Programming the Internet of Uncertain <T>hings

James Bornholt
University of Washington

Na Meng
The University of Texas at Austin

Todd Mytkowicz
Microsoft Research

Kathryn S. McKinley
Microsoft Research

***Abstract*—The transformation from desktops and servers to devices and cloud services—the Internet of things (IoT)—is well underway. A key problem facing IoT applications is their increasing reliance on estimated data from diverse sources, such as sensors, machine learning, and human computing. Current programming abstractions treat these estimates as if they were precise, creating buggy applications. Existing approaches to mitigate noise in estimates either add naive ad-hoc filters or construct sophisticated statistical models. They are either too fragile or too complex for ordinary developers to use. IoT developers need abstractions that help them (1) reason about estimates, because they are noisy and inherently inaccurate; (2) trade accuracy for energy efficiency on battery limited devices; and (3) compose data from disparate sources on devices and in the cloud.**

In prior work, we proposed a new programming abstraction called $Uncertain\langle T\rangle$ to help developers reason about noise in estimates. This abstraction embeds statistical reasoning into programming language semantics for commonly used languages, such as C++, C#, and Java, instead of making developers program their own statistics. This paper further improves programmability and accuracy for estimates by (1) providing two program constructs to specify *context*—application-specific domain knowledge—and showing how improving estimates *requires* such context; and (2) implementing a runtime which automatically composes context with estimates. A case study shows that developers easily use our constructs to add context and improve application accuracy. This paper motivates the need for programming abstractions for estimates to build correct IoT applications, and shows how they make IoT programming more accessible to a wider class of developers.

I. Call to Action

We believe that hardware and software that produce estimates and the systems that consume them are in desperate need of a standard for describing estimates. Put another way, what good is a sensor if a programmer cannot reason about whether their use of that sensor is correct? Consider the IEEE Floating Point standard as an analogy. As computer hardware proliferated in the late 1970s and early 1980s, software that computed and reasoned about floating point values was often incorrect, unreliable, and not portable. Codification of the IEEE Floating Point Standard in 1985 for hardware and software 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 uncertain data, and this inflection point is being accelerated by IoT systems. Hardware and software are producing, computing with, and reasoning about estimates without the appropriate specifications of error distributions and programming models. This paper suggests one programming model the industry could standardize on, in which all systems that produce estimates should produce both the estimate and a model of error in that estimate. We believe this standardization will accelerate innovation in IoT and other applications that consume estimates.

II. Introduction

The explosion of the Internet of Things (IoT) has driven adoption of new *estimated* data sources—both big (machine learning and cloud applications) and small (mobile and sensors). For example, navigation applications turn maps and location estimates from hardware GPS sensors into driving directions; speech recognition turns an analog signal into a likely sentence; search turns queries into information; survey results from sources such as Mechanical Turk must account for human error; and recent research promises approximate hardware and software will trade result quality for energy efficiency. Millions of people rely in their daily lives on these new, connected, ubiquitous IoT applications that use estimates.

Despite their ubiquity, economic significance, and societal impact, the handling of estimates in such applications is surprisingly poor. Most current software and hardware abstraction layers simply ignore or obscure errors in their estimates. Some developers define ad-hoc filters that ignore data beyond a reasonable range of values, such as discarding negative speed readings on a mobile phone or forcing the GPS user interface to draw points only on roads. Such ad-hoc approaches are fragile and inflexible, forcing upon each developer the tedium of manually choosing and tuning their own filters, often without much success. A few sophisticated developers create statistical models to reason about errors. For example, Newson et al. use map data and a probabilistic model to correct noisy GPS readings in navigation applications [9]. For these developers, researchers are exploring augmenting existing languages with probabilistic primitives to deliver significant productivity benefits [5, 6, 8]. Though probabilistic modeling is effective, it requires considerable statistics expertise from developers, a significant barrier to entry for many developers attracted by the potential for real-world impact with new IoT applications and devices. There has been little focus on the programming model needs of this much larger class of developers.

We believe the right programming model can deliver the effectiveness of probabilistic modeling to this much larger class of developers without requiring statistical expertise. To reduce noise in estimates, probabilistic models generally add *context*—application-specific domain knowledge which describes rules to distinguish signal from noise in estimates. Data sources cannot exploit this context automatically because it is application-dependent; for example, a driving navigation application wants GPS data restricted to roads, but a fitness application does not. The right abstraction can provide developers the ability to specify context without needing to build their own probabilistic model. Such an abstraction also needs to automate probabilistic inference, which often requires hand-tuning by experts.

