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

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

Abstract

Applications increasingly seek to solve problems involving *uncertain* data, from sensors (such as GPS), probabilistic models, machine learning, approximate computing, and myriad other sources. Programmers reason about such uncertain data using programming languages. Unfortunately, existing programming languages represent uncertain data with simple discrete types (floats, integers, booleans, etc.), encouraging the programmer to pretend the data is not probabilistic. This illusion of certainty causes random, transient, and hard-to-reproduce bugs in applications.

We introduce the *uncertain type*, *Uncertain*$\langle T \rangle$, a programming language abstraction for uncertain data. In contrast to prior work, *Uncertain*$\langle T \rangle$ balances expressiveness with accessibility to non-expert programmers, who are on the front lines of the increasing variety of problems involving uncertainty. We implement a novel Bayesian network semantics for computation on uncertain data, and intuitive conditional operators for answering questions under uncertainty. The *Uncertain*$\langle T \rangle$ runtime uses goal-oriented random sampling and hypothesis testing to evaluate computations and conditionals lazily and efficiently. We show through three case studies how these contributions create an accessible abstraction that improves program correctness. *Uncertain*$\langle T \rangle$ provides a compelling programming model for a computing landscape which increasingly faces the challenge of uncertainty.

This work appeared at ASPLOS 2014 [2].

1. Problem and Motivation

Modern applications increasingly face the challenge of computing under uncertainty from many sources. Limited sensor resolution creates uncertainty about true values. Approximate computations and algorithms save energy or improve performance, but introduce uncertainty into their results. Machine learning produces inherently uncertain predictions, learned from a finite set of training data. Or the uncertainty might be an inherent part of a probabilistic model, to reason about problems that are random and yet structured.

Regardless of the source of the uncertainty, programmers reason about it using programming languages. But most programming languages do not distinguish between uncertain and certain values, simplistically representing both with discrete types such as floats, integers, and booleans. These simple types belie the complexity of uncertainty, and encourage programmers to treat uncertain data as exact. While motivated programmers can reason about uncertainty with these types, most who face this challenge simply ignore it. For example, our survey of 100 popular smartphone applications that use GPS data (which is uncertain) found only one application reasons about the uncertainty in that data.

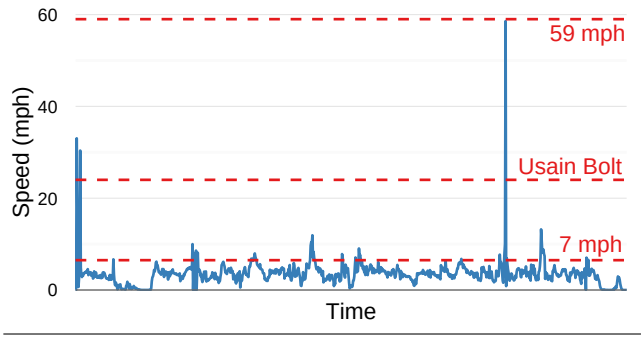


Figure 1. A naive GPS application produces absurd walking speeds.

Figure 1 shows speed data collected from a GPS fitness application while walking. While Usain Bolt runs 100 m at 24 mph, and the average human walks at 3 mph, this data shows a walking speed of 59 mph, and repeated walking speeds above 7 mph (a running speed). Programming languages foster these errors because simplistic types do not reflect the *compounding effect of computation* on uncertainty.

Uncertainty also pervades conditional operators, particularly nefarious given that the goal of most computer programs is to make decisions with data. Esmaeilzadeh et al. train a neural network to approximate the Sobel operator $s(p)$, commonly used in image processing [6]. The resulting predictor achieves 3% average error, indicating a good approximation. But using this same network to evaluate a conditional $s(p) > 0.1$ has a 36% false positive rate. The cause is the failure to consider the output's uncertainty, encouraged by the allure of a simple type system or API (in this case, a neural network library that returns a single predicted value).

