# Uncertain<T>: Abstractions for Uncertain Hardware and Software

Building correct, efficient systems that reason about the approximations produced by sensors, machine learning, big data, humans, and approximate hardware and software requires new standards and abstractions. The Uncertain<T> software abstraction aims to tackle these pervasive correctness, optimization, and programmability problems and guide hardware and software designers in producing estimates.

James Bornholt
University of Washington
Todd Mytkowicz
Kathryn S. McKinley
Microsoft Research

Computing has entered the era of *uncertain* data, in which hardware and software generate and reason about estimates. New hardware sensors, such as those found in smartphones, fitness devices, cars, homes, and games, observe the physical world around them. Approximate computing deliberately exploits software robustness and unreliable hardware in the name of efficiency. Analog and neuromorphic systems perform computation on new hardware substrates. Machine learning helps make sense of large, complex data problems. Speech recognition, natural language processing, and other human–computer interactions face the ambiguity of human input. These data sources already produce estimates that millions of people rely on daily—but can we trust them?

Despite their ubiquity, economic significance, and societal impact, building applications using these uncertain data sources is surprisingly ad hoc. Most current software and hardware abstraction layers ignore the error in estimates, which leads to *uncertainty bugs*. One potential solution that researchers are exploring is probabilistic programming languages,[1] which provide abstractions for reasoning about uncertainty, but these languages are intended for programmers with statistical expertise. The richness and generality of these languages poses a high barrier to entry for programmers who lack such expertise. More broadly, the wide ranging and increasing use of estimates in modern software pose correctness, optimization, and programmer productivity problems that current programming languages do not adequately address.

Here, we describe Uncertain<T>, a simple programming language abstraction that lets programmers without statistics expertise easily and correctly compute with estimates. Uncertain<T>'s semantics automatically propagate uncertainty in an estimate through computation on that estimate and define a statistical interpretation for conditionals that compute with uncertain values. The Uncertain<T> runtime lazily evaluates


Published by the IEEE Computer Society

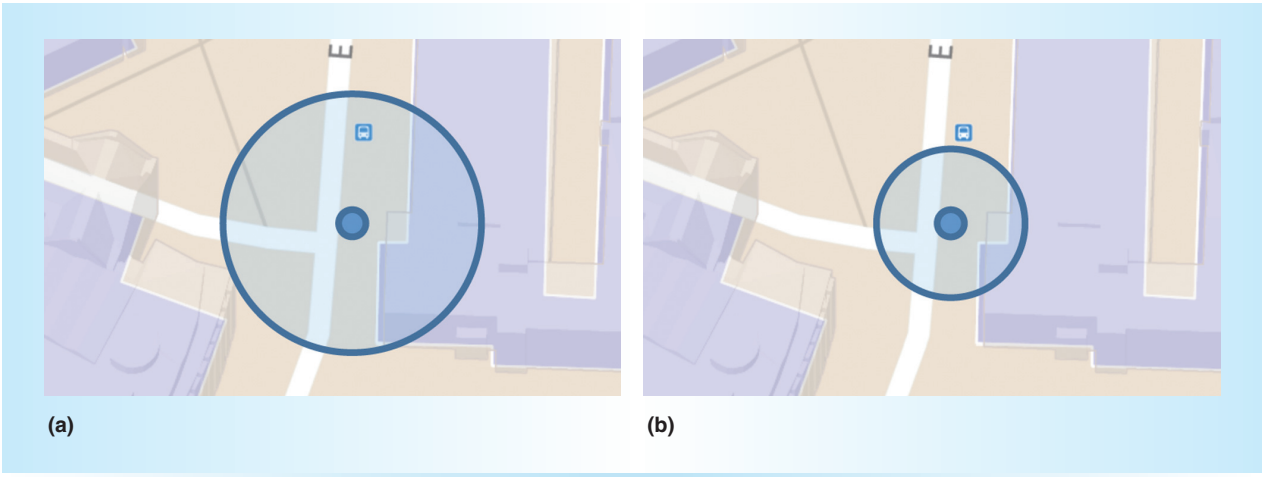


Figure 1. GPS at the same location on two smartphone platforms. (a) One operating system defines horizontal accuracy as a 95 percent confidence interval. (b) Another operating system defines horizontal accuracy as a 68 percent confidence interval. The larger circle is actually more accurate.

computations on estimates and uses hypothesis tests to compute only as much precision as necessary for a particular operation.

Uncertain<T> is less expressive than other probabilistic programming languages, because its goal is a simple API that abstracts the complexities of generating, computing with, and reasoning about estimates. We carefully chose its features such that programmers need not have deep knowledge of statistics when using it, thus improving accessibility and programmer productivity. Our programming model increases the demands on hardware and software systems that produce estimates, whose libraries and APIs must now encode estimates as probability distributions. In return, Uncertain<T> delivers simplicity and accuracy to programmers who consume these estimates.