In this paper, we propose two programming language constructs for exploiting context, and present their compiler and runtime support. The constructs build on our prior work on $Uncertain\langle T\rangle$, a type system for uncertain data in mainstream languages such as C# and Java, which automatically propagates uncertainty through computation and performs automated hypothesis tests to decide conditionals. The two constructs are a *conditional probability* operator for specifying context and a *Bayesian composition* operator for composing context

with estimates. Developers specify context declaratively from variables specific to their domain. The compiler and runtime implement simple probabilistic inference algorithms that they automatically tune to the specific estimate when it is evaluated in conditional expressions. While previous probabilistic programming languages include these constructs [2, 4, 5], developers who use them must explicitly build probabilistic models and decide on Bayesian inference approaches. Our abstraction takes a middle ground, delivering simple statistical models without requiring developers to reason about statistical details.

This paper describes a three-step recipe for developers to improve application accuracy by exploiting context. First, we rely on expert developers to summarize noisy estimates as probability distributions, information they often already have. Second, when application developers consume estimates, they use the conditional probability construct to encode context—their domain knowledge about the constraints of their specific application. For example, a navigation application would use a road map as context, marking GPS readings on roads as more likely to be accurate than those off-road. Third, application developers use the Bayesian composition construct to combine estimates with context.

We evaluate our approach using a real-world Windows Phone application, *Time of my Life*, which automatically assigns semantic labels such as "at work" and "at home" to users' GPS locations using a machine learning model. The original application has poor accuracy because the estimates generated by the machine learning model are often too noisy. We improve the application by leveraging our language constructs to express sophisticated reasoning in a handful of lines of intuitive code. The new application is easy to understand, efficient, and substantially more accurate than directly using noisy data. The programming language extension in this paper is the first to make Bayesian inference accessible to a much larger class of developers, and portend exciting future applications that reason about diverse estimated data.

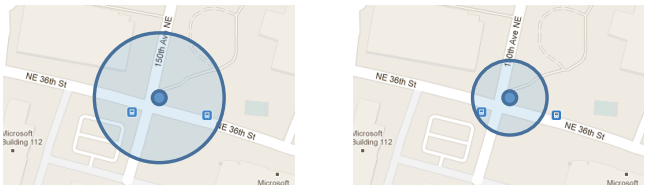
III. Reasoning About Noise With Uncertain<T>

This section provides background on the motivation, type system, and conditional operator in $Uncertain\langle T\rangle$ from prior work [3].

Applications that sense and reason about the complexity of the world use estimates. 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, GPS sensors produce estimates of a user's true location and most modern hardware exposes that estimate as a probability distribution. Figure 1 depicts GPS data with a point and a *horizontal accuracy* 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 so is less accurate; this level of reasoning is too detailed for most programmers. The lack of a standard meaning for estimates and the computations on them is a problem for the systems and humans that consume them.

To explicitly represent and propagate errors in $Uncertain\langle T\rangle$, libraries that generate estimates wrap



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

Fig. 1. GPS samples at the same location on two smartphone platforms. Although smaller circles *appear* more accurate, the sample in (a) is actually more accurate.

their data in the $Uncertain\langle T\rangle$ type and define a sampling function that describes the error in the estimate (e.g., wrapping a single GPS reading [3]). For example, a Gaussian distribution:

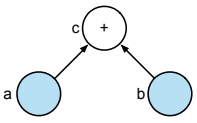
```
Uncertain<double> a = new Gaussian(4, 1);
```

The burden on library writers is low as they often already know the distribution when producing their outputs (e.g., as in GPS). The point of this abstraction is to deliver a familiar and intuitive interface for application developers, i.e., computing with the base type, while under the hood, the compiler and runtime correctly propagate and compute with estimates.

$Uncertain\langle T\rangle$ correctly computes with estimates using lazy evaluation, which is triggered when the program acts on an estimate in a conditional. Conditionals represent hypothesis tests for the runtime to evaluate. An object of type $Uncertain\langle T\rangle$ encapsulates a random variable of the numeric type T. As computation proceeds, the runtime uses the type's overloaded operators to construct a *Bayesian network*, a directed acyclic graph in which nodes represent random variables and edges represent conditional dependences between variables. The leaf nodes of the Bayesian network are data from libraries. 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 the simple Bayesian network on the right, with two leaf nodes and one inner node representing the computation `c = a + b`. $Uncertain\langle T\rangle$ evaluates this Bayesian network only when the program acts on the distribution of `c`.



