13], Church [15], Fun [6], and IBAL [24].

```
earthquake = Bernoulli(0.0001)
burglary = Bernoulli(0.001)
alarm = earthquake or burglary
if (earthquake)
  phoneWorking = Bernoulli(0.7)
else
  phoneWorking = Bernoulli(0.99)

observe(alarm)        # If the alarm goes off...
query(phoneWorking)   # ...does the phone still work?
```

Figure 17. A probabilistic program. To infer the probability of `phoneWorking` requires exploring both branches [10].

Figure 17 shows an example probabilistic program that infers the likelihood that a phone is working given that an alarm goes off. The program queries the posterior distribution of the Bernoulli variable `phoneWorking` given the observation that `alarm` is true. To infer this distribution by sampling, the runtime must repeatedly evaluate both branches. This program illustrates a key shortcoming of many probabilistic programming languages: inference is very expensive for poor rejection rates. Since there is only a 0.11% chance of `alarm` being true, most inference techniques will have high rejection rates and so require many samples to sufficiently infer the posterior distribution. Using Church [15], we measured 20 seconds to draw 100 samples from this model [7].

Uncertain$\langle T \rangle$ is a probabilistic programming language, since it manipulates random variables in a host language. Unlike other probabilistic languages, *Uncertain*$\langle T \rangle$ only executes one side of a conditional branch, and only reasons about conditional distributions. For example, air temperature depends on variables such as humidity, altitude, and pressure. When programming with data from a temperature sensor, the question is not whether temperature can ever be greater than 85° (which a joint distribution can answer), but rather whether the current measurement from the sensor is greater than 85°. This measurement is inextricably conditioned on the current humidity, altitude, and pressure, and so the conditional distribution that *Uncertain*$\langle T \rangle$ manipulates is appropriate. For programs consuming estimated data, problems are not abstract but concrete instances and so the conditional distribution is just as useful as the full joint distribution. *Uncertain*$\langle T \rangle$ exploits this restriction to achieve efficiency and accessibility, since these conditional distributions are specified by operator overloading and evaluated with ancestral sampling.

Like the probability monad [25], *Uncertain*$\langle T \rangle$ builds and later queries a computation tree, but it adds continuous distributions and a semantics for conditional expressions that developers must implement manually using the monad. *Uncertain*$\langle T \rangle$ uses sampling functions in the same fashion as Park et al. [23], but we add accessible and principled conditional operators. Park et al. do not describe a mechanism for choosing how many samples to draw from a sampling function; instead their query operations use a fixed runtime-specified sample size. We exploit hypothesis testing in conditional expressions to dynamically select the appropriate sample size. Sankaranarayanan et al. [27] provide a static analysis for checking assertions in probabilistic programs. Their `estimateProbability` assertion calculates the probability of a conditional expression involving random variables being true. While their approach is superficially similar to *Uncertain*$\langle T\rangle$'s conditional expressions, `estimateProbability` handles only simple random variables and linear arithmetic operations so that it can operate without sampling. *Uncertain*$\langle T\rangle$ addresses arbitrary random variables and operations, and uses hypothesis testing to limit the performance impact of sampling in a principled way.

Domain-Specific Approaches In robotics, CES [30] extends C++ to include probabilistic variables (e.g., `prob<int>`) for simple distributions. Instances of `prob<T>` store a list of pairs $(x, p(x))$ that map each possible value of the variable to its probability. This representation restricts CES to simple discrete distributions. *Uncertain*$\langle T\rangle$ adopts CES's idea of encapsulating random variables in a generic type, but uses a more robust representation for distributions and adds an accessible semantics for conditional expressions.

In databases, Barbara et al. incorporate uncertainty and estimation into the semantics of relational operators [1]. Benjelloun et al. trace provenance back through probabilistic queries [2], and Dalvi and Suciu describe a "possible worlds" semantics for relational joins [11]. Hazy [18] asks developers to write Markov logic networks (sets of logic rules and confidences) to interact with databases while managing uncertainty. These probabilistic databases must build a model of the entire joint posterior distribution of a query. In contrast, *Uncertain*$\langle T\rangle$ reasons only about conditional distributions.