## Three perils of uncertainty

Current APIs and programming languages encourage programmers to ignore uncertainty and introduce errors in their programs. As a running example, we consider GPS sensors. APIs for GPS typically return a position and an estimated error radius (a confidence interval for location):

```
public double Latitude,
Longitude; // location
public double Horizontal-
Accuracy; // error estimate
```

This interface, like many widely used APIs for estimates, obscures the GPS estimate's uncertainty, encouraging three types of uncertainty bugs:

- *Using estimates as facts* ignores random noise in data and introduces errors.
- *Computation compounds these errors* because computations on uncertain data compose the uncertainty of their inputs.
- *Conditionals ask Boolean questions* of probabilistic data, leading to false positives and false negatives.

Uncertain<T> is a programming language abstraction that helps programmers to identify, reason about, and fix these common bugs.

### Using estimates as facts

Current abstractions encourage programmers to ignore uncertainty. For GPS, a lack of standardization makes results hard to interpret. Consider the two smartphone operating systems shown in Figure 1, both of which depict GPS data with a point and a horizontal accuracy circle for the confidence interval. Intuitively, smaller circles should indicate less uncertainty. However, the operating system in Figure 1a defines horizontal accuracy as a 95 percent confidence interval (widely used for statistical confidence), whereas the operating system in Figure 1b defines it as a 68 percent confidence interval (one standard deviation of a Gaussian). So,

## The Need for Standardization

As computer hardware diversified and proliferated in the early 1980s, software that used floating-point values was often incorrect, unreliable, and not portable because hardware and software had not agreed on their semantics. Codifying the IEEE Floating Point Standard in 1985 was wildly successful in delivering programmability, reliability, and portability of applications that reasoned and computed with floating-point numbers.

Computing is at a similar point in its history for computing with estimates. Hardware and software are increasingly producing, computing, and reasoning about diverse estimates without the appropriate specifications of error distributions and a corresponding programming model. Uncertainty is increasingly exposed in emerging fields such as sensor processing, approximate computing, machine learning, and neuromorphic engineering. The devices we design and manufacture are increasingly complex, and yet depend on a growing community of nonexpert programmers for their success. The more programmable we make devices for nonexpert programmers who lack statistical expertise, the more likely we are to see innovation that exploits computing with these devices. Much previous work takes an ad hoc approach to programming these devices, which ignores uncertainty and error in the name of convenience.

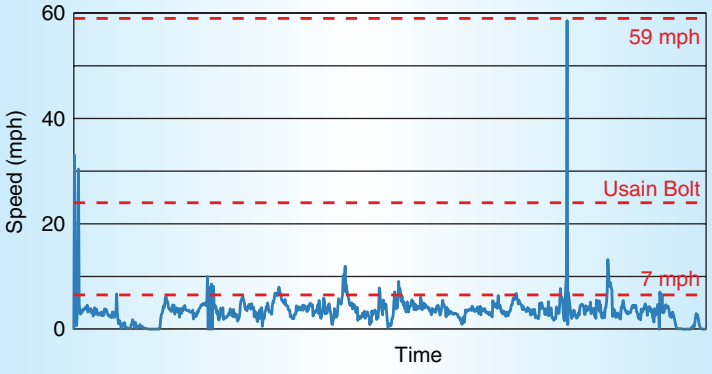


Figure 2. Walking speed computation on GPS data. At one point, the data presents an absurd 59 mph walking speed, and, for 35 seconds, the calculated walking speed is greater than 7 mph (a running pace).

the smaller circle has a higher standard deviation and thus is actually less accurate! This single horizontal accuracy number is insufficient to characterize uncertainty without knowing statistical details of its definition. Indeed, as our earlier work shows,[2] a circle—the standard visualization of GPS uncertainty—obscures the fact that a person is not uniformly likely to be at each point within the circle.

Such common APIs obscure errors and are often undocumented. Furthermore, the lack of an API standard for each type of sensor, much less a cohesive API for different sensors, makes their results hard to interpret, impeding portability and programmability. Together, these factors make programming sensors challenging and encourage programmers to ignore uncertainty completely. The sidebar, "The Need for Standardization," discusses these issues in more detail.

### Computation compounds errors

To show the practical implications of how computation compounds error, we created a simple GPS smartphone application that computes speed from two GPS location readings and prints the output. We took the phone for a walk at speeds of less than 5 mph; Figure 2 shows the results. The application reported an absurd speed of 59 mph in one case, and often reported speeds of greater than 7 mph (a running pace). Compounding error causes these absurd results. Calculating speed from two GPS location readings amplifies the error in the result. Figure 3 illustrates compounded error from calculations using Gaussian error distributions. Summing two Gaussians $a$ and $b$ produces a result $c$ with higher variance than either input.

Current programming models do not represent distributions as first-class values or propagate compounding uncertainty through computations, making it difficult for programmers to reason about and correct errors. Programmers who want to improve application accuracy are instead forced to implement a wide range of ad hoc approaches, such as averaging or imposing filters (for example, Usain Bolt's world record running speed is 24 mph, so no one is likely to walk faster than 24 mph).

### Conditionals

Programs eventually act on estimated data with conditionals. As an example, consider

using GPS to issue speeding tickets for a 60 mph speed limit with the conditional Speed > 60. If your actual speed is 57 mph and GPS accuracy is 4 meters, our GPS error model shows this conditional has a 32 percent probability of issuing a ticket due to random noise alone. Figure 4 shows this probability across speeds and GPS accuracies. Naive conditionals ignore the potential for random error, leading to false positives and negatives. Applications instead should ask probabilistic questions. In this example, we might prefer to issue a ticket only if the probability is very high that the user is speeding according to the available GPS evidence.