Programs 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. To solve this problem, $Uncertain\langle T\rangle$ defines the semantics of conditionals on estimates as *evidence* for a conclusion. Rather than asking "is speed faster than 60 mph?" $Uncertain\langle T\rangle$ asks "how much evidence is there that speed is faster than 60 mph?"

When the developer writes a comparison `if (Speed > 60)`, the program applies a lifted operator

$$> :: Uncertain\langle T\rangle \to Uncertain\langle T\rangle \to Uncertain\langle Bool\rangle$$

to create a Bernoulli distribution. The $Uncertain\langle T\rangle$ runtime computes a concrete boolean from this Bernoulli by using a hypothesis test, drawing only as many samples as necessary to decide whether $\Pr[Speed > 60] > 0.5$, computing if it is more likely than not that speed is faster than 60 mph.

To control false positives and negatives, developers may

specify a threshold for conditionals:

```
if ((Speed > 60).Pr(0.99)) ...
```

This conditional evaluates whether it is at least 99% likely that speed is faster than 60 mph. Higher thresholds reduce false positives at the expense of more false negatives; in this case, we are unlikely to falsely issue a ticket. The $Uncertain\langle T\rangle$ abstraction simplifies how developers control this trade-off, without involving them in the implementation of the statistics.

IV. Exploiting Context

Using a pedagogical example, Bushranger, this section establishes the need for context and inference when using estimates, introduces our language constructs, and describes their probabilistic semantics.

Example: Bushranger is an Xbox Kinect game that takes a photo of a player and then decides whether the player has beard. The game uses a `BeardRecognizer` API that indirectly exposes the output of a well trained machine learning model. Actually, in Kinect SDK, there are many such APIs defined to indirectly expose outputs of machine learning models, showing that the problem we are attacking is widespread and important.

We design a three-step recipe for developers to improve application accuracy by exploiting context:

1) Note what you *have*. *Expert* developers describe noisy estimated data as probability distributions. Compared to random values generated from data sources, probability distributions provide global information about the value set to generate and the likelihood of each value, which we use to exploit statistical expertise to improve accuracy of applications.
2) Encode what you *believe*. *Application* developers encode their understanding of application-specific domain knowledge—*context*, using our new conditional probability construct `<|`. The context describes strategies to decide whether an estimate is noise. Some context is *static*, i.e., general rules applicable to all estimates, such as "The possibility of a beard on females is 1%, and 29% for males". The context may combine static and *dynamic* knowledge. *Dynamic* knowledge describes facts or observations that customize static knowledge for specific application scenarios, such as "The player is female". With the combination of dynamic and static knowledge, our context is refined as "It is 99% unlikely this female player has a beard" and we can treat most "beard" estimates as noise for that player.
3) Compute what you *want*. Application developers specify how to compose estimates with context using our new Bayesian composition operator `#`. The specification describes how to calibrate incorrect estimates and how to enhance correct estimates. Under the hood, our runtime implementation performs Bayesian inference, to compose estimates and context as specified, and to sample estimates sufficiently and efficiently for more accurate computation. The application developer thus declaratively specifies simple inference problems, but the compiler and runtime automate the statistics, so application developers never explicitly codes the statistics, reason about Bayesian models, or choose sample sizes.

A. The Need for Context and Inference

A naive `BeardRecognizer` API implementation compares the probability output by a machine learning model against a predefined probability threshold, like 0.5. If the probability is greater than 0.5, the API returns true; otherwise it returns false. The implementation is problematic, because it assumes that the model can perfectly report a value above 0.5 for any bearded person. However, the model may perform poor for people for whom it is not well trained. As a result, the application will always make wrong judgments and frustrate those players.

Training better models may help reduce noise in estimates, and furthermore improve application accuracy. However, application developers usually do not have sufficient expertise to retrain the models themselves, and most do not have the massive computing resource required by the training task. On the other hand, although model developers can improve models, without knowing how the outputs of models get used and interpreted by applications, model builders have no idea how their model improvement affects application accuracy. Machine learning models are just one type of estimated data source; others include sensors, cloud computing, and human computing.

Given that noise in data sources is difficult to reduce, our approach requires the `BeardRecognizer` implementation to directly expose the probability calculated by its underlying model. In this way, developers obtain the flexibility to implement their own strategies of interpreting estimates for better beard recognition.

