Abstractions and Techniques for Programming with Uncertain Data

James Bornholt

A thesis submitted in partial fulfilment of the degree of
Bachelor of Philosophy (Honours)
at The Australian National University

October 2013

Except where otherwise indicated, this thesis is my own original work.

James Bornholt
24 October 2013

To AJ.

Acknowledgments

First, I want to thank my supervisor, Steve Blackburn. Steve took me on as a first-year undergraduate who had no idea what this "research" thing was all about. That he did this without any obligation to do so speaks to the generosity has has showed me ever since. In the intervening years he has ignited my passion for this field, encouraging me towards new challenges and opportunities with his enthusiasm, and patiently guiding me with his knowledge and advice. Thank you for making my undergraduate career the learning experience it has been.

I've been incredibly lucky to work with Kathryn McKinley at Microsoft Research. I am constantly awed by Kathryn's vision, from the highest abstraction to the smallest technical detail. Meetings with her usually leave me exhausted, as she takes my single new thought for the week, supplements it (on the spot) with three of her own, and links them all to five areas I'd never even considered. Her knowledge, experience, and kindness have made me a better researcher. Thank you for taking me on, particularly on the exciting project that is the topic of this thesis.

Also at Microsoft I want to thank Todd Mytkowicz. I've enjoyed collaborating with Todd on every aspect of our work, small and large. Often this collaboration resulted in hours-long conversations, working through my vague questions and objections with no more detail from me than "I don't think that's right". By the end of these conversations I usually agree with him, a testament not only to his expertise but to his patience!

Steve has very kindly treated me as part of his lab, and so I have had the privilege of working with his excellent graduate students. Thank you for engaging conversations, interesting work, and showing that garbage collection can be fun! Similarly, thank you to my friends outside the lab, who have helped keep me on track, allowed me to vent, and generally made this experience even more enjoyable.

Finally, thank you to my family: Karen, Mark, and Michael. Your belief in me has opened so many doors in my life, and driven me to keep working harder. Even if you don't understand it all yet (I will try!), I hope you are as proud of this work as I am.

This thesis began as an internship project supervised by Kathryn McKinley and Todd Mytkowicz at Microsoft Research, described in a conference paper currently under submission [Bornholt et al., 2013]. Except for the case study in Section 6.2, which Todd conducted, this thesis is my own original work.

Abstract

Rapid growth in the power and ubiquity of computing devices has shifted the focus of programmers towards problems that involve uncertainty. Whether this uncertainty is required to wrangle the sheer scale of big data, or inherent in a digital sensor, or part of a probabilistic model, or a side effect of approximating calculations to save energy and increase performance, programmers reason about and address this uncertainty using programming languages.

Unfortunately, existing programming languages use simple discrete types (floats, integers, and booleans) to represent this uncertainty, and encourage programmers to pretend that their data is not probabilistic. The illusion of certainty causes applications to experience three types of *uncertainty bugs*: (1) random errors by treating estimates as facts; (2) compounded error through computation; and (3) false positives and negatives by asking the wrong questions.

Existing approaches to uncertain data centre on probabilistic programming, providing specialised languages for expert programmers to build and reason about probabilistic models. These tools are inappropriate for non-expert programmers, on the front lines of the increasing variety of problems involving uncertainty, because these languages require a strong understanding of probability theory.

My thesis is that by making uncertain data a primitive first-order type, non-expert programmers will write programs that are more concise, expressive, and correct.

This thesis introduces the *uncertain type*, or $Uncertain\langle T\rangle$, a programming language abstraction for uncertain data. The uncertain type encapsulates probability distributions, similar to other probabilistic programming abstractions. But unlike prior work, the uncertain type provides an accessible interface for non-expert programmers to reason correctly about uncertainty. Probabilistic programming is too blunt a tool for the concrete problems that non-expert programmers face in applications, and pays a performance and accessibility cost for its generality.

I describe the interface that the uncertain type provides through a four step process for programming with uncertainty: (1) domain experts identify distributions; (2) programmers compute with uncertain data; (3) programmers ask questions with conditionals; and (4) domain experts and programmers improve estimates with domain knowledge. The uncertain type's semantics incorporate sampling and hypothesis testing to make the abstraction tractable and practical in performance. Three case studies show how programmers use the uncertain type to produce programs that are more correct, without significant programmer effort or performance overhead.

This thesis argues that principled mechanisms for reasoning under uncertainty are increasingly necessary in the modern computing landscape, and demonstrates how to make these mechanisms accessible, flexible, and efficient.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis explores *uncertainty*, why programmers must consider it, and programming models for doing so that are accessible, efficient, and expressive. Programs are increasingly working with uncertain data, and my thesis is that good programming model abstractions for uncertain data will make programs more concise, expressive, and correct.

1.1 Problem Statement

Modern applications increasingly face the challenge of computing under uncertainty. This uncertainty manifests itself in the data that programs manipulate. In some cases, the uncertainty is due to the limited resolution or accuracy of a sensor. Hardware or software might deliberately introduce uncertainty by choosing to approximate computations, to save energy or to improve performance. The sheer scale of big data might require a program to reason about a smaller subset, creating uncertainty about the true result. Machine learning may produce predictions, learned from a set of training data, that are uncertain. Or the uncertainty might be an inherent part of a probabilistic model, created to reason about problems that are random and yet structured.

What these processes have in common is that they produce estimates of underlying values. For instance, a machine learning algorithm estimates the value of an unknown function, and a sensor estimates the true value of a physical quantity. The difference between an estimate and a true value is uncertainty.

When we write programs that consume estimates, we should recognise their uncertainty, and how it affects the program's behaviour. But most programming languages do not distinguish between estimates and true 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 estimates as certainties, which is problematic.

This problem is not abstract. I demonstrate three types of *uncertainty bugs* that applications experience as a result of the illusion of certainty. Programs experience random errors by treating estimates as facts, leading to absurd outputs. Computation on estimated data compounds the error, creating greater inaccuracy. And misusing simple types in conditionals creates false positives and negatives by asking simple questions of probabilistic data.

Faced with the challenge of uncertainty, most programmers choose to ignore it. I survey 100 popular smartphone applications and find only one application reasons about the error in GPS estimates. But this choice to ignore uncertainty is not a conscious one. I argue that this apathy towards correctness stems from the absence of the right abstraction for uncertain data in everyday programs. The right abstraction needs to be accessible to non-expert programmers, efficient enough for use in applications, and expressive enough to reason about uncertainty. Current abstractions fall short in one or more of these areas:

> **Accessibility.** Probabilistic programming languages are a booming field of research. These languages use stochastic primitives to encode generative models, and apply inference algorithms to reason about specific problems using these models. The encoding is based on the monadic structure of probability distributions [Ramsey and Pfeffer, 2002]. Notable examples include BUGS [Gilks et al., 1994], Church [Goodman et al., 2008], and Fun/Infer.NET [Borgström et al., 2011; Minka et al., 2012].
>
> To use these languages effectively requires expertise in statistics and probabilistic modelling. While experts can potentially use these languages to reason about uncertain data, a non-expert does not have the knowledge to phrase their problems in this way. These languages are not sufficiently accessible for everyday programming.
>
> **Efficiency.** Probabilistic programming languages use *inference* algorithms to reason about problems. The output of an inference algorithm is a posterior distribution, but to compute the entire posterior distribution requires exploring all paths through the probabilistic program. In all but the simplest cases this cannot be done analytically and so must be done by sampling, repeatedly executing the program, but some paths through the program are very unlikely to be sampled, causing significant performance degradation.
>
> Although many algorithms improve the efficiency of sampling (e.g. Metropolis-Hastings [Hastings, 1970], Quick Inference [Chaganty et al., 2013]), inference is still too slow for commodity use in everyday applications.
>
> **Expressivity.** Current abstractions for uncertainty are often simplistic, to overcome the poor efficiency or accessibility of richer techniques. For example, smartphones use a simple abstraction for geolocation data, representing uncertainty as a simple radius around a point. This abstraction is accessible and efficient. But uncertainty engenders new questions about data that such simple abstractions cannot hope to express. Reasoning under uncertainty requires programmers to consider confidence in the available evidence. For example,

> rather than asking “is my speed greater than 10 km/h?”, they should ask “how confident am I that my speed is greater than 10 km/h?”.
>
> This reasoning can be done manually, but requires significant domain expertise to implement. Current simple abstractions are not sufficiently expressive to ask the questions that uncertainty requires programmers to consider.

The challenge is: can an abstraction for uncertainty be suitably accessible, efficient, and expressive as to be practical for everyday programmers?

1.2 Contributions

I contend that the reason for the shortcomings of current abstractions is a failure to recognise the way modern software is developed. Modern applications consume uncertain data not directly from the source, but through a stack of abstractions programmers engage with through an application programming interface (API). It is not reasonable to expect everyday programmers to engage with uncertain data by constructing an entire probabilistic model from first principles; to expect so is to ignore the value of abstraction.

Current APIs for uncertain data hide its probabilistic nature, in the mistaken belief that it is an implementation detail to abstract away. My central argument is that the probabilistic nature of uncertain data cannot stop at the API boundary. Such an artificial restriction makes correct reasoning under uncertainty much more difficult and, in practice, causes programmers to ignore uncertainty completely, resulting in poorer quality software.

This thesis introduces the *uncertain type*, or *Uncertain*$\langle T \rangle$, a programming language abstraction for uncertain data. The uncertain type encapsulates probability distributions, similar to other probabilistic programming abstractions. But unlike prior work on probabilistic programming, the uncertain type focuses on providing an accessible interface for non-expert programmers to reason correctly about uncertainty. For everyday programmers, the uncertain type is accessible, efficient, and expressive.

Probabilistic programming languages are useful for abstract problems involving uncertainty, but the uncertain type is concerned with specific instances of a problem. Whereas probabilistic programming can consider the distribution of variables in isolation by marginalising, the uncertain type considers only conditional distributions. Because in a concrete instance of a problem, all variables have a value, the conditional distribution is appropriate. Probabilistic programming is too blunt a tool for concrete problems, and pays a performance and complexity price for its generality.

The key insight of the uncertain type is that considering accessibility as a first order requirement leads to a restricted subset of probabilistic operators, that are suitably expressive to solve real-world problems, but lend themselves to a very efficient implementation through sampling and hypothesis testing. Rather than eagerly producing unnecessary precision through existing inference algorithms, the uncertain type samples distributions only to the level of precision necessary to evaluate individual queries. The accessibility and efficiency requirements work in concert to strike a natural balance that remains suitably expressive.

I describe the interface the uncertain type provides by presenting a four step process for programming with uncertainty:

> **Identifying the distribution** of uncertain data requires domain expertise since the distribution depends on the particular data source. Domain experts often already know the distribution of their data, and only lack an abstraction to expose this information to programmers.
>
> **Computing with distributions** includes using arithmetic operations, converting units, and combining with other distributions. The uncertain type uses operator overloading to make this step mostly transparent to programmers, and lazy evaluation to make it efficient and tractable.
>
> **Asking the right questions** of distributions requires new semantics for conditional expressions on probabilities rather than binary decisions on deterministic types. The uncertain type defines two types of conditional operations, and implements both using hypothesis testing to balance performance and accuracy.
>
> **Improving estimates with domain knowledge** combines pieces of probabilistic evidence. Programmers specify application-specific information through a simple interface, and libraries incorporate this domain knowledge using Bayesian inference to improve the quality of the estimates they produce.

I show how the uncertain type implements this interface efficiently, by approximating distributions with Monte Carlo sampling, lazy evaluation for computations, and hypothesis testing for dynamically determining the ideal sample size for conditionals.

Finally, I reinforce these contributions with case studies. I show how the uncertain type improves the accuracy and expressiveness of programs that use GPS, one of the most widely used hardware sensors. I present an example of how the uncertain type can incorporate domain knowledge to virtually eliminate noise in a simple digital sensor. And I show how the uncertain type is a promising programming model for approximate computing, a growing research field that trades accuracy for performance and energy efficiency. Together, these case studies demonstrate that the uncertain type allows non-expert programmers to write programs that are more concise, expressive, and correct.

1.3 Thesis Outline

Chapter 2 provides an overview of probability theory and probabilistic programming, and a survey of relevant literature around programming with uncertainty. Chapter 3 motivates the problem this thesis addresses by exploring uncertainty bugs in a simple GPS application.

Chapter 4 presents *Uncertain*$\langle T \rangle$, describes the interface it provides programmers, and develops a four step process for programming with uncertain data. Chapter 5 shows the implementation insights of *Uncertain*$\langle T \rangle$ that allow it to be efficient and tractable. Chapter 6 presents three case studies that demonstrate the effectiveness of *Uncertain*$\langle T \rangle$ against its goals of accessibility, efficiency, and expressiveness.

Finally, Chapter 7 concludes the thesis, identifiying future work, and describing how my contributions show that a primitive first-order type for uncertain data enables programs that are more concise, expressive, and correct.

Chapter 2

Background and Related Work

Uncertainty is a well known problem in many domains. The mathematics of probabilities gives scientists the tools to reason about uncertainty and make decisions in the face of imperfect information. Extensive existing work discusses approaches to uncertainty and probability in programming languages, and in domain-specific problems. This chapter provides an overview of probability theory and probabilistic programming, and surveys relevant literature on programming with uncertainty.

Section 2.1 briefly introduces relevant probability theory. Section 2.2 discusses probabilistic programming, a field of research into using programming languages to address uncertainty. Section 2.3 demonstrates some domain-specific solutions to the problem of uncertainty, and Section 2.4 examines other techniques for addressing uncertainty. Section 2.5 discusses emerging trends that further motivate tools designed to address uncertainty. Finally, Section 2.6 summarises the chapter, emphasising the need to improve upon existing programming models for uncertainty.

2.1 Probability theory

This section introduces the necessary mathematical background to support the rest of the thesis.

2.1.1 Random variables and probability distributions

Probability theory rests on the concept of a *random variable*. In deterministic mathematics, a variable takes on a single value; for instance, the equation $x = 5$ means “the value of x is 5”. Probability introduces the idea of a random variable, whose value varies due to uncertainty. For example, the outcome of a coin flip is a random variable. A fair coin has a 50% probability of being heads and a 50% probability of being tails. Of course, a particular flip has one definite value, either heads or tails. The random variable is a conceptual model for the outcome of a flip, and allows us to reason probabilistically about coin flips in general. For example, we can reason about the probability of seeing three heads in a row using the random variable.

A random variable is defined over a *probability space* Ω, which is the domain of the variable. For a coin, this domain is discrete: $\Omega = \{\text{Heads}, \text{Tails}\}$. Probability spaces can also be countably or uncountably infinite, representing random variables over domains such as the set of real numbers. We call such infinite domains *continuous* (versus discrete). We say a random variable with probability space Ω is a random variable *over* Ω.

The probability that a random variable over Ω takes a value $x \in \Omega$ is represented by a *probability distribution*, a function $p : \Omega \to [0, \infty)$. For discrete Ω, the value $p(x)$ is exactly the probability that the random variable takes the value x (and so in fact $p(x) \in [0, 1]$, and $\sum_{x \in \Omega} p(x) = 1$). But for continuous Ω, the probability that any one value x is observed is zero,[1] since the sum rule of probability says that the sum of the probabilities of every $x \in \Omega$ is 1, and Ω is infinite. In the continuous case we refer to p as a *probability density function*; the value $p(x)$ represents the probability that the random variable takes a value somewhere in the interval $[x, x + dx]$ for an infinitesimal dx. In the continuous case, the sum rule of probability dictates that $\int_\Omega p(x)\,\mathrm{d}x = 1$.

We say two random variables are *independent* if the value of one has no bearing on the value of the other. For example, two coin flips are independent of each other: the value of the first coin flip has no effect on the value of the second. Formally, we say two random variables A and B are independent if, for every possible $a \in \Omega_A$ and $b \in \Omega_B$,

$$\Pr[A = a \text{ and } B = b] = \Pr[A = a] \Pr[B = b].$$

Independence greatly simplifies a number of probability operations.

The *expected value* $\mathbb{E}_A[f]$ of a function f under a random variable A (with probability density function $p(x)$) is a weighted average, defined as

$$\mathbb{E}_A[f] = \int_\Omega f(x) p(x)\,\mathrm{d}x.$$

Most often we take f to be the identity function, in which case we write $\mathbb{E}_A[f]$ as simply $\mathbb{E}[A]$, and say that this is the expected value of A. We also call $\mathbb{E}[A]$ the *mean* or *average* of A.

2.1.2 Families of probability distributions

Two families of probability distributions are particularly relevant to this work.

The normal (or Gaussian) distribution is perhaps the most common probability distribution. The normal distribution is a continuous distribution defined over the real numbers by the probability density function

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

[1] We could have a random variable with continuous domain but whose set of actual outcomes is finite (for example, $\Omega = \mathbb{R}$ but the variable is always either 1.3 or 4.2). For clarity when we say a random variable is continous, we mean it has infinite *support*, i.e., an infinite set of actual outcomes.

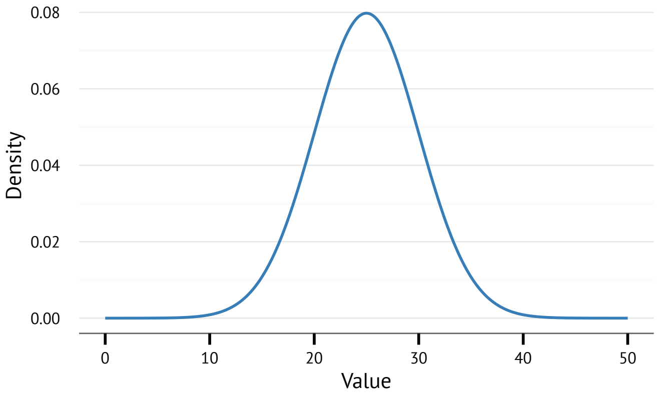


Figure 2.1: A Gaussian distribution with mean $\mu = 25$ and variance $\sigma^2 = 25$

The distribution has two parameters: μ, the mean of the distribution, and σ^2, the variance of the distribution. We call μ the mean because it is also the expected value of the distribution. Figure 2.1 shows a Gaussian distribution.

Normal distributions occur frequently when modelling real-world problems. For example, the Central Limit Theorem tells us that the mean of a large number of independent random variables is approximately normally distributed. This theorem holds regardless of the distribution of the individual random variables being averaged, which allows us to reason about the distribution of the mean of a set of variables even if we know nothing about the distribution of the variables themselves.

The Bernoulli distribution is the most common example of a discrete probability distribution. A Bernoulli distribution has a probability space of two elements, usually denoted either as $\{\text{False}, \text{True}\}$ or as $\{0, 1\}$. The distribution has a single parameter μ which describes the probability that the variable takes value 1. The Bernoulli distribution therefore has probability density function

$$f(x;\mu) = \begin{cases} 1-\mu & ,x=0 \\ \mu & ,x=1 \end{cases}$$

or equivalently

$$f(x;\mu) = \mu^x(1-\mu)^{1-x}.$$

A Bernoulli is simply a formalisation of a biased coin; the parameter μ describes the probability the coin comes up heads. An unbiased coin is therefore a Bernoulli random variable with $\mu = 0.5$.

2.2 Probabilistic programming

Problems involving uncertainty are becoming more prominent of late in many fields of computer science. To solve these problems, we turn to the mathematics of probabilities. *Probabilistic programming* reasons about probabilities by introducing random variables into the syntax and semantics of programming languages, allowing programmers to model and solve problems involving uncertainty.

Formally, the goal of probabilistic programming is to represent *generative models* of problems. A generative model is a full probabilistic model of all the variables in a problem, both inputs and outputs. That is, a generative model describes the joint distribution $f(x_1, \ldots, x_n)$, which is the complete relationship between all variables. In contrast, a *discriminative model* only captures distributions for the target variables, conditioned on the observed variables; that is, the conditional distribution $f(x_1, \ldots, x_i \mid x_{i+1}, \ldots, x_n)$. A discriminative model does not model the distribution of the input variables. The generative model therefore provides a more complete picture of the problem, at the expense of a larger model. Generative models are so named because they allow sampling (i.e., generation) of new synthetic data from the model.

2.2.1 The probability monad

The founding idea of probabilistic programming is that probability distributions form a monad [Giry, 1982]. Much of the work on probabilistic programming exploits this structure to build functional probabilistic programming languages that allow programmers to represent distributions [Erwig and Kollmansberger, 2006; Park et al., 2005; Ramsey and Pfeffer, 2002].

A generative model is a joint distribution $f(x_1, \ldots, x_n)$. To construct a representation of this distribution, we start with distributions $f(x_1), \ldots, f(x_n)$ for each variable, and some information about how they relate. This section builds on a Haskell example [Erwig and Kollmansberger, 2006].

We will represent all probabilities as floats:

```
newtype Probability = Prob Float
```

The simple formulation of the probability monad handles only discrete distributions, representing them as a map from the probability space Ω of possible outcomes to the probability of each outcome. So a distribution of type a is simply a list unD of pairs (x, p) where x is in a and p is a probability:

```
newtype Dist a = Dist {unD :: [(a, Probability)]}
```

We can trivially define basic distributions from this definition; for example, the uniform distribution is:

```
uniform :: [A] -> Dist A
uniform xs = Dist [(x, 1/total) | x <- xs]
    where total = fromIntegral (length xs)
```

That is, each value x in the input domain xs has probability $1/n$, where n is the number of elements in xs. Also note that we can create a simple function for distributions with guaranteed outcomes:

```
certainly :: A -> Dist A
certainly x = Dist [(x, 1)]
```

We can build simple distributions for the $f(x_i)$s using these definitions. There are two ways to combine these distributions to form a joint distribution over all the variables. If the variables A and B are independent, the combination is trivial: we simply form each possible pair (x, y) of values from A and B, respectively, and assign to each pair the probability pq, where p is the probability that $A = x$ and q the probability that $B = y$. In code:

```
prod :: Dist A -> Dist B -> Dist (A,B)
prod (Dist a) (Dist b) = Dist [((x,y), p*q) | (x,p) <- a, (y,q) <- b]
```

A more interesting case is when the variables are not independent. If B depends on A, then for each value of B there is a different distribution of A, the function $f(a \mid B = b)$. Visually, the graph $f(a, b)$ is three-dimensional, and the function $f(a \mid B = b)$ is a slice of that graph along the plane where $B = b$. In code, this means we need a function of type `B -> Dist A`, that tells us for each possible value b of B the distribution $f(a \mid B = b)$ of the corresponding A:

```
prod' :: Dist B -> (B -> Dist A) -> Dist A
prod' (Dist b) f = Dist [(y, q*p) | (x,p) <- b, (y,q) <- unD (f x)]
```

Notice that `prod' b f` is actually the *marginal* distribution over A, rather than the joint distribution of A and B. To get the joint distribution we simply change f to return a distribution over (A, B) instead of just over A (i.e., f now has type `B -> Dist (A,B)`), so that $f(x)$ is a distribution over (A, B) but the second element of each pair is always the input x.