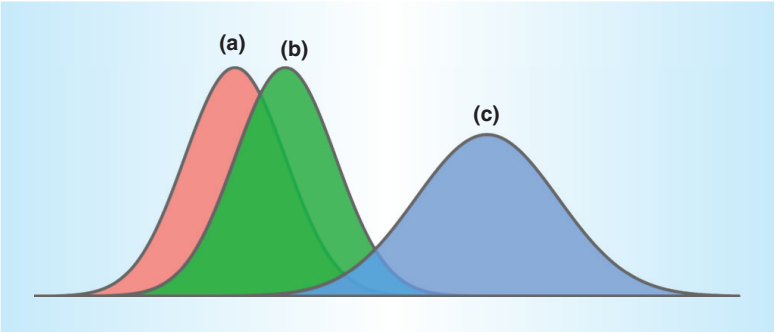


Figure 3. The compounding error using Gaussian error distributions. The sum $c = a + b$ is more uncertain than $a$ or $b$.

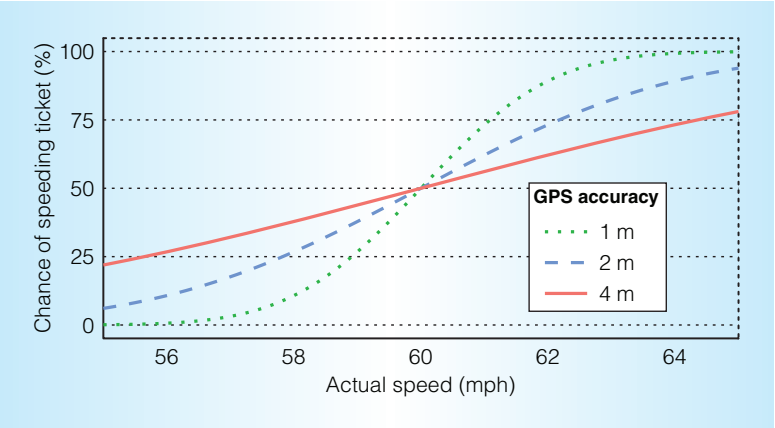


Figure 4. Error model for GPS-issued speeding tickets. Testing the conditional Speed > 60 with a true speed 57 mph and GPS accuracy of 4 meters issues a speeding ticket 32 percent of the time.

## Uncertain<T>

To overcome the perils of uncertain data, our earlier paper introduced Uncertain<T>, a generic data type that expresses, propagates, and manipulates uncertain data.[2] Programmers who consume estimates with Uncertain<T> use familiar syntax, while the type's semantics (and implementation) make their programs accurate and efficient. We implemented Uncertain<T> in C#, as well as developing prototypes in C++ and Python, and believe most other high-level languages would also support the abstraction. Figure 5 shows a simple GPS fitness application, GPS-Walking, in C# without and with Uncertain<T>, illustrating the minor syntax differences.

### Syntax

Uncertain<T> uses operator overloading on a base type $T$ to define an algebra over random variables and to propagate uncertainty through computations. Table 1 shows Uncertain<T>'s basic operators and methods. Programmers write computations with type Uncertain<T> as they would with type $T$. The runtime computes how uncertainty in an estimate flows through computations. For example, in the program in Figure 5b, the line `Uncertain<double> Speed = Distance / dt;` computes an estimate of speed from an estimate `Distance` of distance and time `dt`.

### Defining distributions

Uncertain<T> represents uncertainty as probability distributions that assign a probability to every possible value of a variable. To represent distributions, Uncertain<T> uses *sampling functions,* which approximate distributions by random sampling. A sampling function is a no-argument function that returns a new random sample from the distribution it represents on each invocation.[3] In some cases, library writers might instead write closed-form distributions for their data, which can improve efficiency and accuracy.

However, sampling functions make it possible to express complex data, such as maps and machine-learning algorithms, which have no closed form. Sampling functions thus deliver both expressiveness and efficiency benefits.

Library writers must modify their code by first defining an error distribution for their estimate and then returning an Uncertain<T>

## Related Work on Probabilistic Languages

Probabilistic programming languages help experts build probabilistic models in domains such as machine learning, cognitive science, and robotics. None of this existing work, however, addresses the needs of application programmers who lack statistical expertise. Our earlier paper describes existing domain-specific approaches in robotics, approximate computing, and databases,[1] but those approaches lack the generality of our work.

### Probabilistic programming

Various languages—such as Bayesian Inference Using Gibbs Sampling (BUGS),[2] Church,[3] Fun,[4] and Integrated Bayesian Agent Language (IBAL),[5]—explore probabilistic programming, in which the programs specify probabilistic models and the language provides inference algorithms for querying these models. Using these languages requires programmers to carefully specify a model structure, inputs, and appropriate inference strategies. Such delicate tasks require a facility with statistics beyond what most programmers possess. However, for those with this expertise, probabilistic programming languages deliver considerable productivity and efficiency improvements by abstracting the details of implementing complex statistical models.