2. Background and Related Work

Many disciplines understand the challenges of computation and decision making under uncertainty. Existing approaches try to help programmers with these challenges through improved programming language abstractions. The most successful of these approaches are *probabilistic programming languages*, in which program variables are random variables and program statements describe dependences between them.

```
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 2. A simple probabilistic program [5].

Early work recognised that probability distributions form a monad [8, 14]. Popular probabilistic programming languages include Infer.NET/Fun [1], BUGS [7], and Church [9].

Figure 2 shows an example of a probabilistic program that infers the likelihood that a phone is working given that an alarm goes off [5]. The model captures the dependences between variables: the phone is less likely to work after an earthquake, but either an earthquake or a burglary can trigger the alarm, and a burglary is much more likely. The query first observes that an alarm goes off and then infers the *posterior* probability that the phone works. Of course, probabilistic programs can express much more complex models, including those with continuous variables.

The machine learning community has eagerly adopted probabilistic programming languages because they provide an elegant abstraction of probabilistic graphical models, and so lend themselves to a wide range of common problems. But this expressiveness and flexibility also limits their relevance and adoption outside machine learning, because they require programmers to phrase their problems in terms of probabilistic models. As an increasingly variety of problem domains require reasoning about uncertainty, it is unlikely that the programmers facing these challenges will have the requisite statistics or machine learning expertise for such tools.

Various domain-specific approaches to uncertainty also exist. In robotics, Thrun demonstrates a C++ library for probabilistic variables (e.g., `prob<int>`) with simple distributions [15]. In databases, Kumar et al. propose that programmers write Markov logic networks to query probabilistic databases [10]. In scientific computing, interval analysis is a well-known technique for propagating uncertainty through computations [11]. In approximate computing, Carbin et al. statically reason about the effect of unreliable hardware on the accuracy of program computations [4]. Unfortunately, as with probabilistic programming, these domain-specific solutions often require significant expertise and are unlikely to see wide adoption in new applications.

One of the key tenets of programming language research is *abstraction*: the idea that we hide unnecessary complexity from programmers while empowering them to solve more ambitious problems. The challenge lies in balancing these two competing goals, to expose sufficient detail to create a useful class of applications while hiding enough complexity to ease the burden on the programmer. Existing solutions to the challenge of uncertain data either overwhelm non-experts with too much detail (probabilistic programming languages) or ignore uncertainty completely in pursuit of a simple abstraction (existing APIs for sensor data, machine learning libraries, etc.). In contrast, our work recognises that while uncertainty is not a vice to abstract away, most programmers are not statistics experts and so will not make onerous statistical demands of the abstraction. We will show that a simple, accessible, efficient abstraction is still sufficiently expressive to solve many common and emerging problems.

3. Approach and Uniqueness

We introduce *Uncertain*$\langle T \rangle$, a generic data type with operations that capture and manipulate uncertain data as probability distributions. The operations propagate uncertainty through computations by operator overloading, and provide an intuitive semantics for conditional expressions under uncertainty. We implement *Uncertain*$\langle T \rangle$ in C#, as well as C++ and Python prototypes, to demonstrate the abstraction's generality. Figure 3 shows a simple GPS fitness application, implemented (a) without and (b) with the uncertain type.

Our unique approach focuses on creating an accessible interface for non-expert programmers to solve increasingly uncertain problems. We make four key contributions:

- We show that representing uncertainty using *sampling functions* allows *Uncertain*$\langle T \rangle$ to represent many interesting application domains effectively.
- We provide a *Bayesian network semantics for computations*, in which arithmetic operators dynamically construct graph representations of their computations.
- We define a new *semantics for conditional expressions* involving the uncertain type, which is intuitive but expressive in balancing false positives and false negatives.
- Our implementation exploits these contributions through *goal-oriented sampling*, which uses hypothesis tests and insights from medical clinical trials to dynamically balance performance and accuracy.

3.1 Representing Uncertainty

An object of type *Uncertain*$\langle T \rangle$ is a random variable of a numeric type T. *Uncertain*$\langle T \rangle$ captures uncertainty as a probability distribution, which it stores as a *sampling function*. A sampling function is a function of no arguments that returns a new random sample from its distribution on each invocation [13]. Other approaches that store closed-form representations of distributions are not sufficiently expressive for the growing class of problems involving uncertainty. For example, many important distributions for sensors, road maps, approximate hardware, and machine learning do not have closed forms. Closed form representation also results in symbolic explosion; for example, even the sum of two distributions requires evaluating a difficult integral.

Sampling functions obviate these shortcomings, but are necessarily approximate. Sampling functions can be arbitrarily accurate given sufficient time and space [16], and so sampling creates an efficiency-accuracy trade-off. Our other contributions demonstrate how *Uncertain*$\langle T \rangle$ optimises this trade-off dynamically for each particular computation.