B. Exposing Estimates

Developers represent estimates by a data source as a probability distribution variable of type $Uncertain\langle T\rangle$. The variable encapsulates a *likelihood function*, $f : T \to \mathbb{R}$, which assigns a likelihood to each possible value of the estimates. For Bushranger, the function assigns a probability to *True* (that the user does have a beard) and to *False* (that the user does not).

C. Declaring Domain Knowledge

Developers use our *conditional probability* construct to define context—their domain-specific knowledge about what estimates are invalid or what is the distribution of estimate errors. The knowledge can be defined based on their personal experience, or specifications of data sources, like hardware sensors and machine learning models. Developers define *static* and *dynamic* context.

Static Knowledge: Static knowledge describes general principles to judge whether an estimate is noisy or reasonable that the developer knows and wants to add to the application to improve its accuracy. Given the same estimate, different applications may require different context to decide whether an estimate is valid. For example, in Bushranger, the developer may believe that 15% of their users have beards (perhaps due to demographic information). This knowledge is static: it refers to an *average* instance of the problem rather than to a concrete instance.

Developers may specify static knowledge as probability distributions directly, such as

```
var populationBeard = new Bernoulli(0.15);
```

In the language of Bayesian statistics, this *prior distribution* declares a belief about the value of a variable. While directly declaring static knowledge is sufficient for removing some noise, it is inaccessible to many developers, because it requires them to write down a marginal distribution directly, which they may not know.

Developers may prefer instead to build a distribution using variables they understand more intuitively. In machine learning,

```
Photo photo = Kinect.CapturePhoto();
Func<bool, double> BeardLikelihood
                        = Kinect.BeardRecognizer(photo);
Uncertain<Gender> genderPrior = Uniform(Male, Female);
Func<Gender, Bernoulli> Beard_Gender = (gender) => {
  new Bernoulli(gender == Male ? 0.29 : 0.01);
Bernoulli populationBeard = Beard_Gender <| genderPrior;
GenderPrior.Value = Male;

Bernoulli hasBeard = BeardLikelihood # populationBeard;
if (hasBeard.Pr(0.95))  AddBeardToAvatar();
```

Fig. 2. Bushranger using our new language constructs.

these variables are referred to as *hidden* variables. Developers use our *conditional probability* construct to declare hidden variables. In Bushranger, rather than declare that 15% of users have beards, it is more intuitive for developers to express a beard likelihood for each gender. The developer first defines a hidden variable for gender

```
Uncertain<Gender> genderPrior = Uniform(Male, Female);
```

This statement encodes that males and females are equally likely. Next the developer declares the relationship between this hidden variable and the likelihood the user has beard:

```
Func<Gender, Bernoulli> Beard_Gender = (gender) =>
  new Bernoulli(gender == Male ? 0.29 : 0.01);
```

This function says that the average male has a 29% chance to have a beard, while the average female has a 1% chance. Finally, the developer applies the conditional probability operator `<|` to define a `populationBeard` variable:

```
var populationBeard = Beard_Gender <| genderPrior;
```

Introducing the hidden variable makes this final distribution more intuitive. Note that this conditional probability operator is similar to the bind operation of the probability monad [10]. Below we show how hidden variables help exploit dynamic knowledge as well.

Dynamic Knowledge: Dynamic knowledge augments static knowledge to refine context by customizing the general static domain knowledge to a specific application scenario. This context applies to a particular input or instance of the problem. To define dynamic knowledge, developers must first encode their static knowledge and then fix a concrete value for one or more hidden values. In Bushranger, the hidden variable is the current user's gender, which can be recorded in the user's profile. If we know the user is male, we specify dynamic knowledge as follows:

```
genderPrior.Value = Male;
```

The resulting context is therefore refined to express a 29% chance to detect a beard for the user. In Bayesian inference terms, dynamic knowledge *observes* a value of the variable `genderPrior`, and changes the posterior distribution of `populationBeard` that depend on `genderPrior`.

A clear benefit of our abstraction is that the developer need not understand the details of (i) expressing a generative model, nor (ii) how to implement Bayesian inference over that model. The $Uncertain\langle T\rangle$ abstractions hide that complexity, as we detail in subsequent sections.

D. Composing Estimates and Context

With both an estimate and context in hand, developers define how to integration the two using our new ♯ *Bayesian composition* construct. While context delivers principles to judge whether estimates are noisy or not, integration delivers how to weaken erroneous estimates and enhance correct ones to improve the accuracy of applications.