But now we have defined everything we need for a monad. A monad is simply a type with a unit operation and a bind operation:

```
return :: A -> Monad A
(>>=)  :: Monad A -> (A -> Monad B) -> Monad B
```

These operations must satisfy some simple laws which are not relevant to this discussion. In our case, the return operation is `certainly` and the bind operation `prod'`:

```
instance Monad Dist where
    return x       = Dist [(x,1)]
    (Dist d) >>= f = Dist [(y, q*p) | (x,p) <- a, (y,q) <- unD (f x)]
```

This elegant monadic structure allows the simple definition of a number of probabilistic models; common examples include the Monty Hall problem [Erwig and Kollmansberger, 2006] and applications of Bayes' rule [Kidd, 2007]. But as always, low-level details like the probability monad's representation of distributions can benefit from the abstractions afforded by higher level languages.

2.2.2 Probabilistic programming languages

A probabilistic programming language abstracts the low-level details of representing probabilities and provides a higher-level interface for programmers to define probabilistic models.

```
earthquake = Bernoulli(0.0001)
burglary   = Bernoulli(0.001)
alarm      = earthquake or burglary

if (earthquake)
  phoneWorking = Bernoulli(0.7)
else
  phoneWorking = Bernoulli(0.99)

# Inference
observe(alarm)
query(phoneWorking)
```

(a) A probabilistic program in psuedocode

```
(mh-query
  500 100       ; number of samples and latency between them

  (define earthquake   (flip 0.0001))
  (define burglary     (flip 0.001))
  (define alarm        (or earthquake burglary))
  (define phoneWorking (if earthquake (flip 0.7) (flip 0.99)))

  phoneWorking  ; target variable
  alarm         ; observed variables
)
```

(b) The same probabilistic program in Church [Goodman et al., 2008]

```
Variable<bool> earthquake = Variable.Bernoulli(0.0001);
Variable<bool> burglary   = Variable.Bernoulli(0.001);
Variable<bool> alarm      = (earthquake | burglary);

Variable<bool> phoneWorking = Variable.New<bool>();
using (Variable.If(earthquake)) {
  phoneWorking.SetTo(Variable.Bernoulli(0.7));
}
using (Variable.IfNot(earthquake)) {
  phoneWorking.SetTo(Variable.Bernoulli(0.99));
}

// Inference
InferenceEngine ie = new InferenceEngine();
alarm.ObservedValue = true;
ie.Infer(phoneWorking);
```

(c) The same probabilistic program in C# using Infer.NET [Minka et al., 2012]

Figure 2.2: A probabilistic model captured as a probabilistic program in three different languages. This example problem is used by Chaganty et al. to demonstrate Quick Inference [Chaganty et al., 2013].

Figure 2.2 shows a probabilistic model defined in three different probabilistic programming languages. Figure 2.2(a) defines the model using an imperative psuedocode language. The simple program is based on an example used by Chaganty et al. to demonstrate their work on inference [Chaganty et al., 2013]. The program models a world with four variables: `earthquake`, `burglary`, `alarm`, and `phoneWorking`. There is a 0.01% probability of an earthquake occurring, and a 0.1% probability of a burglary. The alarm goes off if either an earthquake or a burglary occurs (or both). But only earthquakes affect the phone – if an earthquake occurs, the phone will work with 70% probability, otherwise it will work with 99% probability. The next section discusses the significance of the inference statements on lines 10 to 12.

There are two schools of thought as to the design of probabilistic programming languages. The first suggests they should be novel languages, free of the restraints of existing syntax and semantics. This freedom allows these languages to define new primitives and syntax for probabilistic models. Most of these novel languages build on the idea of a stochastic lambda calculus, which adds a notion of probabilistic execution to the lambda calculus. The exact formulation varies; some include probabilistic sampling [Park et al., 2005], others use a primitive for probabilistic choice [Saheb-Djahromi, 1978], and still others directly use the distributions of inputs.

Figure 2.2(b) shows the example probabilistic model implemented in Church [Goodman et al., 2008]. Church is a novel language defined as a subset of Scheme plus evaluation and query primitives used to define generative models. The example code uses the Metropolis-Hastings inference algorithm discussed below to reason about the problem. This program demonstrates the benefits of defining a new language: the code succinctly expresses the model using primitives that are easy to understand. BUGS is another commonly used novel language for probabilistic programming, particularly for inference on graphical models [Gilks et al., 1994]. These languages are popular among experts, but have little likelihood of adoption by non-expert programmers because they are complex and difficult to interface with existing programs.

The second school of thought suggests instead that stochastic functions be embedded inside an existing programming language. This leads to a more complex interface since it is difficult to define new syntax in an existing language. But embedding in popular existing languages democratises access to probabilistic reasoning, since it is easy for programmers to adopt in the programs they are already writing.

Figure 2.2(c) shows the example probabilistic model implemented in C# using the Infer.NET library [Minka et al., 2012]. Infer.NET is a library for Bayesian inference that can be embedded into a general purpose programming language like C#. Programmers use Infer.NET to build up an object graph representation of their problem domain, and then use inference algorithms on that model. Rather than adding new stochastic primitives to the language syntax, Infer.NET provides library functions to define random variables in terms of distributions and conditional probabilities. This results in more verbose code, but allows programmers to access probabilistic reasoning from the languages they are already using for their applications. Other embedded probabilistic languages include a domain-specific language embedded in Objective CAML [Kiselyov and Shan, 2009], and various libraries that provide functions for manipulating probability distributions explicitly [Glen et al., 2001; Jaroszewicz and Korzeń, 2012]. These packages vary in the depth of their embedding; some (like Infer.NET) use operator overloading to allow programmers to manipulate random variables like discrete variables, while others ask programmers to explicitly operate on the underlying distributions.

2.2.3 Inference on generative models

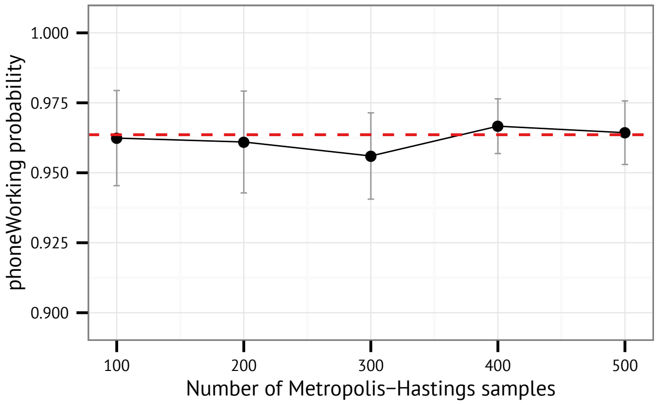
While there are interesting questions about the expressiveness of probabilistic programming languages, the primary motivation for modelling problems in this way is to use the models to perform *inference*. Inference algorithms calculate the posterior distribution of variables under some conditional restriction. In the examples in Figure 2.2, the query posed to the inference algorithm is to calculate the distribution of `phoneWorking`, given the observation that `alarm` is true. Informally, this query asks what the probability is that the phone is working if the alarm has gone off.

Notice that nowhere in the model is this distribution explicitly defined: it must be *inferred* from the relationship between the variables as defined by the model. In particular, the code does define the relationship between `phoneWorking` and `earthquake`, and between `earthquake` and `alarm`. To answer the query, the inference algorithm will infer the probability that an earthquake occurred given that the alarm went off, and then infer from that the probability that the phone is working.

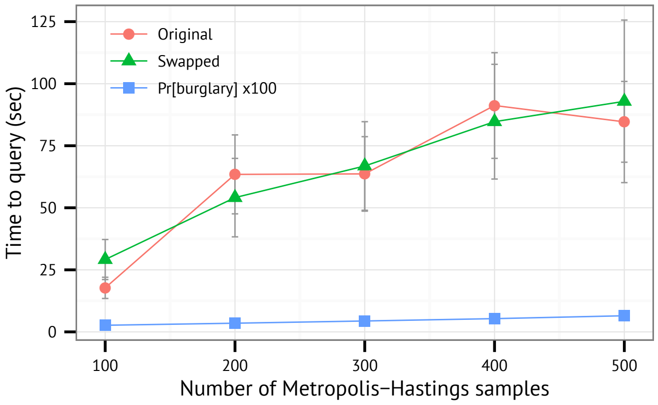
The output of an inference query is a distribution, not a single value, known as the *posterior* distribution because it is the distribution of a variable *after* an observation. In the example in Figure 2.2(a), the output of `query(phoneWorking)` is a Bernoulli distribution – a probability that the phone is working. Although in this example the distribution is simple, in real-world models the distribution will likely have a much larger probability space, and therefore be much more complex to infer.

The logical approach is to infer the answer analytically. For a simple problem like this one, direct analysis is tractable and efficient, and it turns out that the correct answer is approximately a 96.36% probability that the phone works after an alarm. But direct analysis breaks down for even slightly more complex models; in particular, a direct analysis is exponential in the number of random choices being made.

We therefore turn to some form of sampling for efficient inference. A key insight of probabilistic programming is that the program actually defines a joint prior distribution over all the variables, and the program can be executed to generate samples from this distribution. The execution proceeds by drawing a sample from each terminal variable (in this case, the `Bernoulli` instances), and propagating those samples through the program. In the example in Figure 2.2(a), since each Bernoulli variable returns a boolean, the conditional on line 5 executes in the usual way. At the end of the execution, the assignments to each variable together form a sample of the joint distribution defined by the program.



(a) Result of the inference algorithm. The dashed line is the true probability of `phoneWorking` (0.9636).



(b) Performance of the inference algorithm for different probabilities of the variables.

Figure 2.3: Performance of the Church inference algorithm on the simple probabilistic program in Figure 2.2. Error bars are 95% confidence intervals of 50 executions.

Sampling leads us to the key shortcoming of probabilistic programming: its inefficiency. The sampling routine must explore all possible paths through the program to understand its behaviour, and after computing a particular sample, must then apply the conditional observations of the inference query. The issue is that some paths through the program are very unlikely; in Figure 2.2(a), the branch on line 5 is taken with probability 0.01%. To be sure of executing this branch, the inference algorithm must draw many samples. Moreover, since the query is conditioned on `alarm` being true, which occurs with probability approximately 0.01%, many of these samples will have `alarm` set to **`false`** and therefore must be rejected. Only very few samples are able to be accepted, and we need many of these accepted samples to adequately explore the unlikely paths of execution.

Current research focuses on how to make these queries more efficient by reasoning symbolically about the model [Chaganty et al., 2013]. But even efficient algorithms cannot overcome the inherent expense of this approach. Figure 2.3 shows the results of using Church's Metropolis-Hastings inference algorithm on the program in Figure 2.2(b). In both graphs, the x-axis shows the number of samples drawn using the inference algorithm. Figure 2.3(a) shows the output of the inference algorithm (the probability of `phoneWorking` given the alarm went off) at each sample size. As expected, larger sample sizes give smaller variation in the result, and are more accurate. But the tradeoff for this improved accuracy is performance: Figure 2.3(b) shows that drawing 100 samples from the original model (in Figure 2.2(b)) takes 18 seconds.

The poor performance of inference algorithms on even simple examples has two causes. First, even though smart algorithms like Metropolis-Hastings try to improve the performance of inference, the fundamental problems of exploring unlikely execution paths and achieving practical rejection rates remain. To demonstrate these effects, Figure 2.3(b) also shows results for two modified versions of the program in Figure 2.2(b). The first modification swaps the probabilities of `burglary` and `earthquake`. This holds the probability of `alarm` constant while increasing the probability of `earthquake`. This modification does not significantly affect the performance of the inference algorithm. The second modification increases the probability of `burglary` by a factor of 100. Increasing this probability increases the probability of `alarm` being true and therefore decreases the rejection rate of the sampler, resulting in significantly improved performance. Since the probability of `earthquake` is unchanged in this second modification, these results together show that the primary cause of poor performance on this model is the rejection rate of the sampler.

Second, the probability monad's primitive nature of distribution encoding means even simple distributions have complex representations. For example, any distribution over a finite discrete domain can be encoded using only flips of a fair coin, but the encoding is necessarily complex and therefore slow. Some work on static analysis tries to improve on this complexity by reasoning symbolically about the structure of the representation [Chaganty et al., 2013; Sankaranarayanan et al., 2013], but to make substantial improvement requires a different representation for distributions.

2.2.4 Representing distributions

One alternative is to represent distributions as *sampling functions*. A sampling function returns a new random sample from a distribution each time it is invoked. This representation necessarily sacrifices accuracy compared to precise representations – to calculate an exact distribution from the sampling function requires infinite samples. But sampling functions have substantial performance benefits compared to other representations. The sampling function naturally focuses its attention on the most significant regions of the distribution, and so does not expend resources exploring areas that contribute little weight to the total probability. Sampling functions also simplify the representation of continuous distributions compared to the usual implementation of the probability monad.

$\lambda_{\bigcirc}$ is a probabilistic programming language built on sampling functions [Park et al., 2005]. Park et al. show how $\lambda_{\bigcirc}$ can represent many families of distributions. Some, like the exponential distribution, give rise directly to a sampling function. Others, like the Gaussian distribution, must use a sampling function that implements rejection sampling. Unlike other probabilistic programming languages, $\lambda_{\bigcirc}$ does not provide inference, only queries for expected values and for computing Bayes' theorem. $\lambda_{\bigcirc}$ does not reason about sampling error, and suffers the same accessibility problems as other probabilistic programming languages. *Uncertain*$\langle T \rangle$ uses sampling functions to represent distributions, in the same way as $\lambda_{\bigcirc}$, as I will discuss in Chapter 5.

Another alternative is to approximate the probability density function directly. The APPL programming language is a library that allows programmers to compute explicitly with distributions [Glen et al., 2001]. APPL represents distributions with piecewise functional representations of their probability density function. While this approach simplifies implementing some operations and preserves accuracy, it rapidly becomes unwieldy. For example, the authors show that the density function for the sum of 12 uniform random variables is a piecewise function in 12 parts with a total of 133 terms. This rapid growth in the size of the representation makes APPL impractical for all but the simplest problems.

In the same vein, PaCAL is a Python library for computing with independent random variables [Jaroszewicz and Korzeń, 2012]. PaCAL approximates probability density functions with Chebyshev polynomials, a particular sequence of orthogonal polynomials that, in a linear combination, can approximate any function defined on $[-1, 1]$. This representation makes many operations simple to define, but cannot account for dependencies between variables and requires programmers to be explicit about the distributions they are manipulating.

2.3 Domain-specific solutions for uncertainty

In the absence of a general approach for programming with uncertainty, many domains have their own idiosyncratic techniques. These techniques can provide some insight into potential attributes of a more general solution.

Robotics

Robotics is a common application of probabilistic programming – a robot needs to sense information about its surroundings and combine evidence from these sensors in a principled way. CES is a C++ library, motivated by robotics, for programming with random variables using templates [Thrun, 2000]. For example, CES represents a distribution over floating point numbers as an instance of type `prob<double>`. The type `prob<T>` stores a list of pairs $(x, p(x))$ that map values to probabilities, like the probability monad. This restricts CES to simple discrete distributions. But the ideas of encapsulating distributions as a first-order type and using operator overloading to apply operations to distributions mean CES presents an accessible interface to programmers. For this reason *Uncertain*$\langle T\rangle$ uses a similar interface.

Databases

Reflecting uncertainty in databases is a long-studied problem, made more important by the growth of big data. Hazy is a framework for programming with uncertain databases [Kumar et al., 2013]. Hazy asks programmers to write Markov logic networks to interact with databases while managing uncertainty. A Markov logic network is a set of rules and associated confidences for each rule, that together describe the relationship between elements in the database. For example, say we have a database of students and the papers they have written. We could define a rule that says with 30% confidence that if student s is an author of paper t, and s is a student of professor p, then p is also an author of paper t. Markov logic networks encourage programmers to think at a high level about their probabilistic models. But this approach is limited to databases and has accessibility problems, requiring programmers to write Markov logic networks rather than write in their usual programming language.

Other approaches to probabilistic databases include Dalvi and Suciu, who describe a "possible worlds" semantics for evaluating queries on probabilistic databases [Dalvi and Suciu, 2007], and Benjelloun et al., who unify probabilistic databases with data lineage to soundly evaluate probabilistic queries [Benjelloun et al., 2006]. Like Hazy, these approaches are specific to databases.

Geolocation

Geolocation uses inputs such as Global Positioning System (GPS) sensors and WiFi fingerprinting to estimate a user's location. The locations produced are estimates and have uncertainty [Thompson, 1998]. Geolocation libraries are extremely popular in smartphone applications; a cursory study indicates that 40% of top Android applications use geolocation.

Programmers using geolocation must account for uncertainty to produce the most accurate results. For example, one approach for matching GPS samples to road maps uses a Hidden Markov Model to evaluate the best matches [Newson and

Krumm, 2009]. A Hidden Markov Model is a type of generative model suitable for inference. The authors define their own error models for their GPS readings, because the platform did not provide a standardised approach to measuring GPS error. The model achieves good results, but to create it, the authors needed extensive background knowledge of statistics and probability. We cannot expect all consumers of geolocation data to have such faculty. Implementing such an approach in a geolocation library would improve its results somewhat, but would still not consider the effect of the programmer's computations on the data.

Gesture interaction

The rise of touch screens as input devices makes input processing more complex. For simple input devices like the mouse, inputs are focussed at a single point. But for touch screens, the input covers an area of the screen, making the user's intended target ambiguous. One technique to resolve this ambiguity is to treat the input gesture as a probability distribution over the screen [Schwarz et al., 2010, 2011]. This approach allows the system to easily incorporate other data. For example, in the event of a complex gesture like dragging, the system can defer resolving the target of the gesture until more information is gathered (like the direction of the drag). This reduces the incidence of incorrectly recognised inputs.

Software benchmarking

Software benchmarking is fundamental to experimental computer science. Researchers often wish to compare configurations for performance or efficiency; for example, comparing two garbage collectors. For a fair test, researchers run these experiments over suites of benchmarks, thought to be a representative sample of computer programs [Blackburn et al., 2006].

Because modern computer systems are heavily non-deterministic, these experiments produce noisy results. The variance in the results may outweigh the actual difference between the configurations. But too often, researchers neglect this possibility [Georges et al., 2007]. Existing analysis techniques allow researchers to ignore uncertainty in the results, making it too easy to reach the wrong conclusion. While some proposed solutions exist, this problem is caused fundamentally by the lack of easily accessible programming languages that address uncertainty as a first-order concern. Such a language would make it much more difficult for programmers to mistakenly ignore uncertainty.

2.4 Other techniques for uncertainty

Probabilistic programming is not the only way to reason about uncertainty. Rather than generative models and probability distributions, other techniques model uncertain problems differently.

2.4.1 Interval analysis

Interval analysis represents the uncertainty in a value as a simple interval in the domain [Moore, 1966]. In the real numbers, an interval is a subset of $\mathbb{R}$ consisting of all numbers between an upper bound and a lower bound. Interval analysis captures uncertainty by representing the range of possible values a variable can take, and propagating this range through computations. For example, if $X = [\pi/6, \pi/2]$, then $\sin X = [0.5, 1]$, since for each $x \in X$, $0.5 \leq \sin x \leq 1$, and $\sin \pi/6 = 0.5$ and $\sin \pi/2 = 1$, so the bound is tight. Interval analysis is particularly suited to scientific computation, for bounding the effect of floating point error.

The advantage of interval analysis is its simplicity and efficiency. Many operations over real numbers are trivial to define over intervals; among other results, if f is monotonic on $[a, b]$, then $f([a, b]) = [f(a), f(b)]$. In cases like this one, computing the resulting interval is only marginally more expensive than computing with a single point – we need only compute the values of the function at the two endpoints. The primary downside of interval analysis is that it effectively treats all random variables as having uniform distributions. Such a treatment loses track of the fact that some values are more likely than others; the result is an analysis that is far too conservative for most practical purposes outside the world of scientific computation.

2.4.2 Fuzzy logic

Fuzzy logic is a generalisation of the usual binary propositional logic where variables now have a *degree of truth* between 0 and 1 [Zadeh, 1965]. Fuzzy logic builds on the idea of fuzzy sets, which generalise the usual ideas of set theory to ascribe to each element of the set a degree of membership between 0 and 1. Many of the usual set-theoretic and logical operations have fuzzy equivalents. Like probability, fuzzy logic offers a way to reason about variables and problems that are uncertain.

Fuzzy logic is similar to a type of probabilistic programming where all the variables are Bernoulli distributions. Because of this restriction, we can more easily define many operations, and build an intuitive view of models using set theory. But this view is restrictive, because not all distributions are Bernoullis. While many distributions can be approximated in this way (for example, we can approximate a Gaussian by the mean of many independent Bernoullis), the encoding is inefficient and unintuitive. Fuzzy logic is not generally extensible to the types of problems programmers face when dealing with uncertain data.

2.5 Emerging trends

Uncertainty is not a new problem in many domains. But a number of emerging trends make it more important than ever to reason about and control uncertainty at the programming language level.