Uncertain<T> is, by design, not as expressive as these probabilistic programming languages. It supports only tree-shaped Bayesian networks, in contrast to the arbitrary joint-probability distributions that other probabilistic languages support. This restriction is a feature that we exploit to define an efficient implementation. Programs with Uncertain<T> often have only minor syntactic differences from the same program without Uncertain<T>, easing adoption by nonexpert programmers while delivering concrete accuracy benefits.

We exploit sampling functions to represent probability distributions in the same fashion as Park et al.,[6] but add the semantics necessary to use sampling functions in a mainstream imperative programming language. We developed an automatic inference algorithm that uses sequential hypothesis tests to dynamically choose a sample size. In contrast, Park et al. require the programmer to specify a sample size manually and interpret the sampling results themselves.

### Probabilistic semantics

The semantics of a program operating on estimates are not the same as the semantics of a program operating on exact data, because the former program's desirable correctness properties are often probabilistic. For example, a smartphone application might obfuscate a user's location by adding Gaussian random noise, and then assert that the obfuscated location is within 10 meters of the true location so the results are still useful. There is a non-zero probability that this assertion is violated. A more useful correctness property for this program is probabilistic: there is a *high probability* the obfuscated location is near the true location.

Sampson et al. show how to exploit Uncertain<T>'s Bayesian network representation to verify such probabilistic assertions, as well as optimize program execution by applying statistical transformations.[7] These opportunities are available only because of the probabilistic semantics that Uncertain<T>'s Bayesian network representation offers programs. Given these results, future work could perform additional statistical optimizations based on probabilistic semantics.

### References


1. J. Bornholt, T. Mytkowicz, and K.S. McKinley, "Uncertain<T>: A First-Order Type for Uncertain Data," *Proc. ACM Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS), 2014, pp. 51–66.
2. W.R. Gilks, A. Thomas, and D.J. Spiegelhalter, "A Language and Program for Complex Bayesian Modelling," *J. Royal Statistical Society*, Series D, vol. 43, no. 1, 1994, pp. 169–177.
3. N.D. Goodman et al., "Church: A Language for Generative Models," *Proc. Conf. Uncertainty in Artificial Intelligence* (UAI), 2008, pp. 220–229.
4. J. Borgström et al., "Measure Transformer Semantics for Bayesian Machine Learning," *Proc. Euro. Symp. on Programming* (ESOP), 2011, pp. 77–96.
5. A. Pfeffer, "IBAL: A Probabilistic Rational Programming Language," *Proc. Int'l J. Conf. Artificial Intelligence* (IJCAI), 2001, pp. 733–740.
6. S. Park, F. Pfenning, and S. Thrun, "A Probabilistic Language Based on Sampling Functions," *Proc. ACM Symp. Principles of Programming Languages* (POPL), 2005, pp. 171–182.
7. A. Sampson et al., "Expressing and Verifying Probabilistic Assertions," *Proc. ACM Conf. Programming Language Design and Implementation* (PLDI), 2014, pp. 112–122.


variable that encapsulates that error. For GPS, we took on the role of expert library writer. We analyzed the GPS sensor specification and derived a model for its error (see our earlier paper[2] for details). The error model follows a Rayleigh distribution; Figure 6 shows how we implement a sampling function for this distribution.

`GPS.GetLocation` returns an instance of `Uncertain<GeoCoordinate>` by implementing a sampling function that draws samples from the distribution of GPS error.

**Table 1. Uncertain<T> (U<T>) operators and methods.**

| Language construct | Type |
|---|---|
| ***Operators*** | |
| Math (+ - * /) | op :: U<T> → U<T> → U<T> |
| Order (< > ≤ ≥) | op :: U<T> → U<T> → U<Bool> |
| Logical (& \|) | op :: U<Bool> → U<Bool> → U<Bool> |
| Unary (!) | op :: U<Bool> → U<Bool> |
| Pointmass | Pointmass :: T → U<T> |
| ***Conditionals*** | |
| Explicit | Pr :: U<Bool> → [0,1] → Bool |
| Implicit | Pr :: U<Bool> → Bool |
| ***Evaluation*** | |
| Expected value | E :: U<T> → T ± 95 percent confidence |

Although the Uncertain<T> runtime will typically use this sampling function to draw many samples from the error distribution, the GPS sensor's functions (`GetHardwareLocation` and `GetHardwareAccuracy`) are invoked only once, and their results are captured by the `SamplingFunction` closure, so only one observation is taken from the GPS sensor. Writing the GPS sampling function is not onerous given a basic understanding of GPS, which most library writers are likely to already possess.

## Computing with distributions

Uncertain<T> propagates uncertainty by dynamically constructing a Bayesian network representation of computations involving uncertain variables. A Bayesian network is a probabilistic graphical model—a directed acyclic graph whose nodes are random variables and whose edges are conditional dependences between those variables.[4] For example, Figure 7 shows the Bayesian network representation for a single iteration of the main loop in Figure 5b. The shaded leaf nodes are known distributions, either provided by data sources (for the locations `L1` and `L2`) or by a standard set of distributions (`dt` is a constant, which is a point mass distribution). Each leaf must provide a sampling function for its distribution. Computations do not evaluate eagerly; only conditionals and expected value operators trigger Bayesian network evaluation.