The *Bayesian composition* construct implements *Bayes' theorem*, which describes how to combine a *prior* hypothesis with evidence to form a new *posterior* hypothesis. For example, Bushranger first retrieves an estimate from the Kinect API

```
Func<bool, double> BeardLikelihood =
                        Kinect.BeardRecognizer(photo);
```

and defines `populationBeard` as above. The developer next specifies the composition of the estimate with context:

```
Bernoulli hasBeard = BeardLikelihood # populationBeard;
```

This operation completes lazily. The runtime system only evaluates the distribution `hasBeard` when the program acts on the data in a conditional (e.g., `if (hasBeard.Pr(0.95))`) or performs an expected value operation. See Bornholt et al. for additional details on the semantics and implementation of conditionals and expected value without the new operators [3]. With and without the new operators, the runtime determines how many samples are necessary to correctly evaluate conditionals and expected value. Figure 2 shows Bushranger implemented using the additional language constructs. Though concise, under the hood this program performs Bayesian inference to improve the accuracy of the application. The developer does not implement the statistics, but instead expresses their intent declaratively. The compiler and runtime automate simple inference.

The net effect is that the variable `hasBeard` reweighs the different possible values based on context. For example, if `BeardLikelihood` indicates a 75% chance of a beard, but dynamic context indicates the user is female, `hasBeard` specifies a much lower probability of a beard. The results clearly depend both on estimate quality and context accuracy. If the estimate is perfect, nothing can make it better, and if the context is incorrect, it will not improve accuracy. Empirically in our case studies and other real-world applications, estimate quality is not perfect and available context improves accuracy, making our technique useful.

E. Semantics

We define a probabilistic semantics for our language constructs in terms of Bayesian networks. A Bayesian network is a directed acyclic graph in which nodes are random variables and edges are dependences between variables [1].

Static Knowledge: In the Bayesian network, static knowledge is a single node. For example,

```
var populationBeard = new Bernoulli(0.15);
```

indicates 15% of users have a beard. We introduce a single node

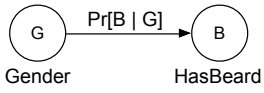
The *HasBeard* node is the *target* variable we wish to reason about. The static knowledge applies a prior distribution to this target variable. The conditional probability construct `<|` provides static context by introducing hidden variables. The developer defines a hidden variable as follows

```
Uncertain<Gender> genderPrior = Uniform(Male, Female);
```

and then uses it to define their knowledge of *HasBeard*

```
Func<Gender, Bernoulli> Beard_Gender = (gender) =>
  new Bernoulli(gender == Male ? 0.29 : 0.01);
var populationBeard = Beard_Gender <| genderPrior;
```

These operations create a Bayesian network with two nodes



The edge from *Gender* to *HasBeard* reflects that the latter *depends on* the former. The edge label $\Pr[B|G]$ that defines that relationship is the function `Beard_Gender`, which declares a distribution of *HasBeard* for each possible value of *Gender*. The conditional probability construct has type

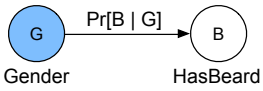
$$(T \to \mathit{Uncertain}\langle R\rangle) \to \mathit{Uncertain}\langle T\rangle \to \mathit{Uncertain}\langle R\rangle.$$

The operation `Q <| Z` says that given a distribution Z for the hidden random variable of type T, and a specification Q of how that hidden variable relates to a random variable O of type R, the result is a new distribution of type R that accounts for that relationship. The function Q maps each value z of Z to a distribution O_z of type R, so Q defines the conditional probability distribution $P[O|Z]$. The result of `Q <| Z` is the marginal distribution $\int \Pr[O|Z]\Pr[Z]\,\mathrm{d}Z$. In this way, `<|` is similar to the bind operation of the probability monad.

Dynamic Knowledge: When the developer provides dynamic knowledge that the user is male,

```
genderPrior.Value = Male;
```

the runtime updates the `genderPrior` variable and constrains its value. In the language of Bayesian networks, this operation *observes* the value of the variable, which we denote by shading the node:

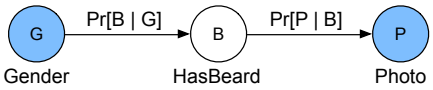


This example illustrates why dynamic knowledge is defined based on static knowledge: static knowledge reveals hidden variables which can influence the target variable, while dynamic knowledge specifies observations of those hidden variables to actually influence the target variable.