2.5.1 Approximate computing

Energy efficiency is replacing performance as the driving force in technological change, from the lowest architectural level to the highest software abstraction. Many programs do not require the computational correctness guaranteed by the underlying hardware, and can cope with some level of error in some calculations. *Approximate computing* exploits this property to improve performance and energy efficiency, since often less reliable operations are cheaper to evaluate.

Many programming models for these unreliable architectures exist. Baek and Chilimbi present Green, a framework for providing approximate versions of an operation and ensuring the overall execution meets quality of service (QoS) guarantees [Baek and Chilimbi, 2010]. Green calibrates the selection of approximations at compile time based on training data, producing a QoS model that is used to select the appropriate locations for approximate operations to be inserted. EnerJ is a framework for programmers to annotate which parts of their code the runtime can approximate, and track how approximate and exact data interact to control error propagation [Sampson et al., 2011]. EnerJ's type system encourages programmers to think about which operations induce error.

Esmaeilzadeh et al. propose a hardware architecture for executing neural networks to approximate software operations [Esmaeilzadeh et al., 2012b]. The compiler identifies functions in the program that a neural network could approximate, trains the neural network, and instead of inserting calls to the function, inserts calls to the neural network hardware to set up and evaluate the network. The result is an approximation of the true value, and should be faster to compute than executing the original function. Other hardware approximation techniques include reducing the power to memory, which results in a higher rate of read and write errors, and reducing the width of floating point registers, which reduces the precision of computation but saves silicon real estate and thus energy.

Rely is a programming language that statically reasons about the quantitative reliability of a program [Carbin et al., 2013]. The programmer specifies an execution model of the unreliable hardware their program will run on (for example, that memory reads flip a bit in the word with probability 10^{-7}), and annotates their code with assertions about the accuracy of the output of an operation in terms of the accuracy of its input. The Rely analyser takes the execution model and the code specifications and verifies that the program executing on that hardware will satisfy the assertions. Approximate computing is not a panacea – there is some limit to the quantum of approximation beyond which the program will not execute acceptably – and Rely reasons about the probability that a program is reliable.

2.5.2 Stochastic hardware

A growing hardware trend is the advent of stochastic hardware, which produces outputs that are probabilistically correct. A stochastic processor executes error-tolerant applications while balancing performance demands and power constraints [Narayanan et al., 2010]. Error-tolerant applications allow the processor to choose between different functional units and different voltage levels to achieve the reliability requirements of the application while conserving energy. But in the worst case, the output of the stochastic processor is only correct with some probability, reflecting the fact that there is a limit to the level of error most programs can tolerate. Similarly, Truffle is a proposed instruction set and microarchitecture that supports approximate programming via dual-voltage hardware operation [Esmaeilzadeh et al., 2012a].

At an even lower level, probabilistic CMOS devices have a fixed probability of correct computation [Chakrapani et al., 2006]. By not requiring guaranteed correctness, these probabilistic CMOS (or PCMOS) devices can achieve significant energy efficiency and performance improvements for many applications.

2.5.3 Sensor ubiquity

Consumers have rapidly adopted smartphones over feature phones. More than half of all mobile phones sold today are smartphones, with sales on track to reach 1.8 billion units in 2013 [Gartner, 2013]. The key difference between smartphones and feature phones is the advanced connectivity of the smartphone. Most smartphones have a wide array of sensors: cameras, Global Positioning System (GPS) sensors, accelerometers, gyroscopes, barometers, thermometers, and countless more. These sensors allow application programmers to deliver compelling experiences by reasoning about the user's personal context.

Programmers are embracing this new-found plethora of sensors. For example, over 40% of Android applications use the GPS sensor (Section 3.2). The applications that use these sensors are diverse, from social networking to photography, retail, and fitness. The ubiquity of these sensors and the explosion of the mobile marketplace has meant that more non-expert programmers are consuming more estimated sensor data in their applications. These programmers cannot be expected to understand the intricacies of uncertainty when writing everyday applications, but almost inevitably these applications will consume uncertain data. We must deliver new programming models for these programmers to consume this data in an accessible but correct way.

2.6 Summary

These disparate emerging trends and existing problem domains share a common issue: they lack compelling programming models for working with uncertain data. Approximate computing produces uncertain results, stochastic hardware results in uncertain execution, and hardware sensors produce estimates of real-world quantities. Many other domains also face uncertainty as a critical issue. But too often, current programming models treat these uncertain results as if they are certain.

Current programming models are built around APIs that abstract the details of the underlying hardware. These models incorrectly assume that uncertainty is an implementation detail to be abstracted away. Current APIs assume that there is no reason for most programmers to care about error, and that those who do will reason about it manually. Most programmers live up to this assumption by completely ignoring uncertainty, but it rests on a faulty premise. As Chapter 3 will show, uncertainty is not simply an academic concern, but a real-world phenomenon that affects the correctness of applications.

The central premise of this thesis is that uncertainty is inherent in these applications, and so cannot stop at the API boundary. Programming models must capture the uncertain nature of data, rather than ignoring it. These programming models will need to be accessible, efficient, and expressive, in order to be useful and see wide adoption. Existing work in this field fails in one or more of these areas, and in any event, most existing work is not sufficiently general for widespread use. The challenge is to find an abstraction that satisfies all three elements.

This thesis answers the challenge.

Chapter 3

Motivating Example: GPS Applications

Uncertainty is not just an abstract problem. This chapter demonstrates that real applications suffer for not considering uncertainty, experiencing three kinds of *uncertainty bugs*. They experience random errors by treating estimates as facts. They unknowingly compound error by computing with estimates. And they create false positives and negatives by asking simplistic questions of estimates.

To demonstrate these uncertainty bugs I focus on the example of Global Positioning System (GPS) applications. Most modern smartphones have GPS sensors to estimate the user's location, and many smartphone applications consume this location data through APIs. I focus on this example because GPS applications are simple to understand, pervasive in the mobile computing landscape, and commonly experience the problems that this chapter describes.

3.1 GPS applications

The Global Positioning System is a collection of satellites that together provide location information to Earth-based receivers. Originally for military use, GPS sensors have benefited monumentally from miniaturisation and are now ubiquitous. Almost all modern smartphones feature GPS sensors to provide location information to applications. The operating system provides this location information to applications through a simple API, which abstracts the details of GPS navigation.

3.1.1 How GPS works

GPS calculates the sensor's location by measuring the time taken for a signal from each visible satellite to reach the receiver [Thompson, 1998]. Each satellite broadcasts its orbit location and the exact time of its transmission. The receiver uses time signals to keep its clock in synchronisation with the satellites. The receiver can thus calculate the time taken for the signal to travel from the satellite to Earth, and from there use the speed of a signal through air to calculate the receiver's distance from the satellite.

This distance defines a spherical shell (in 3D), centred on the satellite, of possible locations of the receiver. The signal from a second satellite defines another shell, and the intersection of these shells forms a circle of possible locations. A third satellite's signal reduces that circle to a single point, which is an estimate of the user's location.

The trouble with this system is that it is susceptible to random error at a number of steps. (1) The satellites' understanding of their own positions in space can be in error, which shifts the location of the spheres. (2) The calculation of the distance from the satellite depends on knowing the speed of the signal, which in a vacuum is the speed of light, but the Earth's atmosphere distorts and slows the signal in an essentially random way depending on many factors, including the angle of the signal, weather, interference, and so on. (3) There is a limit to how well the clocks of each satellite and the receiver can be synchronised, which affects the accuracy of the calculated time and thus distance.

These random errors mean that the location calculated by the GPS is only an *estimate* of location. While the accuracy can be quite good in some cases, it can be very poor in others. Because the distances involved are so large, and the time scales so small, even minute variances can result in large errors in the final location.

3.1.2 GPS on smartphones

Most modern smartphones have GPS sensors built in. All modern smartphone operating systems provide broadly similar APIs for accessing GPS data.

On Windows Phone, the API provides geolocation data through the `GetGeopositionAsync` method of the `Geolocator` class. Geolocation is an abstraction that combines different sources of location data – GPS, WiFi fingerprinting, cell fingerprinting, etc. – into a single API. The `GetGeopositionAsync` method returns a `Geoposition` object with two fields: `CivicAddress` and `Coordinate`. The civic address attempts to map the user's location to an address at the suburb level. More importantly, the `Coordinate` field is an object of type `Geocoordinate`. The vast majority of applications that use GPS use this `Coordinate` field to determine the user's location.

Figure 3.1 shows the `Geocoordinate` class. The class specifies the user's location through the latitude and longitude fields, which together uniquely identify a point on the Earth's surface. The class also specifies the accuracy of the fix in metres. Figure 3.2 shows how this accuracy is displayed in mobile applications. The accuracy defines the radius of a circle around the location specified by the latitude and longitude.

3.2 Treating estimates as facts

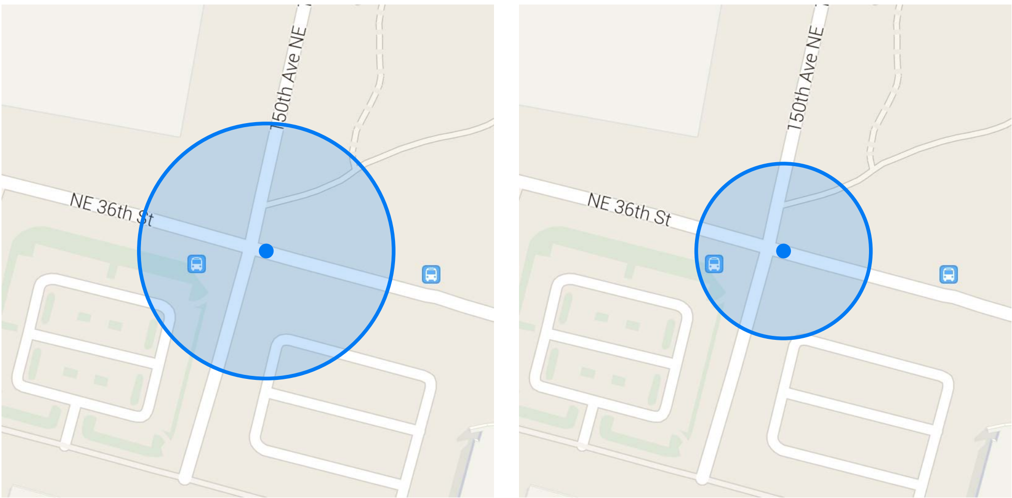
The first type of uncertainty bug is random errors caused by treating estimates as facts. In the GPS case, applications ignore the accuracy of the location and treat the latitude and longitude as the user's true position.

My cursory survey of the top 100 applications on each of Windows Phone and

```
public class Geocoordinate {
  public double Latitude;   // in degrees,  -90 < x <=  90
  public double Longitude;  // in degrees, -180 < x <= 180

  public double Accuracy;   // in metres
}
```

Figure 3.1: The `Geocoordinate` object on Windows Phone



(a) Windows Phone: 95% CI, $\sigma = 22\,\mathrm{m}$

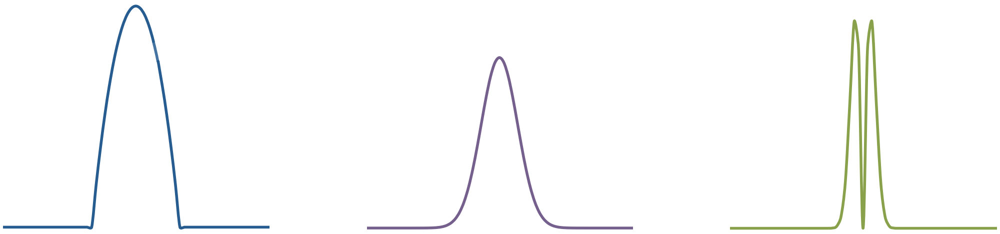
(b) Android: 68% CI, $\sigma = 29\,\mathrm{m}$

Figure 3.2: GPS fixes on two different smartphones

Android shows that 22% of Windows Phone and 40% of Android applications use the GPS APIs to determine the user's location. Static analysis of the Windows Phone applications shows that only 5% of the top 100 read the accuracy field of the `Geocoordinate` object. Manual analysis of these applications finds that only one application – *Pizza Hut* – actually uses this accuracy number in calculations. The other applications use it only for debugging.

Therefore, practically all applications that use the GPS treat its estimate of location as if it were a fact. This usage leads to random errors, because these applications fail to consider the plethora of other possible locations implied by the accuracy radius.

But even reading the accuracy radius is not sufficient. For instance, here are three probability distributions that would each report the same position and accuracy (defined precisely below) using the same abstraction as the `Geocoordinate` API:



Clearly each of these distributions implies a very different set of probabilities for the user's location, but the `Geocoordinate` abstraction cannot distinguish them.

3.2.1 The trouble with accuracy

The root of this problem is that the abstraction that `Geocoordinate` provides is insufficient. The random error that GPS experiences (Section 3.1.1) when determining location results in the GPS effectively returning a *distribution* over possible locations. Each location has a probability of being the user's true location x. By Bayes' theorem, that probability is a function of how likely it is that the GPS generated the result that it did, under the assumption that the user's true location is x. Intuitively, if the GPS generates a location p, the user's true location is unlikely to be a long way from p, because it would take a catastrophic error for the GPS to return such a measurement. For locations close to p, the likelihood that the user's true location is a particular point x depends on the accuracy of the GPS.

The `Geocoordinate` abstraction tries to capture this distribution by returning an accuracy estimate. The accuracy is a number in metres, and is the radius of a circle around the returned location. This circle is a *confidence interval* for the user's location. But the confidence interval does not describe the relative probabilities of each location in the interval. For GPS, it is actually quite unlikely that the user's true location is exactly in the centre of the distribution, but more likely the location lies on a circle near that centre (this is a consequence of the Rayleigh distribution discussed in Section 4.2.1). The `Geocoordinate` abstraction cannot hope to capture this nuance.

The abstraction is also difficult to understand correctly. Each smartphone operating system defines the accuracy estimate differently (emphases added):

Android *"Get the estimated accuracy of this location, in meters. We define accuracy as **the radius of 68% confidence**. [. . .] In statistical terms, it is assumed that location errors are random with a normal distribution."*

iOS *"The **radius of uncertainty** for the location, measured in meters. The [location] identifies the center of the circle, and this value indicates the radius of that circle."*

Windows Phone *"Gets the accuracy of the latitude or longitude, in meters. The accuracy can be considered **the radius of certainty** of the [location]. A circular area [. . .] with the accuracy as the radius and the [location] as the center contains the actual location."*

In fact I found, through reverse engineering, that on Windows Phone the radius actually defines a 95% confidence interval.

The distinction is not merely academic. Figure 3.2 shows GPS fixes taken at the same time and place on two different smartphones. The circle for Windows Phone (Figure 3.2(a)) is larger than the circle for Android (Figure 3.2(b)), which seems to indicate that the fix on Windows Phone was less accurate. But because the two platforms define the circle differently, they are not directly comparable. In fact, normalising for this difference, Windows Phone was actually more accurate. This detail is obscured due to inconsistent treatment of error, and encouraged by the poorly defined abstraction.

3.3 Compounding error

The second type of uncertainty bug is compounding error caused by using estimated data in calculations. Abstractions such as the `Geocoordinate` object encourage programmers to ignore uncertainty when they compute with the API output. In the GPS case, a programmer can take two `Geocoordinate` objects, and calculate the distance between them. Because each location is an estimate, the distance between them must also be an estimate. The `Geocoordinate` abstraction is not powerful enough to track this relationship automatically.

It therefore falls to programmers to try to reason about propagated error. But the relationship between the error of each location and the error of the distance is not obvious. For example, suppose a programmer uses the GPS to calculate the user's speed through the formula $Speed = Distance/Time$, taking samples every one second. Then even if the accuracy of each location is very good, with a 95% confidence interval of 4 m (the most accurate that smartphone GPS generally delivers), the accuracy of the calculated speed is poor, with a 95% confidence interval of 21 km/h. So even

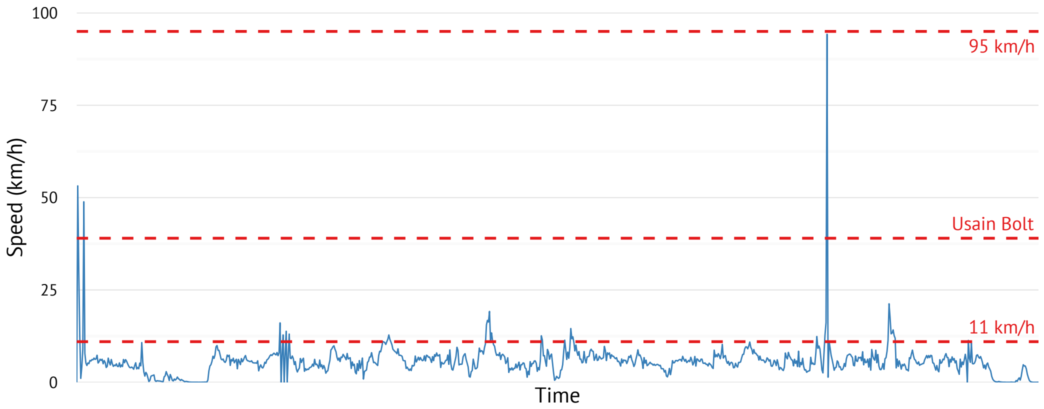


Figure 3.3: Compounding error in GPS walking speeds causes absurd results

very accurate GPS fixes result in significant uncertainty in subsequent calculations, because the distance operation (essentially a subtraction operation on the two points) compounds the error of the two GPS fixes.

To demonstrate this result in practice, I built a simple Windows Phone application that recorded GPS locations every second, and used them to calculate speed between each sample. I used this application while walking. Figure 3.3 shows the speeds the application calculated through the experiment. The average human walks at 5 km/h, and Usain Bolt runs the 100 m sprint at 39 km/h. The experimental data shows an average walking speed of 6 km/h, 35 seconds spent walking faster than 11 km/h (an average running speed), and at one point, a patently absurd walking speed of 95 km/h. These errors are significant in both magnitude and frequency, and are caused by the compounding effect of the speed calculation on the uncertainty in the GPS locations.

3.4 False positives in conditionals

The third type of uncertainty bug is false positives and negatives in conditionals. Most programs eventually act on estimated data by using it in conditional expressions. But estimated data is probabilistic, and naively applying the usual conditional operators to estimated data creates false positives and negatives by failing to consider the effect of uncertainty.

Consider using GPS speed data (as described above) to issue speeding tickets. The application computes speed from the GPS, and compares it to a speed limit of 60 km/h. If the calculated speed is greater than 60 km/h, the application issues a speeding ticket. A programmer implements this specification by calculating the variable *Speed*, and then writing a conditional like:

```
if (Speed > 60)
    IssueSpeedingTicket()
```

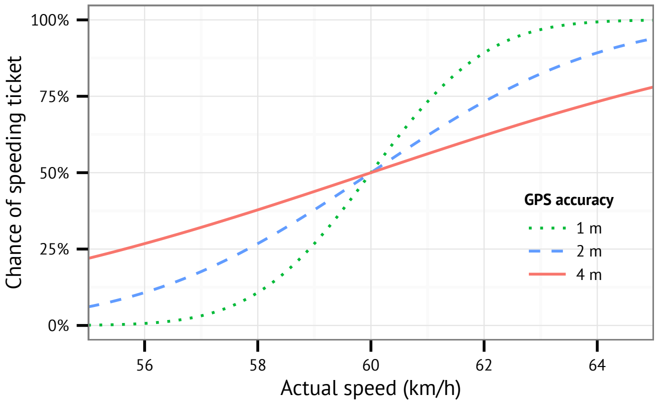


Figure 3.4: Naive conditionals cause false positives when issuing speeding tickets

This conditional does not consider that *Speed* is an estimate. As discussed above, the `Geocoordinate` abstraction is not powerful enough to track the relationship between the locations and the speed automatically, so it falls to programmers to handle this manually. The semantics of conditionals like this one are not clear if we are to consider uncertainty.

This style of naive conditional results in false positives and negatives. Figure 3.4 shows the probability of this program issuing a speeding ticket at various actual speeds and GPS accuracies. For example, at an actual speed of 57 km/h (i.e., 3 km/h *below* the speed limit), if the GPS accuracy is 4 m (very good by smartphone standards), there is still a 32% probability of receiving a speeding ticket due to random error alone.

The issue is that these naive conditionals ask simple questions about probabilistic data. The estimate from the GPS should be viewed as *evidence* towards a conclusion. Rather than asking whether $\mathit{Speed} > 60$, which requires a precise value for *Speed* to answer, programmers should ask *how likely* it is that $\mathit{Speed} > 60$. In the speeding ticket example, we would only want to issue a speeding ticket if this likelihood is high – so the evidence is very strong that the speed is above the limit. The distribution for *Speed* defines this likelihood. Mathematically, the question we want to ask is whether $\Pr[\mathit{Speed} > 60] > 0.95$ (if we say we will only issue speeding tickets when the likelihood is greater than 95%). The `Geocoordinate` abstraction cannot hope to express this type of conditional, and it is unreasonable to expect every programmer to implement it manually.

3.5 Summary

Applications suffer real *uncertainty bugs* by not considering uncertainty appropriately. These applications experience random error by treating estimates as facts, compound error by calculating with estimates, and create false positives and negatives by asking simplistic questions.

Current abstractions for GPS data are responsible for these bugs, because they sacrifice correctness for concision and simplicity. This chapter shows that the bugs that these abstractions cause are non-trivial, and difficult to correct without expert and domain-specific knowledge.

Most importantly, this chapter demonstrates that when faced with the challenge of uncertainty, most programmers simply ignore it. After seeing the limitations of current abstractions, this apathy is understandable. These abstractions offer no insight into how to address the transient, non-trivial bugs that they cause. The challenge these examples poses is simple: is there a simple and intuitive programming language abstraction that correctly represents uncertainty, and empowers programmers to reason about uncertain data? I argue that the uncertain type, $Uncertain\langle T\rangle$, provides exactly such an abstraction.