As we discussed earlier, instances of Uncertain<T> must define a sampling function. The Bayesian network representation defines sampling functions for computations by using *ancestral sampling*. Consider again the speed calculation in GPS-Walking `Uncertain<double> Speed = Distance / dt;` Figure 7 shows the Bayesian network. The sampling function for `Speed` draws two samples, one $d$ from `Distance` and one $t$ from `dt` (a point mass distribution, so all samples are equal). To draw the sample $d$ from `Distance`, the process recurses, drawing a sample each from `L1` and `L2` (which return location samples), and applying the `GPS.Distance` operation to the two sampled locations. Finally, at the root, the sampling function applies the division operation to the samples, returning `d/t`, a sample of `Speed`.

## Making decisions under uncertainty

Uncertain<T>'s host languages require concrete decisions at conditionals. In GPS-Walking, the conditional `if (Speed > 7) ...` must decide whether or not to enter this branch based on `Speed`, a probability distribution of type `Uncertain<double>`. The possible values of `Speed` may include values both less than and greater than 7. Should we enter the branch?

To give meaningful semantics to conditionals involving probability distributions, Uncertain<T> conditionals evaluate evidence for a conclusion. The Uncertain<T> runtime compares the probability (Pr) that `Speed` is greater than 7—that is, $\Pr[Speed > 7]$—to a threshold $\alpha$. We choose a default threshold of 0.5. The conditional given above enters the branch if $\Pr[Speed > 7] > 0.5$ or, in other

```
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 > 7)
         Alert("You're running!");
       L1 = L2; // Last Location = Current Location
}
```

**(a)**

```
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 > 7)
         Alert("You're running!");
       L1 = L2; // Last Location = Current Location
}
```

**(b)**

Figure 5. A simple GPS fitness application in C#. GPS-Walking computes speed and warns users who are moving too fast. (a) Without Uncertain<T>. (b) With Uncertain<T>.

words, if it is more likely than not that `Speed` is greater than 7.

Programmers can override the threshold, which controls the tradeoff between false positives and false negatives. The right tradeoff is application-dependent. To override the default threshold, a programmer writes `if ((Speed > 7).Pr(0.9))` ... to compare the probability to 0.9 (90 percent). Higher thresholds require stronger evidence to say yes, reducing false positives at the expense of more false negatives.

### Implementing accurate and efficient evaluation

To decide conditionals, Uncertain<T> must evaluate probabilities such as Pr[*Speed*>7]. This evaluation turns random samples from the Uncertain<T> variable's sampling function into a concrete value, which will necessarily be approximate due to sampling error. Conventional approaches in probabilistic programming languages do not abstract away the details of when and how to do this evaluation. For example, some systems require a programmer to fix a sample size $n$ for each evaluation, or to select an appropriate inference engine. Uncertain<T> automates these delicate statistical choices.

When evaluating an inequality of the form Pr[*Speed*>7]>0.5, we need to compute the left side only to a level of precision high enough to be confident it is significantly different from the right side. This precision level could be very low if, for example, *Speed* were a Gaussian with mean 20 and variance 1. Uncertain<T> exploits this context sensitivity using sequential hypothesis tests, which are hypothesis tests that take only as many samples as necessary to answer the particular conditional.

The Uncertain<T> runtime performs the hypothesis test using Wald's sequential probability ratio test (SPRT).[5] We specify a step size, say $k = 10$, and start by drawing $n = k$ samples from the Bernoulli distribution *Speed* > 7. The runtime applies the SPRT to these samples to decide if the probability Pr[*Speed* > 7] is significantly different from 0.5. If so, it terminates immediately and branches accordingly. If the result is not significant, the runtime draws an additional $k$ samples and repeats the test. The runtime continues this process until either a significant result is achieved or a maximum sample size is reached (to ensure termination).

We can't apply the same sequential testing approach to the expected value operator $E$ because there are no alternatives to compare against. Instead, we report the sampling error of the mean, which is Gaussian for large enough $n$ by the central limit theorem. Capturing the sampling error ensures programmers understand that the mean of the distribution is approximate and subject to error, while allowing them to cast away uncertainty if necessary for a particular use (such as a library that expects a concrete value of type *T*).

## Uncertain<T> in practice

We now offer two case studies—GPS sensors and approximate computing with machine learning—to illustrate how programming with

Uncertain<T> is accessible to nonexperts and delivers accuracy and efficiency. Our earlier work offers a third case study on digital sensors.[2] As both of the following case studies show, Uncertain<T> helps programmers recognize that uncertainty's effect on programs is inevitably global, tainting all computations that derive from uncertain values.

## Smartphone GPS fitness

Thousands of smartphone applications use the GPS sensor, and many compute distance and speed from GPS readings. For this case study, we continue to use Figure 5's GPS-Walking with and without Uncertain<T>. As Figure 6 shows, the version with Uncertain<T> uses the modified GPS library. Here, we evaluate GPS-Walking, showing how the Uncertain<T> version improves application correctness by eliminating uncertainty bugs, while requiring little programmer effort.

GPS-Walking uses locations from the GPS library to calculate the user's speed, since $Speed = \Delta Distance / \Delta Time$. Because the locations are estimates, the distance and speed are as well. The programmer must change the line `print("Speed: " + Speed);` from Figure 5a's original program because `Speed` now has type `Uncertain<double>`. The easiest change is to instead print the speed distribution's expected value `Speed.E()`. It might also be desirable to print the 95 percent confidence interval.