Bayesian Composition: When the developer composes context with the estimate,

```
Bernoulli hasBeard = BeardLikelihood # populationBeard;
```

our Bayesian composition operator ♯ creates a new graphical model that implements Bayes' theorem. The resulting Bayesian network adds an observed random variable that depends on the context:



For Bushranger we call this new node *Photo* because it reflects the *input* to the data source (i.e., the machine learning model which predicts whether the person in a photo contains a beard) not the output. The edge between *HasBeard* and *Photo* reflects the fact that the photo (and therefore the estimate) depends on the underlying state of whether the person has beard. In essence, whether the person *in the photo* has beard depends on whether the person has beard. Though this statement seems a tautology, it is the key to our technique.

The edge label $\Pr[P|B]$ relates to the probability density function returned by the data source (Section IV-B). Formally, that density function specifies $\Pr[B = b|P = p]$, the likelihood that the person in photo p does ($b = \mathit{True}$) or does not ($b = \mathit{False}$) have beard. This likelihood is the result of applying Bayes' theorem to B and P

$$\Pr[B = b|P = p] = \frac{\Pr[P = p|B = b]\Pr[B = b]}{\Pr[P = p]}$$

if we assume the estimator used uniform priors for $\Pr[B]$ and $\Pr[P]$. Exploiting context replaces the uniform prior for $\Pr[B]$ with a new prior $\Pr'[B]$, specified by the developer, to produce a new posterior distribution $\Pr'[P = p|B = b]$. By Bayes' theorem and using the uniform prior:

$$\begin{aligned}\Pr'[B = b|P = p] &= \frac{\Pr[P = p|B = b]\Pr'[B = b]}{\Pr[P = p]} \\ &= \frac{\Pr[B = b|P = p]\Pr[P = p]}{\Pr[B = b]}\frac{\Pr'[B = b]}{\Pr[P = p]} \\ &\propto \Pr[B = b|P = p]\Pr'[B = b]\end{aligned}$$

where the proportionality on the last line follows since we assumed $\Pr[B = b]$ was a uniform prior. If we write Q as the function $x \mapsto \Pr[B = x|P = p]$ and P the distribution $\Pr'[B = b]$, the Bayesian composition operator has a probabilistic semantics

$$\Pr[P \sharp Q = x] \propto Q(x)P(x) \tag{1}$$

which captures the composition of *probability distributions*, rather than multiplication of the random variables.

V. Implementation

We implemented our language constructs in C# and C++. We found that C#'s dynamic types simplified the implementation, but our language constructs are general enough to add to many other languages.

We overload arithmetic and other operators to construct a Bayesian network representation of computations, which the runtime evaluates lazily. Overloading means that from the developer's perspective, they are computing with values, not distributions. When the program uses an estimate in a branch condition or expected value statement (e.g., printing a value), the runtime evaluates the value through a visitor pattern to sample the Bayesian network by visiting each node in the graph and performing that node's `Sample` method. Unobserved nodes are sampled using ancestral sampling, which simply samples each parent of a node before sampling the node itself [1]. When the visitor encounters an observed node created by the Bayesian composition operator, it returns the observed value as the sample and evaluates the likelihood function (the P argument in $P\sharp Q$). The final weight of the sample is the product of all such likelihoods.

Existing sequential hypothesis tests do not support likelihood reweighting as a sampling technique, because each sample has an associated likelihood s_i. Samples are thus not "worth" the same – a sample with weight 0.01 is much less informative about the distribution than one with weight 0.9. We introduce a new *sequential likelihood reweighting* (SLR) algorithm, which we do not describe in detail here due to space constraints. SLR is a sequential hypothesis test that accounts for weighted samples by adapting Wald's sequential probability ratio test (SPRT) to deal with weighted samples.

VI. Case Study: Time of My Life

This section demonstrates with a case study that our abstraction is (i) concise—modest amounts of code implement powerful scenarios, (ii) efficient—the runtime takes only as many samples as required to make judgments, and (iii) accurate—adding context to estimates significantly improves application accuracy.