Chapter 4

The Uncertain Type Abstraction

The uncertain type abstraction, *Uncertain*$\langle T \rangle$, is a programming language abstraction for uncertain data. *Uncertain*$\langle T \rangle$ is a generic data type that encapsulates and manipulates probability distributions. It propagates error through computations, and helps programmers to reason about uncertainty by providing new semantics for conditional expressions. The primary difference between *Uncertain*$\langle T \rangle$ and other existing work is a focus on defining an accessible interface for non-expert programmers to reason correctly about uncertainty.

The key insight of the uncertain type is that considering accessibility as a first-order requirement leads naturally to a restricted subset of probabilistic operators. These operators are expressive enough to solve real-world problems, but are accessible to non-expert programmers and lend themselves to a very efficient implementation.

This chapter discusses the interface the uncertain type provides, and Chapter 5 discusses how *Uncertain*$\langle T \rangle$ implements this interface efficiently and tractably. This distinction is analogous to floating point arithmetic – there is a theoretical interface (arithmetic over the real numbers), and a practical implementation (the floating point specification) that changes the semantics to make the theoretical problem tractable.

Section 4.1 provides an overview of the uncertain type. Sections 4.2 to 4.5 present a four step process to using the uncertain type in programs: selecting the right distribution, computing with distributions, asking questions with conditionals, and improving estimates with domain knowledge. Section 4.6 contrasts the uncertain type with probabilistic programming languages. Finally, Section 4.7 summarises the chapter, highlighting the advantages of the uncertain type in terms of concision, expressiveness, and correctness.

4.1 Overview of the uncertain type

The uncertain type, *Uncertain*$\langle T \rangle$, is a generic data type that encapsulates a probability distribution with domain T. An instance of *Uncertain*$\langle T \rangle$ therefore represents a random variable. The uncertain type overloads a variety of operators so that for many common operations, programmers can write the same expressions they are accustomed to (e.g., $A + B$) and have their program work as they expect.

Table 4.1: *Uncertain*$\langle T\rangle$ operators and methods.

Operators	Arithmetic ($+\ -\ *\ /$)	$op :: Unc\langle T\rangle \to Unc\langle T\rangle \to Unc\langle T\rangle$
	Equality ($< > \leq \geq = \neq$)	$op :: Unc\langle T\rangle \to Unc\langle T\rangle \to Unc\langle Bool\rangle$
	Logical ($\wedge \vee$)	$op :: Unc\langle Bool\rangle \to Unc\langle Bool\rangle \to Unc\langle Bool\rangle$
	Unary ($\neg$)	$op :: Unc\langle Bool\rangle \to Unc\langle Bool\rangle$
	These operators lift operators from type T to work over distributions. For example, the original $+$ operator for numbers has type $Double \to Double \to Double$. $Uncertain\langle Double\rangle$ lifts this operator to $+'$ with type $Uncertain\langle Double\rangle \to Uncertain\langle Double\rangle \to Uncertain\langle Double\rangle$.	
Queries	Expected value	$E :: Unc\langle T\rangle \to SD\langle T^{\div}\rangle$
	Calculates the expected value of a distribution. The return type is a sampling distribution, reflecting the fact that the expected value is only approximate. Only the *Project* operator can directly extract the expected value from the result of this operator. The type $T^{\div}$is the type T equipped with a division operation.	
	Project	$Project :: SD\langle T\rangle \to T$
	Extracts the approximate expected value from a $SamplingDist\langle T\rangle$ object. This operator is an explicit cast from an uncertain value to a certain value, and for this reason is highly discouraged. However, some applications require this operator for correctness (see Section 5.3.1).	
Evaluation	Implicit	$HypTest :: SD\langle T\rangle \to SD\langle T\rangle \to Bool$
	Performs a hypothesis test, at the 95% confidence level, asking if the population means approximated by the two $SamplingDist\langle T\rangle$ arguments are distinct. This method is called implicitly by the $Uncertain\langle T\rangle$ runtime (see Figure 5.1).	
	Explicit	$HypTest :: SD\langle T\rangle \to SD\langle T\rangle \to [0,1] \to Bool$
	Equivalent to the implicit *HypTest*, but allows the programmer to specify the confidence level for the hypothesis test.	

$Unc\langle T\rangle$ is shorthand for $Uncertain\langle T\rangle$, $SD\langle T\rangle$ for $SamplingDist\langle T\rangle$.

The uncertain type represents arbitrary distributions with a *sampling function*, which returns a new random sample of the distribution on each invocation [Park et al., 2005]. Choosing to represent distributions with sampling makes $Uncertain\langle T\rangle$ more expressive, by being able to represent a wide range of distributions that may not have closed forms. Section 4.2 discusses this benefit of sampling in more detail.

$Uncertain\langle T\rangle$ addresses two sources of error: *domain* error and *sampling* error. Domain error is inherent in the problem domain; it is uncertainty, or the difference between an estimate and its true value. For example, a GPS sensor creates domain error when it takes a measurement to estimate location. While domain error is the fundamental target of $Uncertain\langle T\rangle$, it must also address sampling error, which it induces by approximating distributions with random sampling. $Uncertain\langle T\rangle$ accounts for sampling error by evaluating conditionals using expected values, and using hypothesis testing to reason about the query being evaluated. Figure 5.1 discusses sampling error in more detail.

Table 4.1 shows the operators and methods the uncertain type provides. Most operators that work over type T also work over $Uncertain\langle T\rangle$. This operator lifting is possible because of the sampling representation the uncertain type uses for distributions. For example, since $+$ is defined over type *Double*, it is also defined over type $Uncertain\langle Double\rangle$. The lifted operator simply applies the original operator to samples of each operand. This idea extends to other operators, so that an operator of type $op :: T \to S \to V$ is lifted to have type $op' :: Uncertain\langle T\rangle \to Uncertain\langle S\rangle \to Uncertain\langle V\rangle$. Section 4.3 discusses the computation operations the uncertain type provides in more detail.

The uncertain type provides two methods for reasoning about domain error in conditional expressions. Firstly, programmers can directly query the mean (expected value) of a distribution, asking for example whether $\mathbb{E}[X] > 10$. This query is simple, and is distinct from considering only a single sample from the distribution X, as some current abstractions do, because the expected value takes the entire distribution into account. Secondly, programmers can use probabilities, asking for example if $\Pr[X > 10] > 0.95$ (in words, if the probability that $X > 10$ is at least 95%). This probabilistic query asks *how likely* it is that the assertion is true, and therefore empowers programmers to control false positives. The probability $\Pr[X > 10]$ is the *evidence* for the assertion that the underlying value that X estimates is larger than 10. Importantly for an efficient implementation, both these queries evaluate expected values. Section 4.4 discusses the details of these queries.

Representing estimates as distributions gives expert programmers the opportunity to improve the quality of the estimates. Distributions enable Bayesian inference, a principled approach for incorporating domain knowledge into the estimation process. For example, a programmer can specify for areas of a map a different "prior" likelihood that the user's location is in that area. This knowledge might reflect, for example, the user's calendar indicating that they have a meeting at a particular place. Using the uncertain type, which encapsulates the distribution of estimated data, programmers can apply Bayesian inference using this prior knowledge to improve the quality of the estimate. Section 4.5 discusses this key advantage of *Uncertain*$\langle T \rangle$, and how to make it accessible to non-experts.

The next four sections describe in detail how the uncertain type abstraction provides an accessible and expressive interface for programming with uncertain data. These four sections comprise a four step process for using the uncertain type:

> **Identifying the distribution** of uncertain data requires domain expertise, since it depends on the particular data source. Domain experts often already know the distribution of their data, but lack an abstraction to expose this information to programmers.
>
> **Computing with distributions** includes using arithmetic operators, converting units, and combining with other distributions. The uncertain type uses operator overloading to make this step mostly transparent to programmers.
>
> **Asking the right questions** of distributions requires a new semantics for conditional expressions on probabilities rather than binary decisions on deterministic types. The uncertain type defines two types of conditional.
>
> **Improving estimates with domain knowledge** combines pieces of probabilistic evidence. Programmers specify application-specific information through a simple interface, and libraries incorporate this domain knowledge using Bayesian inference, to improve the quality of the estimates they produce.

4.2 Identifying the distribution

The first step in programming with uncertain data using the uncertain type is to identify the underlying probability distribution of the data. Estimation processes experience random error, so the values that they return have associated probability distributions reflecting the magnitude and location of this error. The distribution is domain-specific, since the magnitude and location of the error depends on the specific estimation process and underlying value being estimated.

In many cases, the expert programmers who provide APIs to access estimated data already know its error distribution. For example, to provide the accuracy estimate for the geolocation API shown in Section 3.1.2, the library programmer must know how to convert data from the GPS sensor into an error radius. This conversion depends on the error distribution, and so the programmer must derive a distribution to implement the conversion. In machine learning, training algorithms can often provide a Bayesian distribution of the error in a prediction, rather than just the prediction itself.

These expert programmers lack an abstraction to expose this distribution information through the APIs they build. Existing abstractions sacrifice either correctness by making an inappropriate projection from a distribution to a simpler value (such as the error radius in GPS), or simplicity by exposing too much detail to non-expert programmers who do not want it. Sacrificing correctness leads to uncertainty bugs, and sacrificing simplicity makes other programmers less willing to adopt the APIs.

The uncertain type provides an abstraction that does not force programmers to choose between correctness and simplicity. Expert programmers can implement their APIs using the uncertain type to encapsulate their knowledge about the distribution of the data. The uncertain type provides a simple API surface to non-expert programmers, who can treat the output of these APIs as they currently do, but avoid uncertainty bugs in the process. Using the uncertain type also supports these non-expert programmers in reasoning correctly about the quality of the estimated data, as this chapter describes.

4.2.1 Knowing the right distribution

The expert programmer can determine the right distribution to use for their particular problem in one of two broad ways.

Selecting a theoretical model. Many sources of estimated data are amenable to theoretical models. The literature abounds with examples of estimated data sources and their associated theoretical error models, which expert library programmers can adopt. For example, the uncertainty in the mean of a dataset is approximately Gaussian by the Central Limit Theorem. In the GPS case, the expert programmer can derive a theoretical model from some simple assumptions, as demonstrated below.

Deriving an empirical model. Some problems are not common enough to have existing theoretical models, or are otherwise impervious to theoretical analysis. Expert programmers can determine an error distribution for these problems empirically, using either a statistical bootstrapping approach (taking many samples from the noisy data source while maintaining the same underlying value), or applying machine learning on a set of training data.

Example: GPS data

This section shows how to derive an error distribution for GPS data, to replace the GPS library method

```
Geocoordinate GetGPSLocation();
```

with the new method

```
Uncertain<Geocoordinate> GetGPSLocation();
```

which expresses a distribution over possible locations.

This section adopts the convention that discrete values like Actual_t are set in italics and random variables like $\mathbf{GPS}_t$ are set in bold.

Theoretical setup. Formally, we can express the GPS process as follows. Define

$$\text{World} := [-90, 90] \times (-180, 180]$$

and say that at time t, our true location is the point

$$\mathit{Actual}_t := (\mathit{TrueLat}_t, \mathit{TrueLong}_t) \in \text{World}.$$

Then the GPS sensor's estimate of our location at time t is a random variable

$$\mathbf{GPS}_t = (\mathit{TrueLat}_t + \mathbf{LatErr}_t, \mathit{TrueLong}_t + \mathbf{LongErr}_t). \tag{4.1}$$

Here the random variables $\mathbf{LatErr}_t$ and $\mathbf{LongErr}_t$ represent the error in each direction, due to inherent flaws or biases in the sensor and due to environmental conditions at time t (e.g., atmospheric conditions, obstructions).

The act of taking a GPS sample at time t draws a sample of the random variable $\mathbf{GPS}_t$, which yields a discrete point

$$\mathit{Sample}_t = (\mathit{SampleLat}_t, \mathit{SampleLong}_t).$$

It is this discrete point that most geolocation libraries return. But this clearly ignores the distribution of $\mathbf{GPS}_t$, and in particular the effect of $\mathbf{LatErr}_t$ and $\mathbf{LongErr}_t$.

The distribution of $\mathbf{GPS}_t$ depends on the distributions of $\mathbf{LatErr}_t$ and $\mathbf{LongErr}_t$. Knowing these distributions precisely is difficult, but the literature suggests a model which we will adopt [van Diggelen, 2007]. This model says that $\mathbf{LatErr}_t$ and $\mathbf{LongErr}_t$ are independent and identically distributed (i.i.d.), and follow a Gaussian distribution, with mean zero and a common unknown variance, such that

$$\begin{aligned} \mathbf{LatErr}_t &\sim \text{Normal}(0, \sigma_t^2) \\ \mathbf{LongErr}_t &\sim \text{Normal}(0, \sigma_t^2) \end{aligned}$$

where the fact that the mean is zero reflects an unbiased sensor, and the fact that σ depends on t reflects the environmental conditions at time t.

Belief in location. The first issue with the theoretical derivation is that of course the GPS sensor does not know the value of Actual_t. What we are trying to do is *estimate* the value of Actual_t, based on observations from the GPS sensor. Bayesian statistics provides a framework to formalise this. We introduce a random variable $\mathbf{Location}_t$, which represents our device's *belief* about our location. Initially, we know nothing about our location, so for every point p in the world,

$$\Pr[\mathbf{Location}_t = p] = \text{Uniform}.$$

An oracle could tell us the "perfect" distribution of $\mathbf{Location}_t$, such that

$$\begin{aligned} \Pr[\mathbf{Location}_t = \mathit{Actual}_t] = 1, &\quad \text{and} \\ \Pr[\mathbf{Location}_t = p] = 0 &\quad \text{for all } p \neq \mathit{Actual}_t. \end{aligned}$$

We use Bayes' theorem to incorporate the GPS sensor as evidence into our belief about $\mathbf{Location}_t$. Intuitively, Bayes' theorem tells us that, if we observe a GPS sample Sample_t, the most likely values of $\mathbf{Location}_t$ are exactly those locations most likely to cause the GPS to generate that sample. So, for example, it is unlikely that $\mathbf{Location}_t$ is a long distance from Sample_t, because this implies very large values for $\mathbf{LatErr}_t$ and $\mathbf{LongErr}_t$, which are very unlikely. Formally, Bayes' theorem says that

$$\begin{aligned}&\Pr[\mathbf{Location}_t = p | \mathbf{GPS}_t = \mathit{Sample}_t] \\ &\quad \propto \Pr[\mathbf{GPS}_t = \mathit{Sample}_t | \mathbf{Location}_t = p] \cdot \Pr[\mathbf{Location}_t = p].\end{aligned}$$

Notice that the left hand side is a function of p. For each point p in the world, this function gives the probability that the true location is p, given that we observed a GPS sample Sample_t. This function is called the *posterior* distribution for $\mathbf{Location}_t$, because it represents our belief about the true location after observing a sample from the GPS. Because we have no prior knowledge about $\mathbf{Location}_t$, we assume that $\Pr[\mathbf{Location}_t = p] = 1$, which simplifies this posterior function to

$$\Pr[\mathbf{Location}_t = p | \mathbf{GPS}_t = \mathit{Sample}_t] \propto \Pr[\mathbf{GPS}_t = \mathit{Sample}_t | \mathbf{Location}_t = p]. \tag{4.2}$$

It is this posterior distribution that we want to return from the GPS sensor. Rather than a discrete point, this distribution captures for each point p in the world how likely the user is to be standing at that point, given the evidence from the GPS.

Deriving a likelihood model. In Equation (4.2) for the posterior, the term

$$\Pr[\mathbf{GPS}_t = \mathit{Sample}_t | \mathbf{Location}_t = p]$$

is a *likelihood model* for the GPS sensor. It captures the likelihood of the GPS generating the particular sample Sample_t under the assumption that the true location is p. Substituting p for Actual_t in the expression for $\mathbf{GPS}_t$ (Equation (4.1)) gives

$$\mathbf{GPS}_t = (p_{Lat} + \mathbf{LatErr}_t, p_{Long} + \mathbf{LongErr}_t).$$

So the likelihood that $\mathbf{GPS}_t = \mathit{Sample}_t$ is exactly the likelihood that

$$(\mathbf{LatErr}_t, \mathbf{LongErr}_t) = \mathit{Sample}_t - p. \tag{4.3}$$

But in our model, we do not know the variance of the distributions for $\mathbf{LatErr}_t$ and $\mathbf{LongErr}_t$, so we cannot evaluate this likelihood directly.

One solution is simply to assume the variance as part of our model. But this assumption does not take into account the fact that the variance depends on the instantaneous environment of the sensor. Most GPS sensors also give an estimated confidence interval for the GPS error. This confidence interval ε_t is the value in metres such that there is a 95% probability that Actual_t is within ε_t metres of the GPS sample. GPS sensors estimate this confidence interval by comparing the data from multiple satellites and calculating the distortion the signals experienced in the atmosphere. Formally, we have that

$$\Pr\left[\|\mathbf{GPS}_t - \mathit{Actual}_t\| < \varepsilon_t\right] = 0.95. \tag{4.4}$$

But observe that

$$\begin{aligned}&\|\mathbf{GPS}_t - \mathit{Actual}_t\| \\ &\quad = \|(\mathit{TrueLat}_t + \mathbf{LatErr}_t, \mathit{TrueLong}_t + \mathbf{LongErr}_t) - (\mathit{TrueLat}_t, \mathit{TrueLong}_t)\| \\ &\quad = \|(\mathbf{LatErr}_t, \mathbf{LongErr}_t)\| \,.\end{aligned}$$

Furthermore, since our model assumes that $\mathbf{LatErr}_t$ and $\mathbf{LongErr}_t$ are i.i.d. Gaussians with mean zero, it is a well-known identity that

$$\begin{aligned}\|(\mathbf{LatErr}_t, \mathbf{LongErr}_t)\| &= \sqrt{\mathbf{LatErr}_t^2 + \mathbf{LongErr}_t^2} \\ &\sim \text{Rayleigh}(\rho_t).\end{aligned}$$

for some unknown parameter ρ_t. The Rayleigh distribution is a continuous single-parameter non-negative probability distribution with density function

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

Since $\|\mathbf{GPS}_t - \mathit{Actual}_t\| = \|(\mathbf{LatErr}_t, \mathbf{LongErr}_t)\|$ and we now know the distribution of the right-hand side, we can calculate ρ_t by applying Equation (4.4):

$$\begin{aligned}0.95 &= \Pr[\text{Rayleigh}(\rho_t) < \varepsilon_t] \\ &= \int_0^{\varepsilon_t} \frac{x}{\rho_t^2}\exp\left\{-\frac{x^2}{2\rho_t^2}\right\} \mathrm{d}x \\ &= 1 - \exp\left\{-\frac{\varepsilon_t^2}{2\rho_t^2}\right\} \\ \therefore \rho_t &= \varepsilon_t \Big/ \sqrt{\ln\left[(1-0.95)^{-2}\right]} \\ &= \varepsilon_t \Big/ \sqrt{\ln 400}\,.\end{aligned}$$

So now we know that

$$\|(\mathbf{LatErr}_t, \mathbf{LongErr}_t)\| \sim \text{Rayleigh}(\varepsilon_t / \sqrt{\ln 400}) \tag{4.5}$$

where ε_t is known from the GPS sensor. This gives us a trivial way to estimate the likelihood in Equation (4.3): we can say that the likelihood that

$$(\mathbf{LatErr}_t, \mathbf{LongErr}_t) = \mathit{Sample}_t - p$$

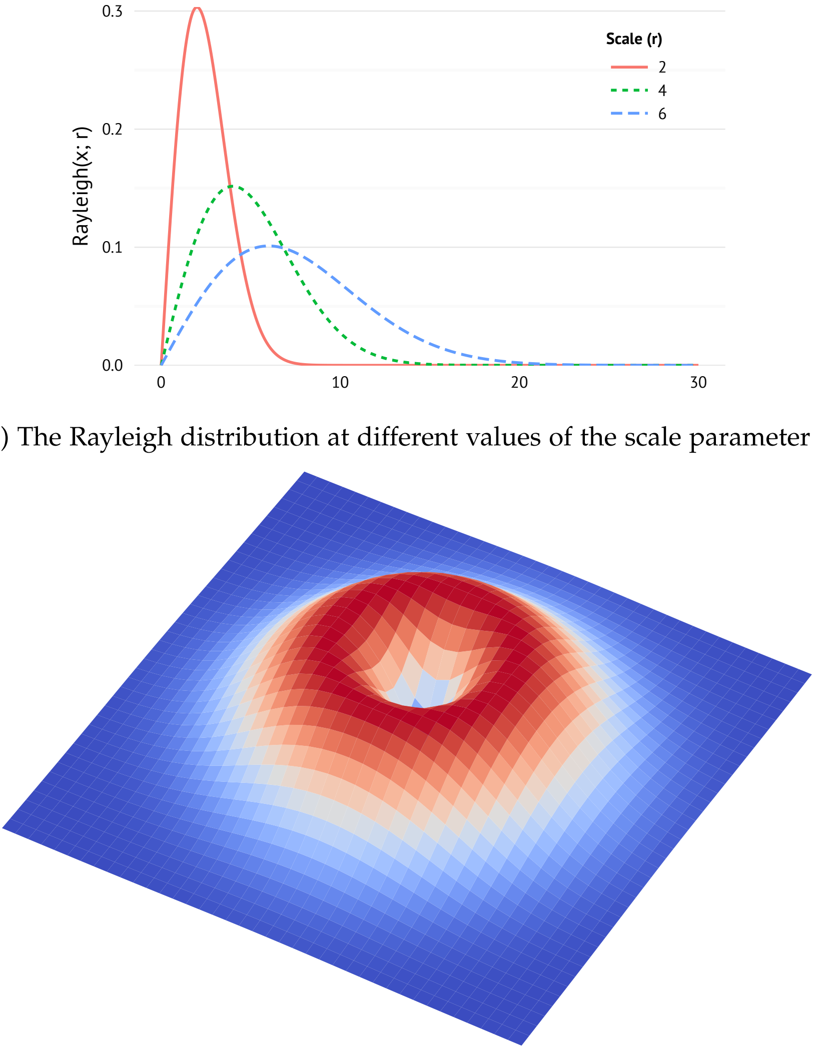
is approximated by the likelihood that

$$\|(\mathbf{LatErr}_t, \mathbf{LongErr}_t)\| = \|\mathit{Sample}_t - p\|\,.$$

Since we know how to evaluate this likelihood, we have found a way to evaluate

$$\Pr[\mathbf{GPS}_t = \mathit{Sample}_t | \mathbf{Location}_t = p]$$

which is the likelihood model needed to evaluate the posterior function in Equation (4.2). This is exactly the distribution function we want to return from the GPS API.