*Evaluation.* We tested GPS-Walking by walking outside for 15 minutes. Figure 8 shows the expected value `Speed.E()` (dark blue line) and the confidence interval for `Speed` (light blue ribbon) as measured each second by the application. The speed calculation's uncertainty, with extremely wide confidence intervals, explains the absurd values highlighted earlier in Figure 2.

Because GPS-Walking is for tracking walking speeds, it warns users who are running by comparing the user's speed to 7 mph. The original implementation (Figure 5a) uses naive conditionals, which are susceptible to random error. On our test data, our user was walking the entire time, yet the naive application triggered this warning 30 times, representing a period of 30 seconds of false positives due to random error.

```
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;
        // Sample the distribution in 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 6. The Uncertain<T> version of `GPS.GetLocation` returns an instance of `Uncertain<GeoCoordinate>`. The sampling function samples the error distribution for a GPS reading, which follows a Rayleigh distribution.

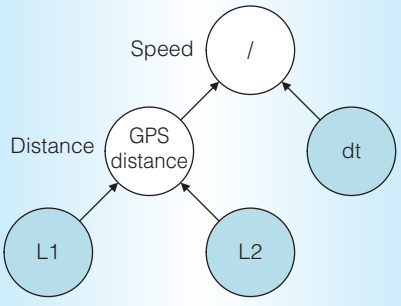


Figure 7. Bayesian network for an iteration of the Uncertain<T> version of Figure 5b's GPS-Walking. The leaf nodes `L1` and `L2` are from the GPS library in Figure 6, and the leaf node `dt` is a constant.

The Uncertain<T> version of GPS-Walking in Figure 5b evaluates evidence to execute conditionals. When `Speed` has type `Uncertain<double>`, the conditional

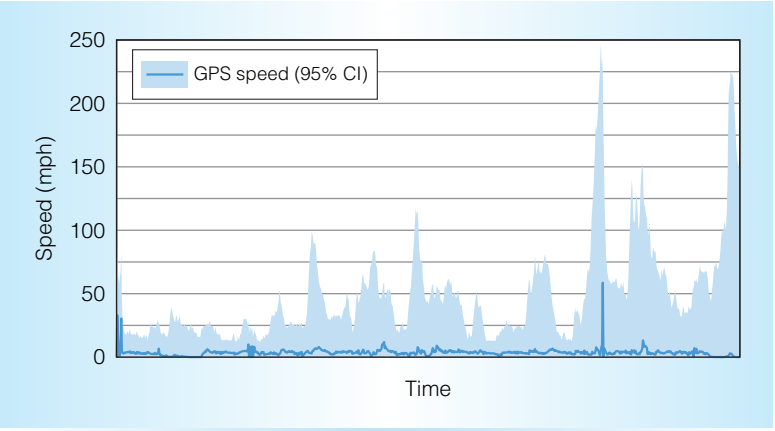


Figure 8. Data from the GPS-Walking application. The ribbon shows the error (95 percent confidence interval). When the error is high, GPS is not trustworthy.

```
if (Speed > 7)
Alert("You're running!");
```

asks if it is more likely than not that the user is running. Even this simple semantics improves the accuracy of GPS-Walking. On our test data, this version triggers the warning only four times. Programmers who want to further limit false positives write