```
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 *Uncertain⟨T⟩*

```
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 *Uncertain⟨T⟩*

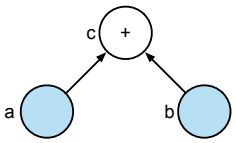
Figure 3. A simple fitness application implemented with and without *Uncertain⟨T⟩*.

3.2 Computations Under Uncertainty

Rather than eagerly perform computation, *Uncertain⟨T⟩*'s overloaded arithmetic operators dynamically construct a *Bayesian network* graph representation of computations. Nodes in these directed acyclic graphs are random variables, and edges represent conditional dependences. Leaf nodes represent known distributions defined in libraries by expert programmers, while inner nodes represent computations on random variables. For example, the program:

```
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) associated with the addition operation.

Each node in the Bayesian network is an instance of *Uncertain⟨T⟩* and so has a sampling function. Expert programmers define the leaf nodes by providing sampling functions; for example, the Gaussian type uses the Box-Muller transform to draw its random samples [3]. For an inner node, *Uncertain⟨T⟩* uses the well-known *ancestral sampling* technique, which draws a sample from each of the node's parents, and applies the node's associated operator to those samples. This implementation is far more tractable than if *Uncertain⟨T⟩* were to manipulate algebraic representations, which tend to grow explosively even in simple programs.

3.3 Conditional Expressions

Most programs use data to make decisions, which programming languages express with conditional expressions. Conditional expressions are straightforward when a program stores data in simple types: if *Speed* is of type `double` then evaluating $Speed > 4$ is trivial. But what if *Speed* is uncertain?

Uncertain⟨T⟩ defines the semantics of conditional expressions involving uncertain data by computing *evidence* for a conclusion. Figure 4 shows that the probability distribution of the variable *Speed* induces a *probability* that $Speed > 4$ equal to the shaded area under the curve. To evaluate the conditional, *Uncertain⟨T⟩* asks whether this probability is at least 50%, so whether the condition is more likely than not

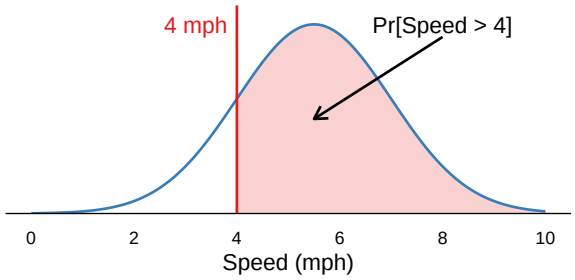


Figure 4. With uncertain data there is a *probability* that a condition is true.

to be true. This semantics gives a reasonable and accessible definition to the conditional `Speed > 4`, easing the demand on programmers in adopting a new abstraction.