(a) The Rayleigh distribution at different values of the scale parameter *r*

(b) The posterior distribution for location is a distribution over the Earth's surface

Figure 4.1: The GPS returns a distribution based on the Rayleigh distribution

Summary. We derived a posterior distribution for the user's location based on GPS evidence. The GPS sensor returns to us a location $Sample_t$ and a confidence interval ε_t. Then for any point p in the world, the likelihood of the user being at location p given this GPS sample is

$$\begin{aligned}\Pr[\mathbf{Location}_t = p | \mathbf{GPS}_t = Sample_t] &= \Pr[\mathbf{GPS}_t = Sample_t | \mathbf{Location}_t = p] \\ &= \text{Rayleigh}(\|Sample_t - p\|\,; \varepsilon_t / \sqrt{\ln 400}) \end{aligned} \tag{4.6}$$

up to a normalizing constant.

Figure 4.1 shows the Rayleigh distribution and the resulting posterior distribution. Notice the posterior distribution in Figure 4.1(b) carries little mass in the centre, implying that a user is actually quite unlikely to be at the centre (which is the location $Sample_t$ returned by the GPS sensor). This counter-intuitive result becomes obvious only by properly considering the effect of GPS error.

4.2.2 Representing arbitrary distributions

A random variable is completely defined by its probability distribution. The uncertain type therefore encapsulates probability distributions in order to work with random variables. There are many techniques for representing a probability distribution in a programming language.

The most accurate approach is to store the probability distribution exactly. For discrete distributions (i.e., with finite probability space, like the Bernoulli distribution defined over $\{True, False\}$), this involves storing a map from each possible value to its probability. The probability monad takes this approach (Section 2.2.1). For continuous probability distributions (i.e., with infinite probability space), we could store the probability density function algebraically. For example, a Gaussian random variable with mean μ and variance σ^2 has density function

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

Encapsulating a specific Gaussian random variable requires storing only this formula and the values of μ and σ^2, and so a Gaussian random variable can be stored in constant space (up to floating point error).

This style of exact representation has two major downsides. Firstly, an algebraic representation quickly becomes impractical due to symbolic explosion. For example, while a Gaussian random variable has a simple density function, the sum of two Gaussian random variables is the convolution of their density functions – an integral that is not easily evaluated. Thus an algebraic representation is impractical for even basic calculations. Secondly, many important distributions do not have closed form density functions. Distributions for sensors, approximate hardware, road maps, and machine learning often cannot be represented exactly, but these domains are exactly the types of problem the uncertain type should be able to handle, since these are the fields where non-expert programmers consume uncertain data.

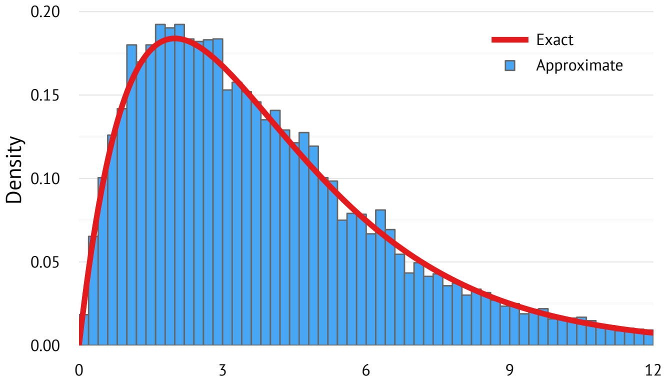


Figure 4.2: Random samples approximate the exact probability density function

For this reason, the uncertain type represents distributions by random sampling. Given a probability density function $f(x)$, the idea of random sampling is that a sample is more likely to take the value x when $f(x)$ is large than when $f(x)$ is small. Drawing a set of many samples in this manner therefore approximates the probability density function. In particular, the histogram of the samples approximates $f(x)$, since if $f(x)$ is large in a particular region of values of x, then those values are more likely to be sampled, and so will be larger in the histogram. Figure 4.2 demonstrates this idea, showing an exact probability density function and a histogram of 10,000 random samples drawn from it.

Specifically, the uncertain type represents distributions by storing a *sampling function*, which returns a new random sample from the distribution each time it is invoked. This same idea pis roposed by Park et al. for a probabilistic programming language [Park et al., 2005], discussed in Section 2.2.4. Given infinite time and space, this approach approximates distributions arbitrarily well, as a corollary of the Glivenko-Cantelli theorem [Topsøe, 1970]. In practice, the sample size embodies the classic speed-accuracy trade-off. Selecting the correct sample size ahead of time is a difficult problem, but one of the key insights of the uncertain type is to use query-specific information to determine the sample size at run time. I discuss this idea in Section 5.3.

Sampling makes for an expressive abstraction because it allows the uncertain type to represent a large range of distributions that do not have convenient closed forms. This is a key shortcoming of much of the existing work in probabilistic programming (Section 2.2), and is fundamental for bringing the ideas of probabilistic programming into the world of everyday programmers. These programmers will not tolerate the complexity of the probability monad and the stochastic lambda calculus, no matter how clever and elegant these representations may be.

```
public class GPSLib {
    // We need to convert between metres and degrees when using the error estimate
    private double EARTH_RADIUS = 6371*1000;
    private double DEGREES_PER_METRE = Math.Degrees(1/EARTH_RADIUS);

    public Uncertain<Geocoordinate> GetGPSLocation() {
        // Firstly, get the estimate from the hardware
        Geocoordinate Point = GetSampleFromGPSHardware();
        double ErrorRadius  = GetErrorEstimateFromGPSHardware();

        // Compute the parameter rho of the Rayleigh distribution (Equation 4.4)
        double rho = ErrorRadius / Math.Sqrt(Math.Log(400));

        // Define a sampling function for this distribution. The sampling
        // function takes no arguments, and returns a Geocoordinate -- a sample
        // from the distribution defined by Point and ErrorRadius.
        Func<Geocoordinate> sampler = () => {
            // Sample the surface (Figure 4.1(b)) in polar coordinates -- the
            // radius is a Rayleigh sample and the angle uniform in [0,2pi)
            double radius = Math.RandomRayleigh(rho, SAMPLE_SIZE);
            double theta  = Math.RandomUniform(0, 2*Math.PI);

            // Convert the polar coordinates to x,y coordinates in degrees
            double x = Point.Longitude + radii*Math.Cos(theta)*DEGREES_PER_METRE;
            double y = Point.Latitide  + radii*Math.Sin(theta)*DEGREES_PER_METRE;

            return new Geocoordinate(x, y);
        };

        // Create an Uncertain<Geocoordinate> that uses the sampling function
        return new Uncertain<Geocoordinate>(sampler);
    }

    // These methods expose the raw estimate from the hardware

    // Return the sampled point
    private Geocoordinate GetSampleFromGPSHardware();
    // Return the estimated 95% confidence interval
    private double GetErrorEstimateFromGPSHardware();
}

public class Math {
    // Generate a Rayleigh random sample
    private double RandomRayleigh(double rho) {
        // If X and Y are N(0, s^2) then R = sqrt(X^2 + Y^2) is Rayleigh(s)
        double x = RandomNormal(0, rho, size);
        double y = RandomNormal(0, rho, size);
        return Math.Sqrt(x*x + y*y);
    }
    // Most math libraries provide these functions
    private double RandomNormal(double mean, double stdev);
    private double RandomUniform(double low, double high);
}
```

Figure 4.3: An implementation of a GPS library using the uncertain type

Example: GPS data

In Section 4.2.1 I showed how to derive the distribution for GPS sensor data. Figure 4.3 shows how to represent this distribution as a sampling function. This code defines the GPS library function

```
Uncertain<Geocoordinate> GetGPSLocation();
```

which takes a GPS sample and returns its associated distribution. Lines 17 to 28 define the sampling function for a particular GPS sample captured on line 8. This sampling function is a closure that captures the values of the `Point` and `rho` variables. Note that each invocation of this sampling function returns a new sample from the GPS *distribution*, but there is still only a single sample from the GPS sensor itself. The GPS hardware functions are not called when invoking the sampling function that this method returns.

4.3 Computing with distributions

The second step in programming with uncertain data using the uncertain type is computing with it. Computing combines data through various operators – arithmetic, logical, and comparisons. The key challenge in computing with estimated data is addressing the question of independence between the operands.

4.3.1 Independent random variables

Two random variables are independent if the value of one has no bearing on the value of the other (Section 2.1). For independent random variables of type $Uncertain\langle T\rangle$, we implement computations by lifting operators from type T to work over distributions. For example, the addition operator over floating point numbers has type $+ :: Double \to Double \to Double$. The uncertain type lifts this operator to have type $+' :: Uncertain\langle Double\rangle \to Uncertain\langle Double\rangle \to Uncertain\langle Double\rangle$.

This lifting operation is made simple by the uncertain type's representation of random variables as sampling functions. For independent random variables, if a is a sample from X, and b is a sample from Y, then the sum $a + b$ is a sample from $X + Y$. The lifted operator therefore simply draws a sample from each operand and applies the original operator to them. This is another benefit of using sampling functions rather than an exact representation – the uncertain type easily represents compound distributions (built by combining other distributions through operators) without the symbolic explosion caused by manipulating the probability density functions directly.

4.3.2 Dependent random variables

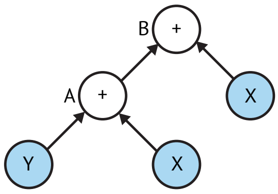
The situation is not as simple for random variables that are not independent. The issue is that the value of a sample from Y may depend on the chosen value of a sample from X. The probability monad (Section 2.2.1) addresses this problem by defining a function of type `X -> Dist Y`, that defines for each possible value of X the distribution of Y given a particular value a of X. This works for discrete distributions, but for the more interesting case of continuous distributions, this representation would require a very complex mapping function. Other possibilities include explicitly providing a sampling function for the joint distribution (X, Y), so that each invocation returns a sample (a, b) that reflects the dependency between X and Y. These approaches are all complex, and the problem of working with dependent variables is an open one in probabilistic programming.

However, we can introduce a useful distinction between two types of dependencies. Some problems have *inherent* dependencies between variables. For example, there is a dependency between a person's age and their strength. A person's age does not completely determine their strength, but there is a relationship between them that is ignored if we treat two random variables *Age* and *Strength* as independent. These inherent dependencies are difficult to reason about because they require domain knowledge, and are a shortcoming of most probabilistic programming approaches.

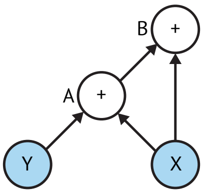
But in probabilistic programming, we also have *programmer-induced* dependencies, which are created when a programmer manipulates and computes with random variables. For example, suppose a programmer takes two independent random variables X and Y, and writes a simple computation with them:

```
A = Y + X
B = X + A
Display(B.E())
```

The computation for B has a programmer-induced dependency, since A depends on X, but the other operand in B also involves X. The naive interpretation of this program produces this expression tree:



This expression tree does not reflect the dependency correctly: both A and B must depend on the *same* instance of X. When the program comes to evaluate `B.E()`, the obvious way to evaluate an expression tree is to work from the bottom up. But to do so here means that when evaluating B, there is no longer any sign of the effect of X on A, because the entire subtree for A has been replaced by single value of A. In essence, evaluating an expression tree is a local operation, but dependence is a global property. The correct interpretation of this program produces instead a graph:



When evaluating `B.E()`, this interpretation correctly represents the dependencies in the program, because the shared effect of X on A and on B is reflected by ensuring all references to X are to the same instance of X.

Many probabilistic programming languages do not handle this type of programmer-induced data dependency, but the uncertain type can, as Section 5.2 describes.

4.3.3 The effect of sample size

Now that we know how to define arithmetic on distributions, we can demonstrate the effect of sample sizes alluded to in Section 4.2.2. In this experiment, we take two Gaussian random variables of known means μ_1 and μ_2 and shared variance. We then compute the sum of these two variables, by taking N samples from each and summing them pairwise. The mean of the resulting samples approximates the mean of the sum of the variables, which we know to actually be $\mu_1 + \mu_2$ since the variables are independent.

Figure 4.4 shows the results of this experiment at different sample sizes N. The left-hand graph shows the time taken to compute the result (i.e., to sum the N samples), and the right-hand graph shows the relative magnitude of the error in the computed result compared to the true value $\mu_1 + \mu_2$. As we said earlier, the sample size embodies the classic speed-accuracy trade-off. At small sample sizes, the calculation is very quick, but very inaccurate. At large sample sizes, the reverse is true: the calculation is very slow but very accurate. Choosing the correct sample size is therefore critically important: too high and the uncertain type will be too slow for practical use; too low and it will be too inaccurate to solve real problems. I discuss how the uncertain type exploits query-specific information to determine the right sample size at run time in Section 5.3.

4.4 Asking the right questions

The third step in programming with uncertain data using the uncertain type is asking questions with conditional operators. In Section 3.4, we saw that using the usual conditional operators on estimated data leads to false positives and negatives. For example, a coffee shop application might send a notification when $\mathit{CoffeeShopDistance} < 200$. But of course, if *CoffeeShopDistance* is calculated using the GPS, it is only an estimate, and so this conditional asks a deterministic question of probabilistic data. This error leads to false positives – in this case, saying the distance is less than 200 m when it is actually not.

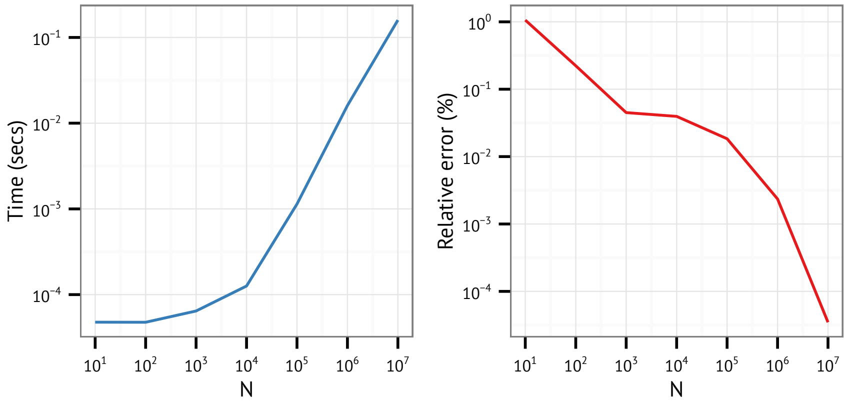


Figure 4.4: Sample size trades speed for accuracy when approximating distributions

The uncertain type evaluates conditionals by considering the *evidence* for a conclusion. Instead of asking "am I within 200 m of the coffee shop?", with the uncertain type programmers ask "how much evidence is there that I am within 200 m of the coffee shop?". This type of question accounts for uncertainty correctly in an intuitive style, and aids programmer accessibility since conditionals still return booleans.

4.4.1 Two styles of conditionals

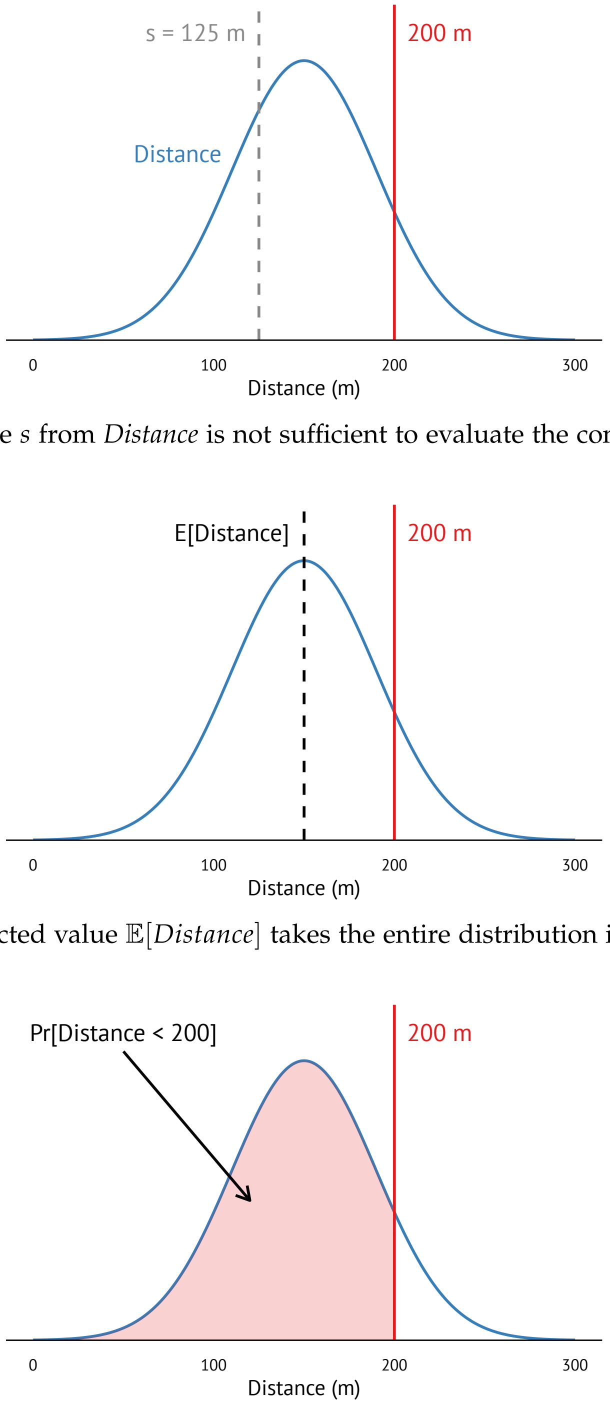
Figure 4.5 shows the situation for the coffee shop conditional. Figure 4.5(a) shows how the conditional would be evaluated without the uncertain type – a single sample s from the distribution (which the program knows nothing about in this case) is compared to 200. Clearly this does not adequately reflect the distribution, and it is entirely possible for this sample s to end up on the wrong side of 200, and therefore trigger a false negative.

The uncertain type has two types of conditional that address this problem. The uncertain type evaluates these conditionals using *expected values*, which are well-defined for almost all distributions in practice. That all conditionals involve expected values is critical to efficiently evaluating conditionals, as discussed in Section 5.3.

Comparing means. Figure 4.5(b) shows how the uncertain type compares the *mean* of the distribution to 200. The programmer writes:

```
if (Distance.E() < 200)
    NotifyUserNearCoffee()
```

$Uncertain\langle T\rangle$ evaluates the mean of the distribution *Distance* and compares it to 200. Using the mean is not the same as using a single sample from the distribution, because the mean is a weighted average of the *entire* distribution. This style of conditional is more intuitive for programmers, because it requires only a minor modification from the original approach, and is suitable for cases where the programmer does not need to explicitly consider the quality of the evidence.



(a) A single sample s from *Distance* is not sufficient to evaluate the conditional correctly.

(b) The expected value $\mathbb{E}[Distance]$ takes the entire distribution into account.

(c) The distribution implies a *probability* that $Distance < 200$.

Figure 4.5: Different interpretations of the conditional $Distance < 200$

Comparing evidence. Figure 4.5(c) shows how the uncertain type evaluates *evidence* for an assertion. Here the assertion is that $Distance < 200$. The area under the distribution to the left of 200 m is the probability $p \in [0,1]$ that $Distance < 200$. Programmers use evidence to write:

```
if ((Distance < 200).E() > 0.85)
    NotifyUserNearCoffee()
```

which asks if there is more than an 85% chance that $Distance < 200$. As described in Section 4.3, the uncertain type lifts operators like $<$ to work over distributions. So the inner expression $Distance < 200$ evaluates to a distribution of type $Uncertain\langle Boolean\rangle$. The code then takes the expected value of this Bernoulli distribution, which is exactly the probability that $Distance < 200$ is true.

The power of evaluating evidence explicitly is that programmers can reason about false positives and negatives. If *Distance* is a very wide distribution (i.e., the value is very uncertain) with a mean less than 200, there is still high likelihood that the underlying true value is larger than 200. Evaluating the evidence for the assertion, and comparing it to a programmer-selected threshold, allows the programmer to decide the trade-off between false positives and false negatives that is suitable for the particular query. A higher threshold requires stronger evidence to satisfy, which reduces false positives but increases false negatives.

4.5 Adding domain knowledge

The final step in programming with uncertain data using the uncertain type is to exploit domain knowledge to improve the quality of estimates. The previous sections treated the estimation process as immutable, but in many cases, adding domain knowledge to estimation will improve accuracy. The uncertain type unlocks this capability because it encapsulates entire distributions, and therefore can exploit Bayesian statistics.

4.5.1 Bayesian inference

Bayesian statistics represents the state of the world with probability distributions that reflect *degrees of belief* about the true value of a variable. Rather than selecting a single most likely value for the variable, Bayesian statistics instead recognises the uncertainty in the variable's value by recording the likelihood of each possible value. For example, a Bayesian view of the output from a GPS sensor provides a degree of belief for each possible location on a map. For each point on the map, there is a probability that the user's true location is that point. Of course, the user's actual true location is fixed and certain; these probabilities represent the degrees of belief as dictated by the available evidence.