```
if ((Speed > 7).Pr(0.9))
Alert("You're running!");
```

which asks if there is at least a 90 percent chance the user is running faster than 7 mph. This requirement is stricter than the first conditional and never triggers the warning on our test data. The tradeoff is that this conditional is more likely to cause false negatives. The right balance is application-specific.

*Lessons.* Uncertain<T>'s semantics for conditionals that operate on estimates help programmers identify noisy, untrustworthy results and thus substantially improve the accuracy of their applications. The programmer need only make minimal changes to the original application to be rewarded with these benefits. Without Uncertain<T>, this complex logic is difficult to implement, because programmers must know the error distribution for GPS, how to propagate error through calculations, how to interpret conditionals with uncertain data, and how many samples to take.

## Approximate computing with machine learning

The promise of approximate computing is to improve efficiency by exploiting the robustness of some applications to output inaccuracies. Recent work on approximate computing proposes *neural acceleration,* in which a function is replaced with a neural network created using a machine-learning algorithm that approximates the function.[6] The neural network executes on specialized hardware, making the program faster and more efficient.

*Approximating Sobel.* We studied the Sobel benchmark,[6] an image-processing operator that calculates the gradient of image intensity at a pixel. Esmaeilzadeh et al. trained a neural network to approximate Sobel with an average error of 3.4 percent. Although this error seems low, it can still adversely impact a program's correctness when used in computation. For example, edge detection compares the output $s(p)$ of the Sobel operator to a threshold (say 0.1) to decide if pixel $p$ is an edge. Despite the low error in the approximation of $s(p)$, our experiments show that when the program computes with the result in the conditional if `(s(p) > 0.1) …`, it suffers a 36 percent false-positive rate.

Machine-learning algorithms such as neural networks estimate a function's true value. One source of uncertainty in their estimates is generalization: predictions might be good on training data but suffer errors on unseen data. To combat this, we use Bayesian machine learning, which considers a *distribution* of estimates rather than a single prediction.

We took the expert library writer's role and wrapped the neural acceleration of the Sobel operator in Uncertain<T>. In particular, we trained a Bayesian neural network for the Sobel operator, so that each input $p$ to $s(p)$ produces a distribution of predictions instead of a single one. Figure 9 shows this predication distribution for a single output. It also shows the single prediction from Esmaeilzadeh and colleagues' neural network and ground truth. This example illustrates the need to propagate error distributions through approximate computations, rather than simply trust a single estimate. However, our Bayesian neural networks might not deliver sufficient performance for use in

approximate computing. Our earlier work contains more details and suggests techniques to trade off some accuracy to make Bayesian neural networks fast enough to preserve neural acceleration's performance benefits.[2]

*Testing.* We evaluated our approximate Sobel operator by training it on 5,000 examples and testing it on a separate set of 500 examples. For each test example $p$, we compute the ground truth $s(p) > 0.1$, then evaluate this same conditional using the approximation of $s(p)$ and Uncertain<T>. The Uncertain<T> version decides if $s(p) > 0.1$ by performing a hypothesis test on the inequality $\Pr[s(p) > 0.1] > \alpha$, where $\alpha$ controls the balance between false positives and false negatives.

Figure 10 shows the results of this conditional as we vary the threshold $\alpha$ on the $x$-axis. The $y$-axis plots precision and recall for the test data. Precision is the probability that a detected edge is actually an edge, and thus it describes false positives. Recall is the probability that an actual edge is detected, and thus it describes false negatives. The naive approach using a single neural network locks the programmer into a single balance between precision and recall, decided once at training time, which provides 100 percent recall but only 64 percent precision. Thus, on our application that uses the network on new images, 36 percent of reported edges are false positives.

With Uncertain<T>, programmers can choose a precision and recall balance suitable for their application. For example, Figure 10 shows that a threshold of $\alpha = 0.8$—that is, the code `if ((s(p) > 0.1).Pr(0.8)) …`)—results in 71 percent recall and 99 percent precision, trading more false negatives (missed edges) for fewer false positives (wrongly reported edges).

*Lessons.* Machine-learning techniques such as neural networks are approximate functions and thus introduce uncertainty. Applications compute with these approximate functions, compounding their uncertainty. Although Sobel's neural network had a low error rate, even a simple conditional compounded the error, generating a 36 percent false-positive rate. Using Uncertain<T> lets programmers explicitly express the right balance between false positives and false negatives for their particular application, without requiring any modifications to the data source producing the estimate.

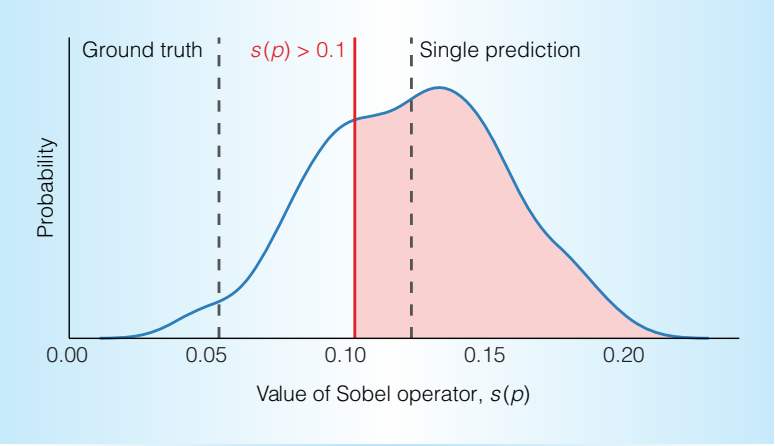


Figure 9. Error distribution for the Sobel operator on a single input. The single prediction is significantly different from the correct value (ground truth), but the error distribution captures the uncertainty of that prediction.

## Discussion: expressiveness and implications

Uncertain<T>'s simple abstraction gives programmers access to a more expressive language to write more sophisticated programs. We envision extensions to Uncertain<T> to further this expressiveness while maintaining its accessibility. We are also excited about the effects of a simple abstraction for uncertainty on the design of future hardware and software systems.