Using an explicit conditional operator, programmers can also specify a probability threshold to compare against. For example, a programmer may ask if the probability is at least 90% by writing `(Speed > 4).Pr(0.9)`. The probability threshold helps the programmer to control the balance between false positives and false negatives. Higher thresholds 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). Unlike existing abstractions, which hide this balance from the programmer and lock them into a single configuration, *Uncertain⟨T⟩* allows programmers to express how sensitive their program is to each class of decision error.

3.4 Goal-Oriented Sampling

Uncertain⟨T⟩ defers most computation by constructing Bayesian networks. But when encountering a conditional branch such as `if (Speed > 4)`, the execution must make a concrete decision about which branch to enter, and so cannot continue deferring work. *Uncertain⟨T⟩* makes this concrete decision by establishing a hypothesis test to decide the result of the conditional. The null hypothesis is $\Pr[Speed > 4] \leq 0.5$ and the alternate hypothesis is $\Pr[Speed > 4] > 0.5$. If we can reject the null hypothesis, we have shown (at a certain confidence level) that $Speed > 4$ is more likely than not to be true. If a programmer provides an explicit probability threshold, the hypotheses use that value instead.

To gather samples for use in this hypothesis test, the runtime calls the sampling function for the Bernoulli distribution $Speed > 4$. Just as with arithmetic operators, *Uncertain⟨T⟩*

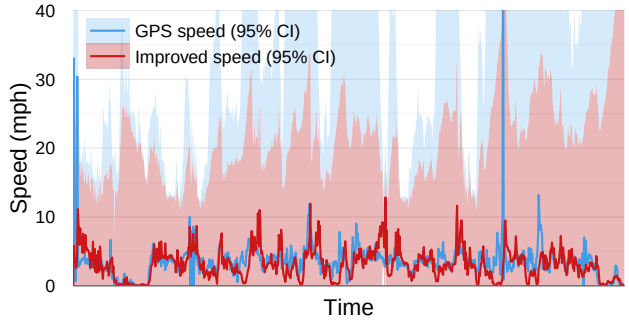


Figure 5. *Uncertain⟨T⟩* makes GPS fitness data more useful and accurate.

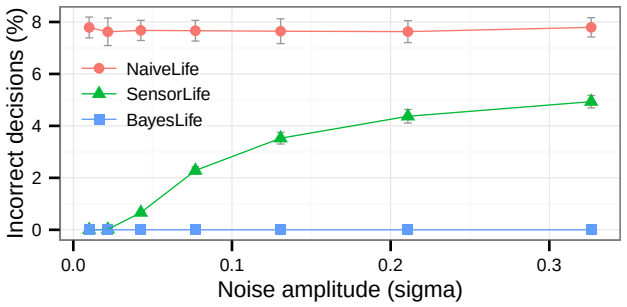


Figure 6. Using *Uncertain⟨T⟩* (SensorLife and BayesLife) improves accuracy of the noisy Game of Life.

overloads the comparison operator $>$ to instead construct a Bayesian network node, with operands *Speed* and 4 and associated operator $>$. To sample this node the runtime samples each operand and applies the operator to the samples.

The runtime uses *goal-oriented sampling* to decide how many samples to draw for the hypothesis test. This technique adapts a sequential sampling approach, Wald's sequential probability ratio test (SPRT) [17], which is widely used in medical clinical trials to optimise the number of trial participants to ensure both statistical significance and minimal cost. The intuition is that the test can examine the results of the hypothesis test incrementally, and terminate early once a result is obvious. The SPRT governs these inspections to guarantee the desired statistical significance. Most importantly, because the SPRT is dynamic, it adapts to the individual conditional the runtime is evaluating, and so can exploit the structure of the particular problem. In contrast, existing work must fix a sample size in advance, and so risks being either too slow or too inaccurate for any given problem.

4. Results and Contributions

We use three case studies to demonstrate the utility and efficiency of *Uncertain⟨T⟩*. (1) We show how *Uncertain⟨T⟩* improves accuracy of speed computations from GPS data. (2) We show how *Uncertain⟨T⟩* can minimise the effect of random noise in digital sensors. (3) We demonstrate how *Uncertain⟨T⟩* encourages programmers to reason about false positives and negatives in machine learning. More detailed analyses of these case studies appear in our recent paper [2].