Figure 4.6 demonstrates why Bayesian statistics is desirable in situations involving uncertainty. Rather than recording the single most-likely value of the variable

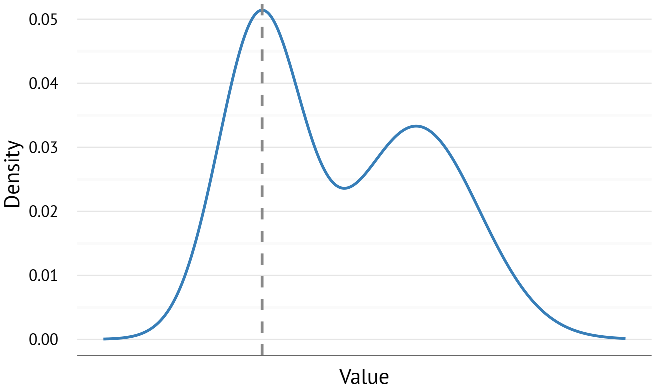


Figure 4.6: Bayesian statistics considers more than a single most-likely value

(the dashed line), a Bayesian viewpoint records the entire probability distribution, assigning a likelihood to each possible value. This viewpoint reflects the situation more robustly: in Figure 4.6, while the dashed line is the most-likely value, there is a 74% probability that the value is larger than this. This nuance is lost by ignoring the distribution.

For estimation processes, there are two random variables, one representing the target variable B we are trying to estimate, and another representing an estimation process E. Bayes' theorem says that

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

The fixed value e is evidence from the estimation process. The left-hand side, a function of the possible true value b, is called the *posterior* distribution. Bayes' theorem allows us to combine the evidence from the estimation process with a hypothesis about the true value.

We call $\Pr[B = b]$ (again a function of b) the *prior* distribution, because it is our belief about the value of B before observing any evidence. The prior distribution is a hypothesis about the value we are trying to model, and specifies domain-specific information. For example, a prior distribution for the speed of a car would specify very low probabilities above 150 km/h, because even before observing any evidence it is very unlikely that a car travels this fast.

The term $\Pr[E = e | B = b]$ is a likelihood model for the estimation process. This function of b specifies the probability that the estimation process would output the value e if the true value of B was b. The likelihood model expresses the error distribution for an estimation process. For a well-behaved estimation process, the likelihood $\Pr[E = e | B = b]$ will be highest for those values of b closest to e – because a good estimator is very unlikely to experience large errors.

The power of Bayes' theorem is that it provides a rigorous model for combining observed evidence with hypotheses. If the hypotheses are accurate, applying Bayes' theorem results in a Bayesian posterior distribution that is more accurate than the prior or the evidence alone. This improves the quality of the estimate, by returning a distribution that more precisely reflects the actual value of the estimated variable. This process is known as Bayesian inference. The uncertain type can exploit Bayesian inference because it encapsulates entire distributions. A single point estimate (like the dashed line in Figure 4.6) is not sufficient to apply Bayes' theorem, and so programmers without the uncertain type must resort to ad-hoc heuristics to improve the quality of estimates.

4.5.2 Incorporating knowledge through priors

In our Bayesian inference framework, programmers specify their domain knowledge as a prior distribution (that is, as a hypothesis). The uncertain type then incorporates this domain knowledge into the estimation process in a rigorous way.

Consider the example of GPS data. In Section 4.2.1 we saw how to derive a posterior distribution for estimating a user's location using the GPS sensor. The resulting posterior distribution (Equation (4.6)) said that

$$\Pr[\mathbf{Location}_t = p | \mathbf{GPS}_t = \mathit{Sample}_t] = \text{Rayleigh}(\|\mathit{Sample}_t - p\|\,; \varepsilon_t / \sqrt{\ln 400}).$$

In the language of Bayes' theorem, the target variable B is $\mathbf{Location}_t$ and the estimation process E is $\mathbf{GPS}_t$. Along the way, this derivation specified a prior distribution for the target variable

$$\Pr[\mathbf{Location}_t = p] = \text{Uniform}.$$

In the Bayesian framework, this prior distribution is a hypothesis about the user's location. We specified the uniform distribution because in this case we assume no specific knowledge about the user's location.

The idea of incorporating knowledge through priors is that in some situations we do have extra domain knowledge. In the GPS case, we may have some information about where the user is more likely to be. For example, suppose we can infer that it is quite likely the user is driving (perhaps because the application is for driving navigation). We can specify this information as a prior distribution over locations, that assigns higher probabilities to roads and lower probabilities to the rest of the map.

Figure 4.7 shows how this idea might work (in a simplified world where locations are one-dimensional). The prior distribution specifies a higher probability for a section of the map corresponding to a road, and a lower probability elsewhere. The likelihood distribution reflects the output of the GPS sensor – the point p is the point estimate, and the distribution reflects the error model from Equation (4.6). The posterior distribution applies Bayes' theorem to the prior and likelihood. Notice how more of the mass of the posterior distribution is centred on the road. The mean of the posterior distribution is the point s. This new estimate of the user's location is not perfectly on the road, because the GPS actually provides quite strong evidence that the user is not on the road.

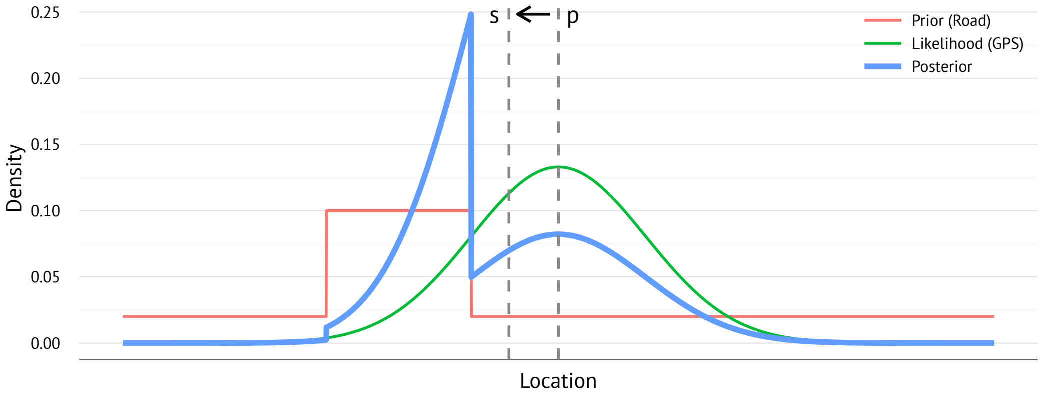


Figure 4.7: Domain knowledge from a prior distribution shifts the estimate

This example reflects an important benefit of Bayesian inference over ad-hoc approaches. Bayes' theorem balances the prior and the evidence in a principled way. If the prior is wrong (in the GPS example, if the user is not actually on a road), strong evidence from the GPS can override it. How strong that evidence must be to do so depends on how certain the prior distribution is. This balance lets programmers reason more intuitively about how to improve the quality of estimates, by explicitly specifying the strength of their hypotheses.

4.5.3 Making Bayesian inference accessible

The difficulty with using Bayesian inference to improve estimates is that specifying prior distributions requires a knowledge of statistics more advanced than that required to use the rest of the uncertain type. In general, programmers using the uncertain type need only a simple understanding of probability to improve the correctness of their programs. On the other hand, specifying a prior distribution requires the programmer to understand Bayesian statistics and be able to determine the appropriate prior for their situation.

A middle ground comes in the form of a *constraint abstraction*. Such an abstraction would allow non-expert programmers to specify domain knowledge through a set of flags defined by expert programmers. These expert programmers specify a preset collection of prior distributions for common scenarios related to their particular library. For example, a GPS library would include prior distributions for driving (e.g., roads, and driving speeds), walking (walking speeds), and being on land. The library exposes these prior distributions through a set of flags that the non-expert programmer can toggle when calling the library. The library applies the selected priors to the estimation process, improving the quality of the estimates without significant burden on everyday programmers.

Hiding the prior distributions behind this abstraction opens a variety of exciting applications of Bayesian inference. A library could use machine learning to improve the quality of its estimates over time. For example, a GPS library could learn the user's location history, and improve the quality of future GPS estimates by incorporating this historical knowledge as a prior. This approach would often produce better GPS estimates, since humans are creatures of habit, but still allows strong new GPS evidence to override the history if the user travels somewhere new. The constraint abstraction makes this learning and correction transparent to the programmer consuming the data, who sees only an improved GPS estimate.

4.6 Probabilistic programming and the uncertain type

Section 2.2 described in detail the idea of *probabilistic programming*, using programming languages to reason about probabilistic models. This chapter described the uncertain type, a programming language abstraction for reasoning about uncertainty in data. At first glance, it may seem that the uncertain type is a form of probabilistic programming – both involve combining random variables using operators, and conclude with questions being asked about the data.

But unlike most applications of probabilistic programming, the uncertain type does not reason about arbitrary posterior distributions through inference. Section 2.2.3 detailed the poor performance of inference algorithms even on simple problems. Because the uncertain type is embedded inside a deterministic host language, it does not reason about entire models but rather only particular instances of a problem. This distinction means the uncertain type does not have to explore very unlikely regions of a model.

This difference in semantics is best illustrated by a simple hypothetical example. Suppose we have a simple mobile device with two sensors, one for measuring the device's velocity, the other for its acceleration. Both of these sensors provide only estimates of the underlying physical quantities, and suppose we have data sheets specifying the error distribution for each sensor. In software, these sensors both return distributions of type *Uncertain*⟨*Double*⟩.

Suppose we are implementing the library for the acceleration sensor, which has a hardware bug. The sensor underestimates the device's acceleration by a factor of 2 when the device is moving at more than 10 km/h. We would like to correct for this effect in the software library. The problem defines two random variables *Speed* and *Accel*. We could represent this problem as a model in a probabilistic programming language as follows:

```
Speed = GetSpeedReading()
if Speed > 10:
    Accel = GetAccelReading() * 2.0
else:
    Accel = GetAccelReading()
```

This probabilistic program defines a generative model – a joint distribution over the variables *Speed* and *Accel*. Inference allows us to ask questions about this model, for example, “how likely is it that *Accel* is greater than $1\,\mathrm{m\,s^{-2}}$?”. The inference algorithm would take the joint distribution the model defines, and marginalise out the variable *Speed* to leave a distribution over *Accel* (i.e., aggregate over all possible values of *Speed* so the remaining distribution involves only *Accel*). This marginalisation explores both branches of the program to understand how *Accel* depends on *Speed*.

We can also implement this same correction using the uncertain type:

```
Uncertain<double> Speed = GetSpeedReading();
Uncertain<double> Accel;
if (Speed.E() > 10)
    Accel = GetAccelReading() * 2.0;
else
    Accel = GetAccelReading();
```

Though this program appears similar to the first, it actually reasons about a conditional distribution rather than a joint one. The uncertain type is embedded inside an existing programming language, and so this program does not explore both branches – to do so would conflict with the host language’s semantics. At the end of this segment of code, the variable `Accel` holds the conditional random variable $\Pr(\mathit{Accel} \mid \mathit{Speed})$. The evaluation has implicitly conditioned *Accel* on *Speed*. The uncertain type never infers the marginal distribution $\Pr[\mathit{Accel}]$ as the probabilistic program does.

This is a virtue of the uncertain type, not a vice. The conditional distribution is appropriate because in this problem domain, *Accel* never occurs in isolation – it is always conditioned on *Speed*. Probabilistic programming solves a different problem – inferring distributions that are not explicitly defined – in a way that is more abstract and general. For applications that consume estimated data (from e.g., sensors, machine learning, approximate hardware, big data), the problem is not an abstraction but a concrete instance where the conditional distribution is appropriate. For example, temperature depends on humidity, altitude, and other variables. When programming with a temperature sensor, the question is not whether temperature can ever be greater than 30°, but rather whether a particular measurement, which is always conditioned on particular values of humidity and altitude, is greater than 30°.

The reduced set of functionality required to reason only about a specific instance works in our favour, because it allows the uncertain type to be considerably more efficient than existing, generic inference algorithms. We should not try to force a generic set of semantics onto our problem if a more efficient, specific approach exists.

4.7 Summary

The uncertain type is an abstraction for handling uncertain data in programming languages. It provides an interface that is accessible to non-expert programmers but still remains expressive enough to solve real problems. This interface makes extensive use of operator overloading, making programs more concise compared to those that manually reason about uncertainty. Since programmers can reason about the quality of evidence, rather than naively evaluate inappropriate conditionals, the uncertain type improves the correctness of programs. The uncertain type also exploits domain knowledge in the form of prior distributions to improve the quality of estimated data.

This chapter describes an ideal abstraction for programming with uncertain data. The challenge this thesis now turns to is implementing this abstraction in a way that is tractable and efficient. As we will see, the restricted semantics that the ideal abstraction defines enable an efficient implementation, exploiting the choice of sampling functions as a representation to improve significantly over more general probabilistic programming approaches.

Chapter 5

Implementing the Abstraction

The previous chapter described the uncertain type, *Uncertain*$\langle T \rangle$, a programming language abstraction for uncertain data. That chapter showed how the uncertain type provides a simple, accessible interface to non-expert programmers. It showed the expressive range of operations the uncertain type provides in order for programmers to reason correctly about uncertainty.

This chapter addresses the third promise of the uncertain type: that it delivers a programming model that is efficient enough for practical use in real-world applications. In Chapter 2 we saw that many existing programming models for uncertain data suffer poor performance because they are too general. The semantics of the uncertain type are restricted to uncertain data, making the problem tractable and the implementation efficient. This chapter describes the insights that enable this efficiency.

Section 5.1 describes how the uncertain type exploits lazy evaluation to avoid eagerly producing unnecessary precision. Section 5.2 shows how the uncertain type eliminates programmer-induced dependencies through this lazy evaluation framework. Section 5.3 explains how the uncertain type addresses *sampling error* (induced by approximating distributions) by defining conditional operations in terms of hypothesis tests. Importantly, these insights all work in concert; each depends on the others to deliver on the uncertain type's promise of efficiency and tractability.

5.1 Lazy evaluation

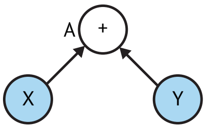
Section 4.2.2 discussed the benefits and drawbacks of different techniques for representing probability distributions in a programming language. Many potential representations are inadequate for the uncertain type because they cannot encapsulate a sufficiently large variety of distributions. For this reason, the uncertain type uses random sampling to represent distributions, in the form of sampling functions. Random sampling defines a distribution in terms of samples from it, which can encode a much broader range of possible distributions than, for example, storing a closed form density function.

Most distribution representations suffer significant performance issues when combining distributions. For example, a representation using approximating polynomials requires 133 terms to hold the sum of 12 uniform random variables [Glen et al., 2001]. A naive approach to random sampling, representing a distribution as a vector of n random samples, would suffer the same type of performance issue. Taking the sum of two random variables would require adding the two vectors element-wise, resulting in n addition operations and a new vector of length n. If n is large enough to ensure reasonable accuracy, this implementation would make arithmetic operations prohibitively slow (see Figure 4.4).

Fixing a sample size n in advance also forces the program to compute a level of precision that is potentially unnecessary. For example, suppose we have two Gaussian random variables X and Y, each with mean 5 and small variance. We would like to decide whether $\mathbb{E}[X + Y] > 1$. Evaluating this expected value requires summing the vectors for X and Y together pairwise and then computing the mean of these n new samples. But the true value of the mean of $X + Y$ is 10, much larger than the threshold of 1 we are testing against, and the variance in the new samples is very small. After computing only a few samples of $X + Y$, it is clear with very high confidence that $\mathbb{E}[X + Y] > 1$. Because each sample will be very close to 10, which is a long way from 1, the central limit theorem implies that it is extremely unlikely that the true expected value is very far from 10. The naive implementation cannot short-circuit this logic to stop after only a few samples, because we cannot know in advance whether future operations will require more samples when they use the result of the current operation.

The uncertain type uses lazy evaluation to ensure that the runtime only ever computes as much precision as necessary at each conditional query. The uncertain type can make use of problem-specific information because the only points where distributions are evaluated involve expected values (Section 4.4), and so we can apply the central limit theorem. In particular, the uncertain type never reasons about intermediate distributions, but rather only about distributions used in queries (which involve expected values) or through explicitly evaluating the expected value of a distribution.

Lazy evaluation of distributions with the uncertain type involves constructing expression trees when executing the various operators. The leaves of the tree are sampling functions defined elsewhere; the inner nodes are operators that combine or modify their operand children. For example, the statement $A = X + Y$ results in the runtime building the following expression tree:



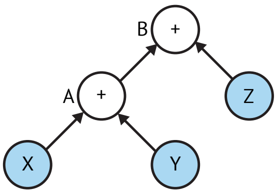
Neither leaf distribution is evaluated when the program executes this statement; concrete values are only computed when a query is performed on A. The constructed expression tree is itself a distribution of type *Uncertain*$\langle T \rangle$.

Queries trigger evaluation of a distribution by drawing random samples from it. Drawing a sample from an expression tree proceeds recursively in a bottom-up fashion. We draw samples from the leaf nodes (which are sampling functions), and then propagate these samples up through the tree. At each inner node with operator op', we apply the original unlifted operator op to the incoming samples (recall from Section 4.3 that the uncertain type lifts an operator of type $op :: T \to S \to V$ to have type $op' :: Uncertain\langle T\rangle \to Uncertain\langle S\rangle \to Uncertain\langle V\rangle$). The result of this process at the root of the tree is a sample from the root random variable.

5.1.1 Graphical models

To demonstrate why this sampling approach works, we observe that our expression trees are actually a Bayesian network. A Bayesian network is a type of probabilistic graphical model, widely used in machine learning to reason about the relationships between variables [Bishop, 2006]. A Bayesian network is a directed acyclic graph, in which each node is a random variable. The edges in the graph indicate the dependencies between the variables; a directed edge from U to V indicates that V depends on U.

Bayesian networks are a valuable representation because they encode the factorisation of the joint distribution of the random variables they contain. For example, consider this simple expression tree for the program $A = X + Y; B = A + Z$:



This model contains five random variables (X, Y, Z, A, B). The shaded variables X, Y and Z are leaf nodes, representing sampling functions defined elsewhere. This graph defines a joint distribution for the five variables:

$$\Pr[X, Y, Z, A, B] = \Pr[X]\Pr[Y]\Pr[Z]\Pr[A|X, Y]\Pr[B|A, Z]$$

Notice how each edge in the graph becomes a dependency in the factorisation, so for instance, A depends on X since there is an edge from X to A. We call the set of nodes a variable depends on the *parents* of the node, denoted pa_X, even though in a normal expression tree they would actually be children, not parents. So for example, $\mathrm{pa}_A = \{X, Y\}$ and $\mathrm{pa}_X = \emptyset$. With this notation, we can write

$$\Pr[\mathcal{Q}] = \prod_{V \in \mathcal{Q}} \Pr[V \mid \mathrm{pa}_V]$$

where $\mathcal{Q}$ is the set of all random variables in the model.

One approach to draw samples from the random variable B is to draw samples from the joint distribution over all the variables in the expression tree. Each of these samples will be correctly distributed over all the variables, and so by simply discarding all the elements of the sample apart from the value for B we achieve correctly-weighted samples for B. A simple technique for sampling the joint distribution is *ancestral sampling*. Because the graph is directed and acyclic, we can topologically order the nodes of the graph, such that each node appears in the list after all of its parents. To sample the joint distribution we simply visit the nodes of the graph in this topological order, at each step drawing a sample from the conditional distribution $\Pr[V \mid \mathrm{pa}_V]$. The topological ordering ensures that all the variables in pa_V already have samples to use in the conditional distribution.

Drawing samples from the leaf nodes, where $\mathrm{pa}_V = \emptyset$, is trivial, since leaf nodes are exactly those with sampling functions already defined for them. For inner nodes, where $\mathrm{pa}_V \neq \emptyset$, we must know how the distribution of V depends on each of its parents. In expression trees, each inner node is associated with an operator, so the distribution $\Pr[V \mid \mathrm{pa}_V]$ of an inner node V with operator *op* is simply a pointmass distribution centred at the point $op(\mathrm{pa}_V)$ (i.e., the result of applying *op* to the samples from the parent nodes). Applying this process recursively up the expression tree results in a sample from B that is correctly distributed.

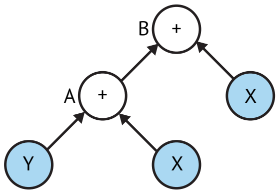
5.2 Eliminating programmer-induced dependencies

Section 4.3.2 highlighted the issue of dependent random variables. The above process for combining random variables works if the leaf distributions are independent. But if they are not, a more sophisticated approach is needed. However, there is a useful distinction between *inherent* dependencies, where the variables are inextricably related, and *programmer-induced* dependencies. When a programmer writes code involving random variables, she can create dependencies between variables. For example, suppose a programmer takes two independent random variables X and Y and writes

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

The computation for B has a programmer-induced dependency, because B depends on X and on A, which also depends on X.

Dependencies become an issue when we apply the expression tree approach from the previous section. If a query evaluates B, the samples it draws from B will not be correctly distributed, and therefore the results will be incorrect. A naive parsing of this code into an expression tree yields:



But this expression tree does not reflect the dependency correctly because each leaf occurrence of X is different. To see why, recall that the expression tree is a graphical model where each node represents a different random variable. This expression tree therefore describes a scenario with *two* random variables X_1 and X_2 that happen to share the same distribution. The graphical model actually describes the joint distribution

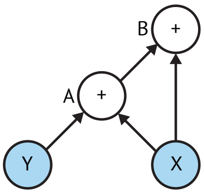
$$\Pr[X_1, X_2, Y, A, B] = \Pr[X_1] \Pr[X_2] \Pr[Y] \Pr[A|X_1, Y] \Pr[B|A, X_2].$$

When we apply the ancestral sampling technique to draw samples from B, we will in fact draw *two* samples from X – one for X_1 and one for X_2.

The solution is to ensure there is only one instance of X that both A and B depend on. To correctly capture the dependency, we actually want the joint distribution

$$\Pr[X, Y, A, B] = \Pr[X] \Pr[Y] \Pr[A|X, Y] \Pr[B|A, X].$$

This factorisation translates into the graphical model:



This model makes clear that the operands for B are not independent – both operands depend on X. When we apply ancestral sampling to this model, we will only draw one sample from X, and reuse it for both $\Pr[A|X, Y]$ and $\Pr[B|A, X]$. Notice that this model is no longer an expression tree (though we will continue calling it such), but rather a directed acyclic graph.