Interval analysis represents an uncertain value as a simple interval and propagates it through computations [19]. For example, if $X = [4,6]$, then $X/2 = [2,3]$. Interval analysis is particularly suited to bounding floating point error in scientific computation. The advantage of interval analysis is its simplicity and efficiency, since many operations over real numbers are trivial to define over intervals. The downside is that intervals treat all random variables as having uniform distributions, an assumption far too limiting for many applications. By using sampling functions, *Uncertain*$\langle T\rangle$ achieves the simplicity of interval analysis when defining operations, but is more broadly applicable.

In approximate computing, EnerJ [26] uses the type system to separate exact from approximate data and govern how they interact. This type system encourages developers to reason about approximate data in their program. *Uncertain*$\langle T\rangle$ builds on this idea with a semantics for the quantitative impact of uncertainty on data. Rely is a programming language that statically reasons about the quantitative error of a program executing on unreliable hardware [9]. Developers use Rely to express assertions about the accuracy of their code's output, and the Rely analyzer statically verifies whether these assertions could be breached based on a specification of unreliable hardware failures. In contrast to *Uncertain*$\langle T\rangle$, Rely does not address applications that compute with random variables and does not reason dynamically about the error in a particular instance of uncertain data.

7. Conclusion

Emerging applications solve increasingly ambitious and ambiguous problems on data from tiny smartphone sensors, huge distributed databases, the web, and simulations. Although practitioners in other sciences have principled ways to make decisions under uncertainty, only recently have programming languages begun to assist developers with this task. Because prior solutions are either too simple to aid correctness or too complex for most developers to use, many developers create bugs by ignoring uncertainty completely.

This paper presents a new abstraction and shows how it helps developers to correctly operate on and reason with uncertain data. We describe its syntax and probabilistic semantics, emphasizing simplicity for non-experts while encouraging developers to consider the effects of random error. Our semantics for conditional expressions helps developers to understand and control false positives and false negatives. We present implementation strategies that use sampling and hypothesis testing to realize our goals efficiently. Compared to probabilistic programming, *Uncertain*$\langle T\rangle$ gains substantial efficiency through goal-specific sampling. Our three case studies show that *Uncertain*$\langle T\rangle$ improves application correctness in practice without burdening developers.

Future directions for *Uncertain*$\langle T\rangle$ include runtime and compiler optimizations that exploit statistical knowledge; exploring accuracy, efficiency, and expressiveness for more substantial applications; and improving correctness with models of common phenomena, such as physics, calendar, and history in uncertain data libraries.

Uncertainty is not a vice to abstract away, but many applications and libraries try to avoid its complexity. *Uncertain*$\langle T\rangle$ embraces uncertainty while recognizing that most developers are not statistics experts. Our research experience with *Uncertain*$\langle T\rangle$ suggests that it has the potential to change how developers think about and solve the growing variety of problems that involve uncertainty.

Acknowledgments

We thank Steve Blackburn, Adrian Sampson, Darko Stefanovic and Ben Zorn for comments on earlier versions of this work and the anonymous reviewers for their helpful feedback.

References

[1] D. Barbara, H. Garcia-Molina, and D. Porter. The management of probabilistic data. *IEEE Transactions on Knowledge and Data Engineering*, 4(5):487–502, 1992.

[2] O. Benjelloun, A. D. Sarma, A. Halevy, and J. Widom. ULDBs: Databases with uncertainty and lineage. In *ACM International Conference on Very Large Data Bases (VLDB)*, pages 953–964, 2006.

[3] E. R. Berlekamp, J. H. Conway, and R. K. Guy. *Winning Ways for Your Mathematical Plays*, volume 4. A K Peters, 2004.

[4] S. Bhat, A. Agarwal, R. Vuduc, and A. Gray. A type theory for probability density functions. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 545–556, 2012.

[5] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[6] J. Borgström, A. D. Gordon, M. Greenberg, J. Margetson, and J. Van Gael. Measure transformer semantics for Bayesian machine learning. In *European Symposium on Programming (ESOP)*, pages 77–96, 2011.