4.1 Uncertain<T> for GPS Data

Many popular smartphone applications use data from GPS sensors. A common use of this data is computing the user's current speed, or their distance travelled. Because GPS data is uncertain, so too are these derivative calculations. Figure 1 showed that ignoring this uncertainty leads to absurd results.

Figure 3(b) shows a simple GPS fitness application written with *Uncertain⟨T⟩*. The application uses GPS observations to compute the user's walking speed, and either congratulates them on a good workout or encourages them to speed up. While the congratulations message uses the default semantics for conditionals, the speed up message uses an explicit conditional `(Speed < 4).Pr(0.9)`. This increased probability threshold prefers false negatives over false positives, to ensure we do not unfairly chastise a user due to random error.

Figure 5 shows data from this improved version of the GPS fitness application. The blue line shows the same data as in Figure 1, but *Uncertain⟨T⟩*'s propagation of error through computations means we can now provide a 95% confidence interval, showing that the data is very inaccurate.

Because *Uncertain⟨T⟩* encapsulates distributions, more knowledgable programmers (such as those implementing the platform's GPS library) can use Bayesian statistics to improve the quality of data. In Figure 5, the red data reflects the same GPS data but with a physics model applied using Bayesian statistics. This model eliminates the major errors by noting that humans cannot walk at such absurd speeds. Unlike existing work, *Uncertain⟨T⟩* enables such corrections without resorting to ad hoc heuristics.

4.2 Uncertain<T> for Digital Sensors

One shortcoming of studying GPS is the lack of ground truth data for comparison. This case study artificially injects Gaussian noise into a binary sensor to show how *Uncertain⟨T⟩* can improve the accuracy of sensor applications.

We take Conway's Game of Life, a cellular automaton that operates on a two-dimensional grid of cells that are each either dead or alive in each generation. To move to the next generation, each cell senses the current states of its neighbours and applies simple rules to the results to decide if the cell should be alive in the next generation. We inject zero-mean Gaussian random noise $N(0,\sigma)$ into the result of each sensor, where σ is the amplitude of the noise, so that each sensor now returns a real number.

The rules of the Game of Life compare the number of live neighbours to different thresholds; for example, if a live cell has less than two live neighbours, it dies in the next generation to simulate underpopulation. The number of live neighbours is the sum of the sensor values from sensing each neighbour. We define three versions of our noisy Game of Life: NaiveLife uses a single noisy value from each sensor, SensorLife applies *Uncertain⟨T⟩* to each sensor, and BayesLife uses Bayesian statistics to improve the results.

We compare these three noisy versions against non-noisy (ground truth) results from the same game grid, and count the number of incorrect survival decisions each version makes.

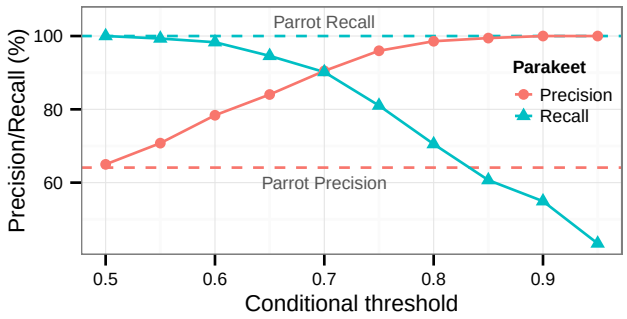


Figure 7. *Uncertain⟨T⟩* helps to balance false positives and negatives in machine learning.