While this correction seems simple, it demonstrates the power of the uncertain type's approach to lazy evaluation, and in turn its choice of sampling functions to represent distributions. Other representations, which eagerly combine distributions when evaluating operators, would lose this dependency information and therefore incorrectly compute the final distribution. For example, if we stored distributions as probability density functions, and eagerly executed the addition operator when called, the first addition $A = Y + X$ would immediately compute a new distribution for A. Then, when computing $B = A + X$, there would be no record of A's dependency on X, and so the addition would treat A and X as independent when they are actually not. The uncertain type's design decisions thus work in concert to deliver correctness when manipulating random variables, all without requiring non-expert programmers to recognise that they have (perhaps indirectly) introduced complex dependencies between variables.

5.3 Evaluating queries with hypothesis tests

Section 4.1 identified the two sources of error that the uncertain type addresses. *Domain error* is inherent in the problem domain, the difference between an estimate and a true value. The uncertain type addresses domain error by providing a new semantics for conditional operators described in Section 4.4. The uncertain type induces *sampling error* by approximating distributions with random sampling. Because computations that use the uncertain type are only ever evaluated through expected values, we can exploit hypothesis testing to address sampling error in a principled and efficient way.

The questions the uncertain type asks compare the means of two distributions. For example, a programmer might write a conditional

```
if (Speed.E() > 10)
    IssueWarning();
```

This question asks whether the expected value of the *Speed* distribution is greater than the expected value of the pointmass distribution with value 10. If we could precisely evaluate distributions, we could compute an exact value for $\mathbb{E}[\mathit{Speed}]$ and compare it directly to 10. But the uncertain type does not precisely evaluate distributions. Instead, we can only *approximate* $\mathbb{E}[\mathit{Speed}]$, by drawing samples from *Speed* and computing sample means. The sample mean approximates the actual expected value, according to the law of large numbers, but the quality of the approximation depends on the sample size, according to the central limit theorem. The uncertain type's expected value operator E (see Table 4.1) reflects this distinction between exact and approximate expected values by returning an object of type $\mathit{SamplingDist}\langle T\rangle$. Except in very specific cases described below in Section 5.3.1, programmers cannot directly access the expected value of type T that this object encapsulates. Instead, they indirectly and transparently compute with the expected value through hypothesis tests.

A hypothesis test is a mechanism for making decisions based on sampled data, and is the commonly accepted method to compare sample means. When the uncertain type executes the conditional above, it establishes a hypothesis test with null hypothesis $H_0 \equiv \mathbb{E}[\mathit{Speed}] \leq 10$ and alternate hypothesis $H_1 \equiv \mathbb{E}[\mathit{Speed}] > 10$. Because we are using sample means, the hypothesis test uses Student's t-distribution to either reject or fail to reject the null hypothesis. If we can reject the null hypothesis at the chosen confidence level, we can accept the alternate hypothesis and therefore return true for the conditional. By default, the uncertain type uses a 95% confidence level, since this is the accepted practice for statistical significance. Programmers can override this choice of confidence level with an additional argument (see Table 4.1).

The challenge is to determine what the right sample size is for the hypothesis test. On one hand, we could draw a very large number of samples, which would almost certainly approximate the true expected value well, but would be slow. On the other hand, we could draw only a few samples, ensuring good performance, but providing

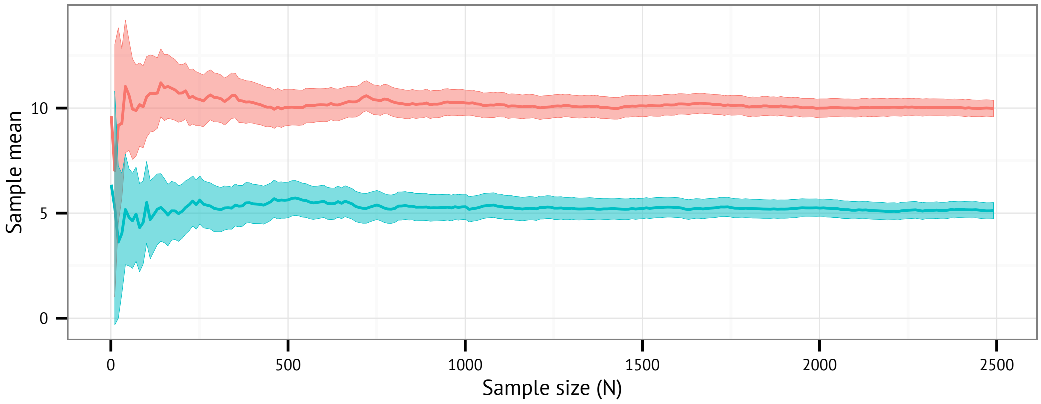


Figure 5.1: Larger sample sizes result in smaller confidence intervals

a poor approximation of the population mean, so the resulting decisions will be very inaccurate. Section 4.3.3 illustrated the nature of this speed-accuracy trade-off. The variety of possible queries means it is very difficult to choose a sample size ahead of time that strikes an ideal balance between speed and accuracy.

The uncertain type's solution to this problem builds on its choice of representation for distributions. Using sampling functions, either defined explicitly by experts for leaf distributions, or implicitly by non-experts in the form of expression trees, allows us to draw an infinite number of samples from a distribution. This profusion of samples overcomes the finite, fixed sample size that a naive implementation would require. We are thus free to dynamically choose a sample size at run time.

To choose the right sample size, we exploit query-specific information when executing the hypothesis test. Once the particular hypothesis test has been established, we progressively increase the sample size used to compute the sample means. We continue increasing the sample size until either (i) the hypothesis test successfully rejects the null hypothesis, or (ii) the sample size reaches a predefined limit, which signifies the uncertainty is too great to evaluate the conditional.

This approach works because the power of a hypothesis tests increases with the sample size. Figure 5.1 demonstrates this effect by comparing two Gaussian random variables with means 10 and 5, respectively. As the sample size increases, the 95% confidence intervals around the two sample means shrink, because the power of the hypothesis tests increases with the sample size. Even after only very few samples, the confidence intervals do not overlap, indicating that the test successfully rejects the null hypothesis. Of course, how quickly this happens depends on how close the true means are, and the variance in each distribution. For this reason we impose a maximum on the sample size to ensure termination.

Note that correctly applying this technique requires careful consideration of how samples are reused. For example, it is a well known problem in medicine that it is unsound to append extra samples to an experiment in order to make it statistically significant [Whitehead, 1999]. Reusing samples can overstate the statistical significance of the result, creating false positives when null hypotheses are incorrectly rejected. We will continue to explore the effect of this reuse for future work.

5.3.1 Ternary logic

The trouble with using hypothesis testing for conditionals is that it introduces a ternary logic and therefore changes the semantics of the uncertain type. Programmers rightfully expect that conditionals satisfy the law of excluded middle – either a condition is true or it is not. They often write simple code like:

```
double Speed = GetSpeedReading();
if (Speed > 10)
    DoSomething();      // assumes Speed > 10
else
    DoSomethingElse();  // assumes Speed <= 10
```

If this program enters the **else** branch, the programmer assumes within that branch that the conditional is false, that is, that $\mathit{Speed} \leq 10$.

A straight translation of this program to use the uncertain type's interface does not appear to invalidate this assumption:

```
Uncertain<double> Speed = GetSpeedReading();
if (Speed.E() > 10)
    DoSomething();      // assumes Speed > 10
else
    DoSomethingElse();  // assumes Speed <= 10
```

But using hypothesis testing to execute the conditional changes the semantics of this program. If the conditional fails and enters the **else** branch, it does not necessarily mean that the converse is true, because failing to reject the null hypothesis is not the same as accepting it. So the conditional can fail either because $\mathbb{E}[\mathit{Speed}] \leq 10$ or because the null hypothesis could not be rejected at 95% confidence.

This semantics change actually introduces a ternary logic. Consider this program without the uncertain type:

```
double Speed = GetSpeedReading();
if (Speed > 10)
    DoSomething();         // assumes Speed > 10
else if (Speed <= 10)
    DoSomethingElse();     // assumes Speed <= 10
else
    assert(false);         // unreachable
```

The third branch is unreachable, because the two conditions in the earlier branches are exhaustive – at least one of them (in fact, exactly one) must be true. This is no longer the case with the uncertain type and hypothesis testing:

```
double Speed = GetSpeedReading();
if (Speed.E() > 10)
    DoSomething();         // Speed > 10 with 95% confidence
else if (Speed.E() <= 10)
    DoSomethingElse();     // Speed <= 10 with 95% confidence
    else
        DoNothing();            // neither true with 95% confidence
```

The third branch is now reachable when neither conditional's null hypothesis can be accepted with 95% confidence.

This distinction is an unavoidable consequence of the uncertain type's choice for representing distributions. We cannot expect to distinguish all possible distributions with an approximation – if we could, it would not be an approximation. In many practical cases, this problem does not arise with a reasonable choice of the maximum sample size, since most practical distributions can be distinguished at some reasonable sample size. Also note that the uncertain type is not unique in having to change its ideal semantics in order to accommodate the realities of a practical implementation. Floating point arithmetic is in a similar situation; the ideal semantics of real arithmetic are not practical on a computer, and so the semantics must change in the implementation. Programmers have adapted to this change (by not directly comparing floating point numbers), and we suggest the same can be true of the uncertain type, potentially with the aid of compiler guidance and static analysis.

For applications that *require* the total ordering property this behaviour breaks, the uncertain type provides the *Project* operator (see Table 4.1). This operator takes an object of type $SamplingDist\langle T\rangle$, returned from the expected value operator E, and returns the raw expected value of type T it encapsulates. This operation is the only point at which uncertain data can become certain data, and is highly discouraged because it allows programmers to ignore uncertainty. Making this operation explicit at least forces programmers to consider whether they truly need to ignore uncertainty for their program to function, which in most cases is not actually necessary.

5.4 Summary

The uncertain type is a principled abstraction for computing with uncertain data. While this principled approach is a virtue, and ensures the uncertain type is accessible and expressive, it also has the potential to make a practical implementation impossible. But insights into lazy evaluation and hypothesis testing make the uncertain type practical to implement in the real world. Lazy evaluation allows the uncertain type to only produce information about a distribution when it is required, rather than eagerly producing unnecessary precision. Hypothesis testing enables the uncertain type to dynamically determine the precision necessary to answer a particular query, exploiting lazy evaluation to avoid wasteful oversampling or failure through undersampling. Lazy evaluation also enables the uncertain type to automatically correct for programmer-induced data dependencies; the seeming simplicity of this insight belies its contribution to making the uncertain type accessible to non-expert programmers.

Using these insights, I implemented the uncertain type in both C# and Python. I believe the uncertain type can be readily implemented in most object-oriented languages, but the vision for an accessible interface requires certain features. The language must support operator overloading to make propagation of uncertainty transparent to programmers. It must also provide an expressive type system which recognises that the type T in $Uncertain\langle T\rangle$ is numeric (i.e., supports arithmetic operations). These requirements made C# and Python (which is dynamically typed) good choices for a prototype implementation, used for the case studies in the next chapter.

The uncertain type's principled design, combined with its implementation insights, ensure that it can achieve its goals of making programs that deal with uncertain data more concise, expressive, and correct. The next chapter demonstrates how these contributions are realised in three case studies.

Chapter 6

Case Studies

My thesis is that by making uncertain data a primitive first-order type, non-expert programmers will write programs that are more concise, expressive, and correct. Previous chapters show how the uncertain type's design and implementation present an abstraction that is accessible, efficient and expressive. This chapter demonstrates through three case studies that programs written with the uncertain type achieve concision, expressiveness, and correctness.

Section 6.1 demonstrates the uncertain type on smartphone GPS data, one of the most widely used hardware sensors, showing how correct reasoning about uncertainty is implemented concisely and leads to improved estimates. Section 6.2 shows how to use the uncertain type to virtually eliminate the effect of noise on a digital sensor, by specifying prior knowledge about the noise. Section 6.3 applies the uncertain type to approximate computing, showing that it is a promising programming model for an emerging trend towards trading accuracy for efficiency.

6.1 Smartphone GPS sensors

Almost every modern smartphone includes a GPS sensor that applications use to reason about the user's current location. My cursory survey in Section 3.2 found that 40% of top Android applications use the GPS. Given the number of applications an average user installs, the vast majority of smartphone users likely use GPS applications.

One particularly common application using a smartphone's GPS sensor is fitness, tracking the user's movement while exercising. This case study builds a simple GPS application called GPSWalking, which measures the user's walking speed using the GPS. GPSWalking invokes a modified GPS library that provides the function

```
Uncertain<Geocoordinate> NewGPSLib.GetGPSLocation();
```

as described in Section 4.2. To calculate the user's speed, the application takes two GPS samples, with a fixed time interval between them, and calculates the distance travelled between the samples. Because $Speed = \Delta Distance / \Delta Time$, this calculation estimates the user's speed over a fixed time interval.

Figure 6.1 shows the main loop of the GPSWalking application both with and

```
int dt = 1;

Geocoordinate LastLocation  = GPSLib.GetGPSLocation();
while (true) {
    Sleep(dt); // wait for dt seconds

    Geocoordinate Location = GPSLib.GetGPSLocation();
    double Speed = GPSLib.Distance(Location, LastLocation) / dt;

    Display(Speed);

    if (Speed > 5)
        GoodJobMessage();

    LastLocation = Location;
}
```

(a) Without the uncertain type

```
int dt = 1;

Uncertain<Geocoordinate> LastLocation = NewGPSLib.GetGPSLocation();
while (true) {
    Sleep(dt); // wait for dt seconds

    Uncertain<Geocoordinate> Location = NewGPSLib.GetGPSLocation();
    Uncertain<double> Speed = NewGPSLib.Distance(Location, LastLocation) / dt;

    Display(Speed.E().Project());

    if ((Speed > 5).E() > 0.75)
        GoodJobMessage();

    LastLocation = Location;
}
```

(b) With the uncertain type

Figure 6.1: The main loop of the GPSWalking application

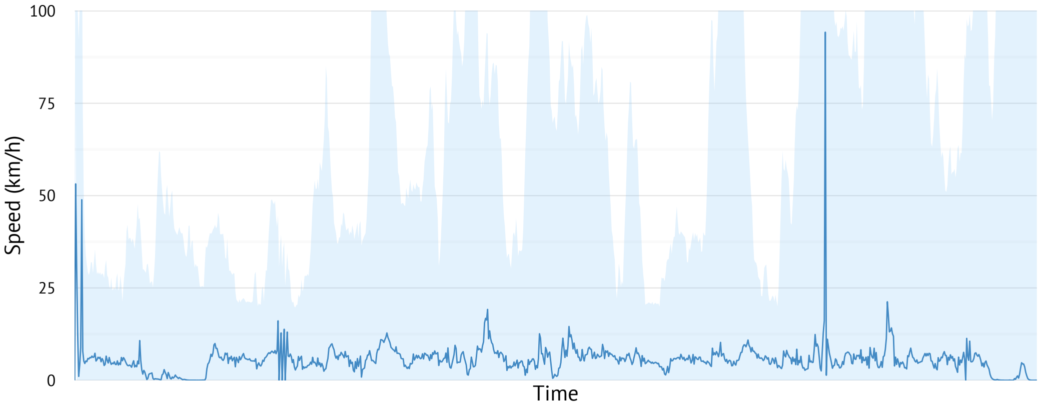


Figure 6.2: The uncertain type provides a 95% confidence interval for speed

without the uncertain type. Once the application has calculated the user's speed, it displays it on the screen. It also tests whether the speed is greater than 5 km/h, and if so, congratulates the user on keeping up a good pace.

Identifying the distribution

Current GPS APIs, described in Section 3.1.2, provide a location and an error radius. Chapter 3 demonstrated that programmers almost always ignore the error information, and as a result, programs experience uncertainty bugs. To fix this, the updated version of `GPSWalking` in Figure 6.1(b) uses a modified GPS library that exposes the error distribution described in Section 4.2.1. In practice, the conversion of this library to use the uncertain type would be done by the expert programmers implementing the platform libraries for the smartphone, and so from the point of view of the non-expert programmer, very little extra work is required. The differences between the versions of `GPSWalking` without the uncertain type (Figure 6.1(a)) and with the uncertain type (Figure 6.1(b)) are minimal, showing that there is very little extra effort required by non-expert programmers to use the uncertain type.

Computing with distributions

`GPSWalking` calculates the user's speed based on their recent locations. Of course, since the locations are estimates from the GPS sensor, distance and therefore speed are estimates too. Since we are only interested in the magnitude of speed rather than the direction, the resulting distribution of speed estimates is of type $\mathit{Uncertain}\langle\mathit{Double}\rangle$.

Programmers must adapt their programs to account for this change. In particular, the call to `Display(Speed)` on line 10 of Figure 6.1(a) must be updated to account for speed being a distribution. Displaying the distribution of speeds is not particularly helpful to most users, so this case warrants using the *Project* that operator the uncertain type provides. The updated version of this call is therefore

```
Display(Speed.E().Project());
```

which displays the mean of the speed distribution.

Distributions for the estimated speeds explain the incorrect speed results we saw in Section 3.3 and Figure 3.3, where the GPS produced walking speed estimates that were consistently too high, and at one point an absurdly high 95 km/h. Figure 6.2 shows the same data but with 95% confidence intervals calculated with the uncertain type in GPSWalking. These confidence intervals are very wide, indicating that the speed estimates are very uncertain. This result confirms the hypothesis of Section 3.3: the absurd speeds are caused by compounding uncertainty making computations unreliable.

Asking the right questions

GPSWalking encourages users to walk fast enough to get a good workout. The version of the program without the uncertain type implements this specification with the following conditional:

```
if (Speed > 5)
    GoodJobMessage();
```

Of course, because speed is an estimate, this naive conditional is susceptible to false positives and negatives. For example, despite the fact that all the data in Figure 6.2 were collected while the user was walking, 12% of the estimated speeds are greater than 8 km/h, a jogging pace, and 4% are greater than 11 km/h, a running pace.

The uncertain type allows the programmer to balance false positives and negatives by evaluating *evidence* for a conditional. Section 4.4.1 describes the two types of conditionals that the uncertain type provides. The modified version of GPSWalking evaluates the conditional using evidence explicitly:

```
if ((Speed > 5).E() > 0.75)
    GoodJobMessage();
```

This conditional asks whether the probability that $Speed > 5$ is at least 75%. The choice of 75% reflects the programmer's chosen balance between false positives and false negatives – higher thresholds give fewer false positives but more false negatives. Using this conditional, less than 1% of estimated speeds are greater than 8 km/h.

Adding domain knowledge

Using the updated GPS library means the application is reasoning about distributions, and therefore has the power of Bayesian statistics at its disposal. Programmers can use prior knowledge to improve the quality of estimated data. In the case of GPSWalking, the application is targeted at users walking for fitness. We can therefore assume that the application is only used when the user is on foot, and encode this knowledge as a prior distribution over speeds. It is extremely unlikely that a person walks at 20 km/h or above, and somewhat more likely that they walk at 4 km/h. The uncertain type can combine the prior distribution specifying this domain knowledge with the evidence from the GPS sensor. Importantly, strong evidence from the GPS sensor can still override the prior distribution.

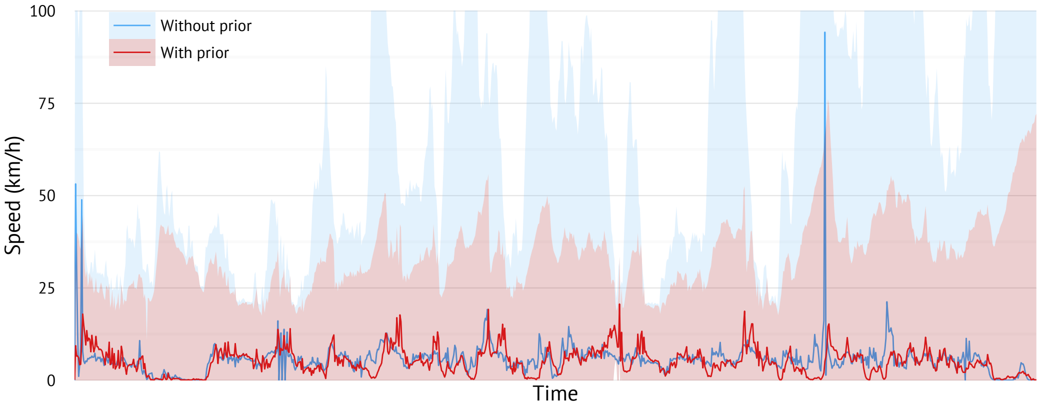


Figure 6.3: Prior knowledge improves the quality of estimated speeds

Figure 6.3 shows the results of applying a prior distribution over speeds to the GPSWalking application. The absurd spikes are removed from the data, and the 95% confidence interval is much tighter, reflecting the extra information we have incorporated. In a real implementation, this prior distribution would be implemented by an expert programmer and activated by the application programmer through the constraint abstraction (Section 4.5.3), modifying the call to the GPS library:

```
Uncertain<Geocoordinate> GPSLib.GetGPSLocation(GPSLib.WALKING);
```

This abstraction absolves application programmers from having to explicitly identify the right prior distribution for their use case, and instead asks them only to determine which properties are applicable.

Summary

GPSWalking is an application that is trivial to translate from existing code to use the uncertain type. This translation rewards the programmer with improved correctness, by both reasoning correctly about absurd data, and eliminating it with added domain knowledge. This type of complex logic would be difficult for even expert programmers to implement without the uncertain type, and certainly outside the realm of the everyday programmer's skill set. The uncertain type eases this burden by encapsulating the GPS error distribution, providing a transparent way to propagate this error through computations, and an abstraction for incorporating extra domain knowledge to improve the results. GPSWalking shows that the uncertain type can deliver on its promise of helping non-expert programmers write applications that are more concise, expressive, and correct.