This case study improves *Time of My Life*, a popular Windows Phone application with 50,000+ active users. Time of My Life associates sequences of geolocations to semantic places, such as 'Home' or 'Work', and profiles where users

```
// current geographic location from GPS
Point point = GetPointFromGPS();
string[] labels = GetAllSemanticLabels();
// Application Agnostic Placer
Func<string, double> Likelihood = lbl => Placer(point, lbl);
// Application Specific Placer#
Func<Point, Uncertain<string>> PlacerSharp = point => {
  // Get closest semantic label as tuple
  LabeledPoint userDefinedPoint = FindClosestLabel(point);
  return new Uncertain<string>(() => {
    // if distance between semantic label and GPS point
    // is small, return the user's semantic label
    if (Distance(point, userDefinedPoint.point) <
        Gaussian(20,4)) // model error in user placement
      return userDefinedPoint.label;
    // GPS is far from semantic label; bias away from it
    return UniformlySelectFromSetExcept(labels,
      userDefinedPoint.label);
  }); };
Uncertain<Point> pointPrior = UniformPointPrior();
Uncertain<string> userPrior = PlacerSharp <| pointPrior;
pointPrior.SetValue = point; //constrain to current GPS
Uncertain<string> label = Likelihood # userPrior;
```

Fig. 3. Placer# implementation

spend time. A machine learning model creates *semantic-label estimates* from user GPS locations. This user-agnostic model is called Placer [7]. To improve these estimates, we add *user-defined priors*, where the user tags one or more locations with semantic labels. To exploit this context, we then combine semantic-label estimates with the user-label priors using the Bayesian composition operator. We call this version Placer#.

Improving Placer Estimates: The original Placer service does not incorporate information from an individual user in that user's predictions. Instead, the global Placer model combines labels, GPS, and data from public government surveys and maps. The deployed Placer model then takes a user's geographic location history, clusters them into nearby groups, and produces a multinomial likelihood over semantic labels. Figure 3 shows how Placer# creates a new likelihood function for Placer's estimates, and then composes those estimates with nearby user-defined labels to improve accuracy.

We believe user-defined labels are more likely correct than what Placer infers when a user's label is near the current location. The function `PlacerSharp` specifies context. Given a GPS point, Placer# (i) finds the closest user-labeled point to that location and (ii) if the distance between them is small (less than 20 m) Placer# assigns that label to the location, otherwise (iii) Placer# uniformly selects from its semantic labels. We assume Gaussian error in user label placements.

Placer# encodes static context by incorporating users' labels and their distance to GPS locations as a prior over labels. It uses dynamic context by *observing* a new GPS location, and combines this dynamic context with the machine learning model using the Bayesian composition operator.

Evaluation: We randomly select 10 active Time of My Life users, where each user has a minimum of 3 user-defined labels. Time of My Life collects a minimum of 2,000 GPS locations per user, but most have nearly 300,000. We split these GPS locations into training and test sets (90%/10%) and use the training set to build Placer's model as normal.

A user-defined 'Home' label has two effects. First, new GPS points *close* to the user label 'Home' are likely 'Home'. Second, new GPS points *far* from 'Home' are unlikely 'Home' (most users have one home) and so are *more* likely a different semantic label. Figure 4 shows the labels assigned by both Placer and Placer# to 1240 test-set GPS points for a single

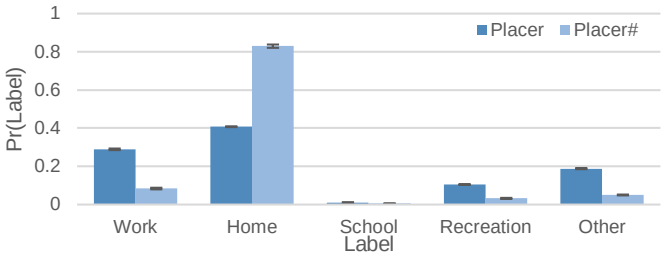


Fig. 4. Points near a user-label 'Home' are more than twice as likely to be correctly labeled by Placer#.

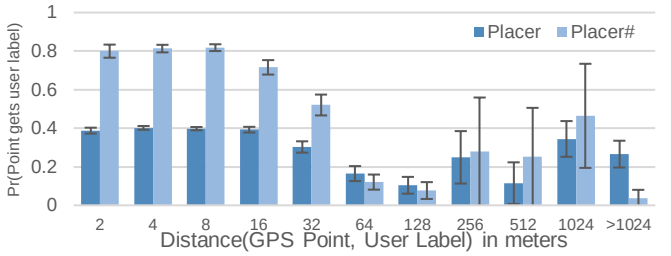


Fig. 5. The probability a GPS point is attributed to a user-label as a function of distance between the two.

user, all within 20 m of the user label 'Home'. The x-axis is the set of labels and the y-axis the average probability a point is given that label. The context that Placer# uses makes it much more accurate. Points close to the user label 'Home' are more than twice as likely to be labeled as such. All 10 users show qualitatively similar results.