### Expressiveness

Uncertain<T> identifies uncertainty in data and helps programmers reason about its effect on their programs. Often, applications combine disparate data sources to achieve novel features that data sources cannot achieve in isolation. We believe Uncertain<T> can help programmers combine these uncertain data in a declarative way so that they need not understand the statistical details behind the composition. For example, programmers should be able to write `Location = GPS + RoadMap + Acceleration` to improve GPS estimates' accuracy by including road map data and accelerometer readings. To personalize a restaurant recommendation, programmers could write `Restaurants = Zagat(Seattle) + Favorites + Price`

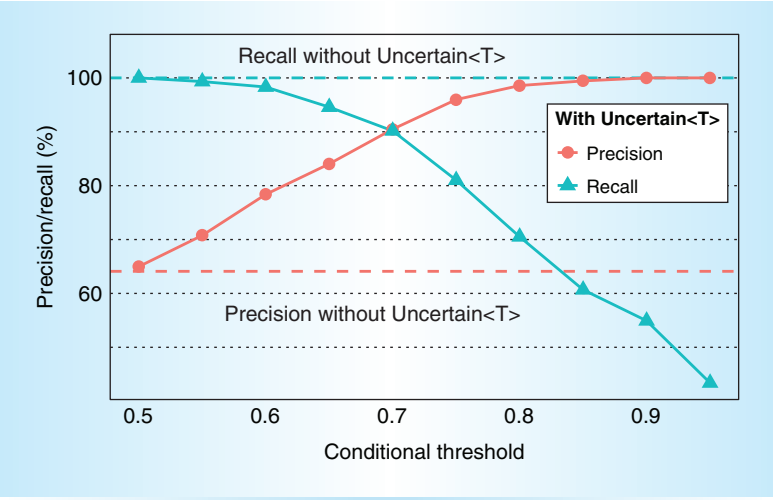


Figure 10. Results of evaluating the conditional $s(p)>0.1$ with and without Uncertain<T>. Using Uncertain<T> helps programmers balance false positives (precision) and recall (false negatives).

and thereby combine general reviews with the user's preferences and a desired price range.

We believe that making it easy to compose disparate data sources is a critical next step in improving the programmability of systems that compute with estimates. The challenge is to take the well-known statistical problem of Bayesian inference and abstract the implementation details so that programmers can benefit from an accurate and efficient program without needing to understand the inference details.

### Implications for hardware and software

In this article, we have focused primarily on an abstraction to meet the needs of software that consumes estimates. However, such an abstraction also has implications for estimate producers and for negotiations between producers and consumers.

The burden on producers is to describe the distribution of error in their hardware or software, which exposes information about the estimation process. In the simplest case, consumers will always benefit from better producers, such as a new, more accurate and efficient sensor. But because consumer applications that use Uncertain<T> are more robust to errors, producers can more freely optimize a range of accuracy and efficiency. For example, the GPS library can degrade its accuracy by contacting the satellites less frequently. Similarly, approximate computing algorithms can degrade their results, because they now describe their error and require consumers to explicitly reason about it. However, when estimates are too inaccurate, they are useless; each application requires some level of accuracy to produce useful results. A robust interface should communicate desirable levels of accuracy to producers, but consumers must nonetheless adapt when that level is not possible.

Architectural advances in sensing, approximation, analog, and neuromorphic computing, and software advances in machine learning, big data, and human-computer interaction, increasingly introduce uncertainty into applications. Uncertain data poses correctness, optimization, and programmability challenges to these applications and their programmers. Rather than ignore these challenges and introduce uncertainty bugs, Uncertain<T> embraces uncertainty, helping programmers understand and control its effect on their applications without demanding statistical expertise. Our experience with Uncertain<T> suggests that it has the potential to change how programmers think about and solve the growing variety of problems involving uncertainty.

MICRO

## Acknowledgments

For helpful feedback and suggestions, we thank Steve Blackburn, Luis Ceze, Dan Grossman, Margaret Martonosi, Madan Musuvathi, Adrian Sampson, Darko Stefanovic, Emina Torlak, Ben Zorn, members of the programming languages and computer architecture groups at the University of Washington, and the anonymous reviewers.

## References

1. A.D. Gordon et al., "Probabilistic Programming," *Proc. Future of Software Eng.*, 2014, pp. 167–181.
2. J. Bornholt, T. Mytkowicz, and K.S. McKinley, "Uncertain<T>: A First-Order Type for Uncertain Data," *Proc. ACM Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS), 2014, pp. 51–66.
3. S. Park, F. Pfenning, and S. Thrun, "A Probabilistic Language Based on Sampling Functions," *Proc. ACM Symp. Principles of*

*Programming Languages* (POPL), 2005, pp. 171–182.

4. C.M. Bishop, *Pattern Recognition and Machine Learning*, Springer, 2006.
5. A. Wald, "Sequential Tests of Statistical Hypotheses," *Annals Mathematical Statistics*, vol. 16, no. 2, 1945, pp. 117–186.
6. H. Esmaeilzadeh et al., "Neural Acceleration for General-Purpose Approximate Programs," *Proc. IEEE/ACM Int'l Symp. Microarchitecture* (MICRO), 2012, pp. 449–460.

**James Bornholt** is a PhD student in the Department of Computer Science and Engineering at the University of Washington. His research focuses on how programming languages, computer systems, and hardware architectures evolve and cooperate to address the competing demands of performance, energy efficiency, and accuracy. Bornholt has a BPhil in computer science from the Australian National University. Contact him at bornholt@cs.washington.edu.

**Todd Mytkowicz** is a researcher at Microsoft Research. His research focuses on creating abstractions that help programmers easily express complex problems and yet are sufficiently constrained to deliver efficient and powerful implementations. Mytkowicz has a PhD in computer science from the University of Colorado at Boulder. Contact him at toddm@microsoft.com.

**Kathryn S. McKinley** is a principal researcher at Microsoft Research. Her research focuses on creating systems (programming languages, compilers, runtimes, and architectures) that make programming easy and the resulting programs efficient. McKinley has a PhD in computer science from Rice University. She is a fellow of IEEE and the ACM. Contact her at mckinley@microsoft.com.