[7] J. Bornholt. *Abstractions and techniques for programming with uncertain data*. Honors thesis, Australian National University, 2013.

[8] G. E. P. Box and M. E. Muller. A note on the generation of random normal deviates. *The Annals of Mathematical Statistics*, 29(2):610–611, 1958.

[9] M. Carbin, S. Misailovic, and M. C. Rinard. Verifying quantitative reliability of programs that execute on unreliable hardware. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 33–52, 2013.

[10] A. T. Chaganty, A. V. Nori, and S. K. Rajamani. Efficiently sampling probabilistic programs via program analysis. In *Proceedings of the 16th international conference on Artificial Intelligence and Statistics, AISTATS 2013, Scottsdale, AZ, USA, April 29 - May 1, 2013*, pages 153–160. JMLR, 2013.

[11] N. Dalvi and D. Suciu. Management of probabilistic data: Foundations and challenges. In *ACM Symposium on Principles of Database Systems (PODS)*, pages 1–12, 2007.

[12] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 449–460, 2012.

[13] W. R. Gilks, A. Thomas, and D. J. Spiegelhalter. A language and program for complex Bayesian modelling. *Journal of the Royal Statistical Society. Series D (The Statistician)*, 43(1): 169–177, 1994.

[14] M. Giry. A categorical approach to probability theory. In *Categorical Aspects of Topology and Analysis*, volume 915 of *Lecture Notes in Mathematics*, pages 68–85. 1982.

[15] N. D. Goodman, V. K. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum. Church: A language for generative models. In *Conference in Uncertainty in Artificial Intelligence (UAI)*, pages 220–229, 2008.

[16] S. Jaroszewicz and M. Korzeń. Arithmetic operations on independent random variables: A numerical approach. *SIAM Journal on Scientific Computing*, 34:A1241–A1265, 2012.

[17] C. Jennison and B. W. Turnbull. *Group sequential methods with applications to clinical trials*. Chapman & Hall, 2000.

[18] A. Kumar, F. Niu, and C. Ré. Hazy: Making it Easier to Build and Maintain Big-data Analytics. *ACM Queue*, 11(1):30–46, 2013.

[19] R. E. Moore. *Interval analysis*. Prentice-Hall, 1966.

[20] R. M. Neal. *Bayesian learning for neural networks*. PhD thesis, University of Toronto, 1994.

[21] P. Newson and J. Krumm. Hidden Markov map matching through noise and sparseness. In *ACM International Conference on Advances in Geographic Information Systems (GIS)*, pages 336–343, 2009.

[22] A. Papoulis and S. U. Pillai. *Probability, random variables, and stochastic processes*. New York, NY, 4th edition, 2000.

[23] S. Park, F. Pfenning, and S. Thrun. A probabilistic language based on sampling functions. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 171–182, 2005.

[24] A. Pfeffer. IBAL: a probabilistic rational programming language. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 733–740, 2001.

[25] N. Ramsey and A. Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 154–165, 2002.

[26] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. EnerJ: Approximate data types for safe and general low-power computation. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 164–174, 2011.

[27] S. Sankaranarayanan, A. Chakarov, and S. Gulwani. Static analysis for probabilistic programs: inferring whole program properties from finitely many paths. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 447–458, 2013.

[28] J. Schwarz, J. Mankoff, and S. E. Hudson. Monte Carlo methods for managing interactive state, action and feedback under uncertainty. In *ACM Symposium on User Interface Software and Technology (UIST)*, pages 235–144, 2011.

[29] R. Thompson. Global positioning system: the mathematics of GPS receivers. *Mathematics Magazine*, 71(4):260–269, 1998.

[30] S. Thrun. Towards programming tools for robots that integrate probabilistic computation and learning. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 306–312, 2000.

[31] F. Topsøe. On the Glivenko-Cantelli theorem. *Zeitschrift fr Wahrscheinlichkeitstheorie und Verwandte Gebiete*, 14:239–250, 1970.

[32] A. Wald. Sequential Tests of Statistical Hypotheses. *The Annals of Mathematical Statistics*, 16(2):117–186, 1945.