Figure 6 shows the results across different values of the noise amplitude σ on the x-axis. NaiveLife has a constant error rate of 8%. SensorLife wraps each sensor with *Uncertain⟨T⟩*, and so is able to mitigate the error at low noise levels by exploiting hypothesis testing and guided sampling. BayesLife builds on SensorLife's use of *Uncertain⟨T⟩* with Bayesian statistics, observing that underlying each noisy sensor is an (unknown) boolean value, and makes no mistakes at all. These results quantitatively demonstrate that *Uncertain⟨T⟩* can improve the accuracy of sensor applications.

4.3 Uncertain<T> for Machine Learning

Machine learning algorithms, which predict the value of an unavailable function, produce uncertain output. One particular source of uncertainty is *generalisation error*, caused by the finite set of training data not covering all possible inputs. For example, Parrot trains a neural network to approximate the Sobel operator $s(p)$, commonly used in image processing for edge detection [6]. The resulting network achieves an average error of 3% on an evaluation set, suggesting that this predictor generalises well to unseen data.

As with other sources of uncertainty, generalisation error compounds under computation. On the same neural network and evaluation data set as above, the conditional $s(p) > 0.1$ has a 36% false positive rate. To better reason about generalisation error, machine learning practitioners apply Bayesian methods, which consider a *posterior predictive distribution* of potential predictions, weighted by the training performance of the predictor(s) that generate them.

We applied this Bayesian method to the Sobel operator example, using the hybrid Monte Carlo algorithm to sample from the posterior predictive distribution [12]. The prediction function returns an instance of *Uncertain⟨T⟩* that encapsulates the posterior predictive distribution, allowing programmers to compute and ask questions about the prediction of the algorithm while propagating uncertainty correctly. In the spirit of Parrot [6], we call our framework Parakeet.

We used this framework to evaluate the same conditional $s(p) > 0.1$ that returned 36% false positives. We also varied the probability threshold α in the conditional $\Pr[s(p) > 0.1] > \alpha$ to examine its effect on false positives and negatives. Figure 7 shows the results, with α on the x-axis and precision/recall on the y-axis. Parrot fixes a single balance between precision (false positives) and recall (false negatives) because it learns only a single predictor. In contrast, Parakeet uses *Uncertain⟨T⟩* to explore a frontier of precision-recall trade-offs, and *Uncertain⟨T⟩*'s semantics place the selection of this trade-off in the hands of the programmer.

5. Conclusion

Making decisions under uncertainty is difficult, and many disciplines have evolved sophisticated techniques for such problems. Unfortunately, programmers are unlikely to be experts in these techniques, and yet increasingly confront the same problems. Existing solutions focus on the needs of experts, and leave non-experts stranded and forced to ignore uncertainty in the name of convenience. It is time for the programming language community to bear the burden of providing sound, thoughtful abstractions for these problems.

Our experience with *Uncertain⟨T⟩* suggests it is such an abstraction. With a new semantics for computation and conditionals, and implementation insights in sampling, our case studies demonstrate that *Uncertain⟨T⟩* balances simplicity and efficiency with expressiveness to solve real problems.

References

[1] J. Borgström, A. D. Gordon, M. Greenberg, J. Margetson, and J. Van Gael. Measure transformer semantics for Bayesian machine learning. In *ESOP 2011*.

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

[3] 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.

[4] M. Carbin, S. Misailovic, and M. C. Rinard. Verifying quantitative reliability of programs that execute on unreliable hardware. In *OOPSLA 2013*.

[5] A. T. Chaganty, A. V. Nori, and S. K. Rajamani. Efficiently sampling probabilistic programs via program analysis. In *AISTATS 2013*.

[6] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. In *MICRO 2012*.

[7] 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.

[8] 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.

[9] N. D. Goodman, V. K. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum. Church: A language for generative models. In *UAI 2008*.

[10] 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.

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

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

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

[14] N. Ramsey and A. Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *POPL 2002*.

[15] S. Thrun. Towards programming tools for robots that integrate probabilistic computation and learning. In *ICRA 2000*.

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

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