Figure 5 demonstrates the generality of our results. We take all training GPS points from all 10 users. The x-axis is the distance of a GPS point from the nearest user label, and the y-axis is the likelihood the GPS point is attributed to that user label. Placer is very inconsistent; for example, it is less than 50% likely to label a GPS point as 'Home' even if that point is less than 2 m from a 'Home' label. In contrast, Placer# is much more consistent, with close points labeled appropriately over 80% of the time. For points far from the nearest label, the results in Figure 5 are less meaningful, because the selection of the "nearest" label is noisy.

VII. Context Examples

In previous sections of the paper, we described several examples of context: the *Time of My Life* case study, avatar enhancement, and GPS road snapping. A significant category of context is *personalization* for a specific user, which should enhance many IoT applications. For instance, the automated temperature setting for your house may both respond reactively to your movements at home (e.g., sleeping and cooking) and it can proactively and automatically anticipate that when you leave work, you always return home in about 30m on weekdays, but not on Thursdays when Poker is on your calendar, and pre-heat your house accordingly when you leave. In this case, context includes your calendar and the probability of you returning home, and the estimate is when you leave work (an estimate from the GPS on your phone). Dynamic context may be collected by combining observing your movements over time.

VIII. Conclusion

As computer hardware proliferated in the late 1970s and early 1980s, software that computed and reasoned about floating

point values was often incorrect, unreliable, and not portable. Codification of the IEEE Floating Point Standard in 1985 for hardware and software was wildly successful in delivering programmability, reliability, and portability of applications that reasoned and computed with floating point numbers.

We contend that computing is at a similar point in its history for probability distributions. Hardware and software are producing, computing, and reasoning about estimates without the appropriate specifications of error distributions and programming models. 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 more and more complex and yet depend on a growing community of non-expert developers for their success. Furthermore, the more programmable we make devices for non-experts, developers that lack statistical expertise, and even for non-programmers, the more likely innovation that exploits computing will proliferate. Previous work takes an ad-hoc approach to programming these devices that often ignores uncertainty and error in the name of convenience.

This paper identifies that the wide ranging use of estimates poses application correctness, programmer productivity, and optimization problems. We are the first to propose a sound, principled, and accessible programming language abstraction to address the increasing pervasiveness of uncertainty. In the spirit of hardware-software co-design, a key contribution of this paper is recognizing that the hardware problem of noise and non-determinism has a software solution. The $Uncertain\langle T\rangle$ abstraction balances the competing needs of experts and non-experts. $Uncertain\langle T\rangle$ gives expert developers a programming model for complex problems that compute with noisy data. This abstraction requires hardware and software experts define a richer API that exposes the uncertainty in their estimates. The developers who consume these APIs are protected from understanding or writing any statistics by the programming model, yet the resulting programs are statistically correct. The benefit to all developers is that they may ask more sophisticated questions of their data and correctly combine estimates from multiple sources. The $Uncertain\langle T\rangle$ abstraction is general enough to describe hardware and software estimates and to include in most programming languages. It therefore has the potential to improve programmability, correctness, and portability across a wide range of systems.

References

[1] Christopher M Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[2] Johannes Borgström, Andrew D. Gordon, Michael Greenberg, James Margetson, and Jurgen Van Gael. Measure transformer semantics for Bayesian machine learning. In *European Symposium on Programming (ESOP)*, pages 77–96, 2011.

[3] James Bornholt, Todd Mytkowicz, and Kathryn S. McKinley. Uncertain<T>: A First-Order Type for Uncertain Data. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems, ASPLOS 2014, Salt Lake City, UT, USA, March 1 - 4, 2014*. ACM, 2014.

[4] 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.

[5] Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: A language for generative models. In *Conference in Uncertainty in Artificial Intelligence (UAI)*, pages 220–229, 2008.

[6] Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. Probabilistic programming. In *Proceedings of the Future of Software Engineering, FOSE 2014, Hyderabad, India, May 31 - June 7, 2014*, 2014.

[7] John Krumm and Dany Rouhana. Placer: Semantic place labels from diary data. In *Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing, UbiComp 2013, Zurich, Switzerland, September 8 - 12, 2013*, 2013.

[8] Vikash K. Mansinghka, Daniel Selsam, and Yura N. Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *CoRR*, abs/1404.0099, 2014.

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

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