6.2 Digital sensor noise

This case study was performed by Todd Mytkowicz as part of our conference paper about the uncertain type, which is currently under submission [Bornholt et al., 2013]. I include it in this thesis because it demonstrates the success of the uncertain type in a situation where ground truth data is available for evaluation.

The previous case study demonstrated the effectiveness of the uncertain type on sensor data. It is difficult to evaluate improvement of GPS data since there is usually no ground truth data available for comparison. This section presents a case study that demonstrates how the uncertain type can improve the quality of sensor data in a case where ground truth data is available, allowing us to quantitatively evaluate the improvement.

Conway's Game of Life is a cellular automaton that operates on a two-dimensional grid of cells [Gardner, 1970]. The game is broken into generations (i.e., steps in time), and in each generation, each cell is either alive or dead. To step from one generation to the next, the game applies to each cell simultaneously the following rules that simulate various phenomena in life:

1. A live cell with 2 or 3 live neighbours remains alive (survival)
2. A live cell with less than 2 live neighbours dies (underpopulation)
3. A live cell with more than 3 live neighbours dies (overcrowding)
4. A dead cell with exactly 3 live neighbours becomes alive (reproduction)

The game begins with an initial seed pattern, where each cell is either alive or dead.

Despite simple rules, the Game of Life has complex and interesting dynamics, including patterns that remain constant, repeating patterns, and patterns that can replicate themselves. Perhaps most surprisingly, the Game of Life is Turing complete [Berlekamp et al., 2004].

I recast the Game of Life as SensorLife, a game featuring sensor data, by noticing that we can view each cell as having eight digital sensors, one for each of its neighbours. This sensor data will form the basis of this case study.

Identifying the distribution

The usual implementation of the Game of Life does not have sensors and therefore has no sensor noise. In SensorLife, each cell has eight binary sensors, one for each neighbour, that return either 1 (if the neighbour is alive) or 0 (if the neighbour is dead). For this case study, we will artificially add noise to each of the sensors in SensorLife. We take each binary sensor and add to it zero-mean Gaussian noise, with a predefined standard deviation σ. We define three levels of noise, low, medium, and high, with standard deviations $\sigma_{Low} = 0.01$, $\sigma_{Medium} = 0.05$, and $\sigma_{High} = 0.1$, respectively. Because each sensor has noise, and we know its distribution, we wrap each sensor in the uncertain type, so that the sensor now has signature

```
Uncertain<double> SenseNeighbour(Cell me, Cell neighbour);
```

The resulting distribution is over real numbers since the reading is a binary 1 or 0 plus real-valued Gaussian noise.

Computing with distributions

To apply the rules to a cell, the Game of Life first counts the number of live neighbours of the cell, using a function:

```
int CountLiveNeighbours(Cell me);
```

In SensorLife, we can view this step as taking the sum of the sensor values of each neighbour, redefining `CountLiveNeighbours`:

```
Uncertain<double> CountLiveNeighbours(Cell me) {
    Uncertain<double> sum = new Uncertain<double>(0.0);
    foreach (Cell neighbour in me.GetNeighbours())
        sum = sum + SenseNeighbour(me, neighbour);
    return sum;
}
```

Because the errors in each sensor are independent of each other, this code suffices to define the distribution of the final result.

Asking the right questions

After counting the neighbours, the Game of Life applies the four rules by a series of four conditionals:

```
bool IsAlive = IsCellAlive(me);
int  NumLive = CountLiveNeighbours(me);
if (IsAlive && NumLive < 2)
    SetIsAlive(me, false);
else if (IsAlive && 2 <= NumLive && NumLive <= 3)
    SetIsAlive(me, true);
else if (IsAlive && NumLive > 3)
    SetIsAlive(me, false);
else if (!IsAlive && NumLive == 3)
    SetIsAlive(me, true);
```

For SensorLife, `CountLiveNeighbours` now returns a distribution $Uncertain\langle Double\rangle$, and so each of the four conditionals involving `NumLive` is now a hypothesis test on `NumLive.E()`, as described in Section 5.3. For example, the first rule is now:

```
if (IsAlive && NumLive.E() < 2.0)
    SetIsAlive(me, false);
```

This conditional implicitly performs a hypothesis test at the 95% significance level.

For our evaluation below, we compute ground truth data by also calculating the state of each cell without using the uncertain type; that is, we run SensorLife and the original Game of Life against the same state and compare their results. We also vary the significance level of the hypothesis tests to demonstrate the trade-off that hypothesis testing embodies.

Evaluation of SensorLife

We compare SensorLife to the original Game of Life across a number of configurations. Each configuration consists of a confidence level for the hypothesis tests, and the amount of noise added to the sensor (low, medium, or high, as described above).

For each configuration, we execute SensorLife 30 times. Each execution starts with a newly randomised seed pattern on a 20 × 20 board, continues for 25 generations, and therefore evaluates 10 000 cells in total.

Figure 6.4 plots the number of incorrect decisions made by SensorLife as a function of the confidence level for hypothesis tests, at each of the noise configurations (BayesLife will be introduced below). The *naive error* line indicates the number of incorrect decisions made at the low noise level *without* the uncertain type, that is, if the program just naively takes the noisy sensor values and uses them. The SensorLife bars show the number of incorrect decisions made *with* the uncertain type at each noise level.

As the confidence level increases, these results show that the accuracy of SensorLife increases too. Even at very low confidence levels, SensorLife improves on a naive approach. Insisting on high confidence reduces error rates by 25% to 99% depending on the noise level.

Mitigating noise in this way has a cost, because the uncertain type must draw more samples to satisfy higher confidence levels for each cell update. Figure 6.5 shows the relationship between performance (on the x-axis) and accuracy (on the y-axis) at each noise level. Each point on a noise level's line is a different confidence level. These results show that if the program insists on higher accuracy by choosing a higher confidence level, the program must do more work to satisfy the conditionals. These results also show that higher noise levels create more work for the uncertain type, since as the noise level increases, the performance-accuracy frontier shifts outwards.

Adding domain knowledge

SensorLife eliminates a significant amount of sensor noise, especially at lower noise levels. To improve these results even further, an expert programmer can further model the noise distribution as domain knowledge. So far, the program has not demonstrated any knowledge of the fact that the underlying true state of a cell is binary, either 1 or 0, and that it is the sensor noise that creates real-valued readings. This represents domain knowledge which can be encoded as a prior distribution.

Let v be the raw, noisy sensor reading, and let r be the underlying true state of the cell being sensed, which of course we do not know. Then we know that $v = r + \mathrm{Normal}(0, \sigma)$ for some standard deviation σ that we also know. Because we

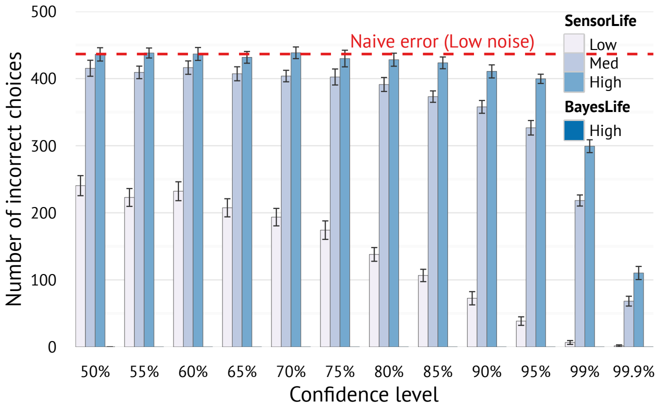


Figure 6.4: The uncertain type reduces incorrect decisions in SensorLife

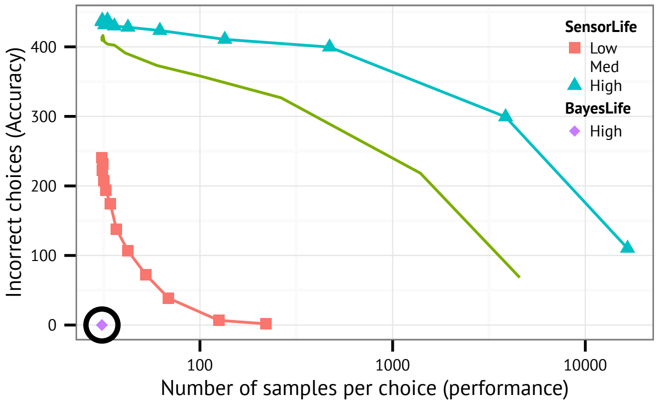


Figure 6.5: The confidence level trades performance for accuracy in SensorLife

know that r is binary, we essentially have two hypotheses for r: H_0 says that $r = 0$, and H_1 that $r = 1$. Bayes' theorem allows us to calculate a posterior probability

$$\Pr[H_0|v] = \frac{\Pr[v|H_0] \cdot \Pr[H_0]}{\Pr[v]}$$

and similarly for $\Pr[H_1|v]$. To improve an estimate, we will calculate these posteriors and fix the result to either 0 or 1 depending on which probability is higher.

To use Bayes' theorem for these posteriors requires us to define (i) the prior likelihoods of H_0 and H_1, and (ii) a likelihood function to calculate $\Pr[v|H_0]$ and $\Pr[v|H_1]$. Note that because will only be comparing the posteriors, and they share a common denominator, we need not calculate $\Pr[v]$.

We assume for this example that $\Pr[H_0] = \Pr[H_1] = 0.5$, that is, that we have no prior knowledge about whether a cell is more likely to be alive or dead. Because we know the error model is Gaussian, we also know the likelihood functions: $\Pr[v|H_0]$ is the probability that $0 + \text{Normal}(0, \sigma) = v$, and similarly $\Pr[v|H_1]$ is the probability that $1 + \text{Normal}(0, \sigma) = v$. Computing these values only requires us to evaluate the Gaussian distribution's density function.

We can therefore calculate the posterior probabilities $\Pr[H_0|v]$ and $\Pr[H_1|v]$. The output of the sensor will be 1 if H_1 is more likely (i.e., $\Pr[H_1|v]$ is larger), and 0 otherwise. We call this updated version of the program BayesLife.

Evaluating BayesLife

We evaluate BayesLife only on the high noise level ($\sigma_{High} = 0.1$) from SensorLife. Figure 6.4 shows the performance of BayesLife. In fact, BayesLife makes no incorrect choices even at the lowest confidence levels, and so there is no bar in the figure.

BayesLife achieves these results without the performance impact that SensorLife suffered at high confidence levels. In Figure 6.5, the single point for BayesLife is circled in the lower-left corner. BayesLife requires only a handful of samples to eliminate virtually all error in the noisy Game of Life, because it exploits domain knowledge.

Summary

This case study added random noise to sensor readings from the Game of Life to illustrate quantitatively that the uncertain type can help correct errors in noisy data. SensorLife demonstrates how a non-expert programmer would use the uncertain type, consuming a distribution from an API and making only minimal changes to their program. In this case, the uncertain type significantly reduced the incidence of errors, at a small performance cost. BayesLife demonstrates how an expert programmer can bring domain knowledge to bear on a problem using the uncertain type. The expert programmer used the fact that the underlying sensor value was binary to eliminate the sensor noise, making no incorrect decisions in this case. Although synthetic, SensorLife and BayesLife demonstrate concretely that with the uncertain type, programmers can improve the correctness of programs that use uncertain data.

6.3 Approximate computing

Section 2.5.1 describes *approximate computing*, an emerging field of research that explores trading accuracy for performance or efficiency. Many programs do not require the computational correctness guaranteed by the underlying hardware, and can cope with some amount of error in some calculations. Approximate computing exploits this property to improve performance and energy efficiency.

Machine learning approximates functions by extrapolating from a set of training inputs and outputs. Often a machine learning algorithm results in a prediction function that is much cheaper to evaluate than the original function, but this performance comes at the cost of accuracy. This trade-off makes machine learning a good fit for approximate computing, especially if the prediction function is generic enough to be executed by specialised machine learning hardware. This case study builds on *Parrot*, a compiler transformation that trains a neural network to approximate a segment of code [Esmaeilzadeh et al., 2012b]. The neural network is specified by a vector of weights and is executed on a specialised neural network hardware unit. If the segment of code being approximated is sufficiently hot, and the program can tolerate approximate results, this transformation delivers significant performance gains.

The downside of such a compiler transformation is that there is no way for programmers to reason about the error the transformation introduces into their computations. Common machine learning practice trains a single maximum likelihood model and uses it to predict a single output for each input, ignoring the uncertainty in the prediction. This case study introduces a modified version of the Parrot transformation, called Parakeet, that uses the uncertain type to reason about the error the approximation induces. Parakeet expresses and reasons about uncertainty in approximate computing. It therefore combines correctness and approximate computation, to determine if the approximation is accurate enough for the programmer's specific needs.

Identifying the distribution

Parakeet uses a Bayesian formulation of a neural network to define the error distribution it returns. A neural network is a function $y(x, \mathbf{w})$ that approximates the output of a target function $f(x)$ by evaluating a fixed network of functions, with coefficients combining the functions given by the weight vector $\mathbf{w}$. The common formulation of a neural network learns a single weight vector $\mathbf{w}$ that is then used for prediction. On the other hand, A Bayesian formulation learns a posterior predictive distribution $p(y|x, \mathcal{D})$, a distribution over possible predictions of the value of $f(x)$ given the training data set $\mathcal{D}$.

Learning the posterior predictive distribution is intractable in general, and so we must turn to approximation. We adopt a *hybrid Monte Carlo* sampling approach, first implemented for neural networks by Neal [Neal, 1994]. The hybrid Monte Carlo sampler draws samples from the posterior distribution $p(\mathbf{w}|\mathcal{D})$ of weight vectors,

```
float Sobel(float[3][3] p) {
    float x, y, r;
    x  = p[0][0] + 2*p[1][0] + p[2][0];
    x -= p[0][2] + 2*p[1][2] + p[2][2];
    y  = p[0][0] + 2*p[0][1] + p[0][2];
    y -= p[2][0] + 2*p[2][1] + p[2][2];
    r  = sqrt(x*x + y*y);
    r  = min(0.7070, r);
    return r;
}
```

Figure 6.6: The Sobel operator calculates the gradient of image intensity

which we then use to approximate the posterior distribution as

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

by Monte Carlo integration. The intuition of the hybrid Monte Carlo sampler is that each sample from $p(\mathbf{w}|\mathcal{D})$ describes a configuration of the neural network. Those configurations that perform well on the training data will have higher values of $p(\mathbf{w}|\mathcal{D})$ and therefore are more likely to be sampled. Different configurations can give vastly different predictions on unseen data. By considering more configurations of the neural network, weighted by their performance on the training data, we get a better picture of how the prediction generalises to unseen data.

The downside of the Bayesian approach is that it is more expensive than the common single-configuration approach. The hybrid Monte Carlo sampler must evaluate multiple neural networks (one per sample), and requires hand tuning to achieve practical rejection rates. There are other approximations to the posterior predictive distribution which strike different trade-offs. A Gaussian approximation to the posterior $p(y|x, \mathcal{D})$ is trivial to sample, making its performance much closer to Parrot, but is inaccurate when the Gaussian approximation is a poor one [Bishop, 2006].

Asking the right questions

We evaluate the Parakeet transformation using the Sobel benchmark from the original evaluation of Parrot [Esmaeilzadeh et al., 2012b]. The Sobel operator $s(p)$, shown in Figure 6.6, computes the gradient of image intensity at a pixel p. It considers a 3×3 window centred on the pixel p, and uses the (greyscale) intensity of each pixel in the window to determine the gradient of intensity at p.

The Sobel operator is used for edge detection in images. If the gradient of intensity at the pixel p is high, there is a sharp change in intensity at that point. This suggests an edge exists in the image at that point. Formally, to decide whether an edge exists at pixel p, the edge detection algorithm evaluates the conditional $s(p) > \alpha$ for some detection threshold $\alpha \in [0, \sqrt{2}]$.

Esmaeilzadeh et al. showed in their evaluation that Parrot could approximate the value of $s(p)$ with average error of 3.44% [Esmaeilzadeh et al., 2012b]. But they did not consider how this error manifests in the programs where the Sobel operator is used. The uncertain type explicitly exposes this error to the program, and this case study evaluates its effects.

Evaluation

We use Parakeet to approximate the Sobel operator, using a training set of 5000 inputs and outputs. We evaluate the approximation on a test set of 500 inputs. Both training and test sets are randomly sampled from a single test image, as in the original Parrot evaluation, to simulate a real-world distribution of inputs.

The training process learns an approximate Sobel operator $\hat{s}(p)$. For each test input p, we evaluate the ground truth boolean value $s(p) > 0.1$, and evaluate the conditional $\hat{s}(p) > 0.1$ using the uncertain type. We test the two different styles of conditional that the uncertain type provides: comparing means ($\mathbb{E}[\hat{s}(p)] > 0.1$) and evidence at the 90th percentile ($\Pr[\hat{s}(p) > 0.1] > 0.9$). We evaluate both these conditionals using different confidence levels for the implicit hypothesis test they represent.

Figure 6.7 shows that in most configurations, Parakeet makes fewer mistakes than Parrot. While Parrot can approximate the Sobel operator with a low average error of 3.44%, when these approximations are used in conditionals the error becomes much higher – for the test set, 33% of the Parrot estimates resulted in an incorrect decision being made for the conditional $\hat{s}(p) > 0.1$. The uncertain type for the expected value conditional achieves the same error rate at low confidence intervals, but improves significantly as the confidence level increases. The 90th percentile conditional, which requires even stronger evidence, achieves even better results. At the highest confidence level, the 90th percentile conditional makes 45% fewer incorrect decisions than Parrot.

Figure 6.8 shows that the improved correctness of the uncertain type comes at a relatively low performance cost. For example, even the highest confidence levels for the expected value conditional require fewer than 30 samples. The 90th percentile conditional achieves error rates of 20% at most confidence levels (40% better than Parrot), costing less than 15 samples. Of course, as demonstrated repeatedly in this thesis, the confidence level allows the programmer to trade performance for accuracy, according to the needs of their specific program.

Summary

Parakeet demonstrates that the uncertain type helps improve the correctness of approximate computing. Existing work in the field tends not to consider the effect of the error on programs, beyond a general statement along the lines of "the computation is robust to error". The uncertain type allows programmers, not researchers, to be the judge of this robustness. Programmers can explicitly reason about the error in their approximations. While the results of this case study are early, they demonstrate that the uncertain type is a promising programming model for approximate computing.

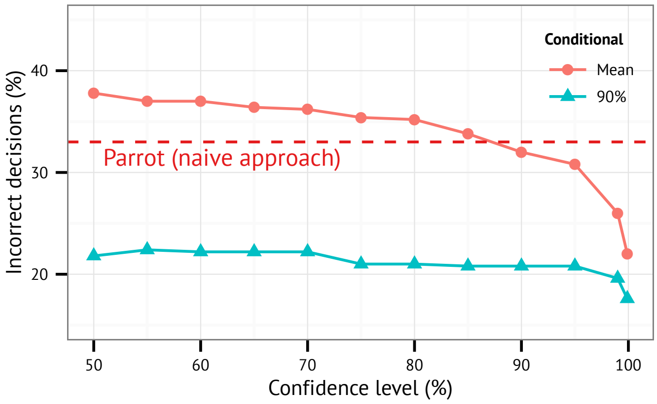


Figure 6.7: The uncertain type reduces incorrect decisions in approximate programs

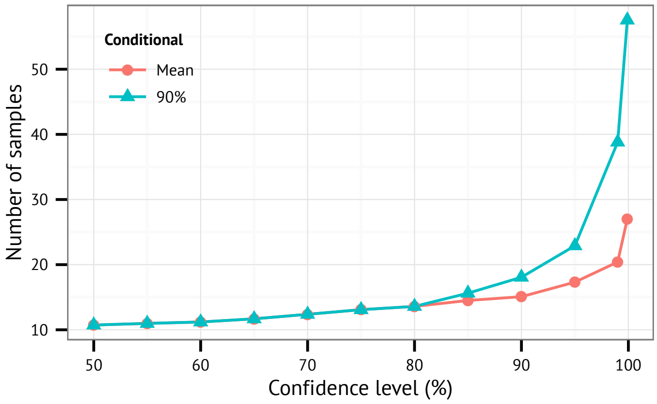


Figure 6.8: The uncertain type imposes a relatively small cost to improve correctness

6.4 Discussion

The uncertain type is an abstraction for addressing uncertainty in programs. Uncertain data occurs in a number of domains, and we have only sampled three. The case studies in this chapter choose three domains (smartphones, sensors, and approximate computing) for which the uncertain type is particularly appropriate, because these are domains where (a) expert and non-expert programmers work side by side, and (b) uncertainty is currently underappreciated. The results of these case studies demonstrate that the uncertain type is a good fit for the problems these domains face, because it bridges the gap between experts and non-experts.

The experience of using the uncertain type in these case studies exposes the potential that new abstractions may have to change how both expert and non-expert programmers think about and solve problems involving uncertainty. Preparing these case studies required explicitly considering the effects of random error throughout the code, a compulsion that is sorely lacking in existing programming models. Moreover, using the uncertain type in these case studies was in most cases intuitive and straightforward. The results demonstrate concretely that the uncertain type improved the quality of these programs. Together, these case studies demonstrate my thesis that the uncertain type helps programmers to write programs that are more concise, expressive, and correct.

Chapter 7

Conclusion

7.1 Future Work

The uncertain type brings uncertainty into the programming mainstream, suggesting a number of future lines of exploration. If programs consider uncertainty, they will be able to more robustly respond to error when it does arise. They can also reason about error over time in a principled way. There is also considerable room for further work on the semantics, implementation, and usage of the uncertain type itself. Below are four areas of future research that could be particularly fruitful.

Sensor applications. Adopting the uncertain type means that programmers make explicit the effect of uncertainty on their programs. Without the right abstraction, the robustness of a program to random error is an implicit property without any clear manifestation. For example, most work on approximate computing involves programs that are amenable to approximation, without any description of how this property is identified. Some recent work attempts to ascribe a formal definition of robustness to programs [Chaudhuri et al., 2011], but this work does not make clear *how* a program becomes robust to random error. The uncertain type makes this property explicit, because programmers must declare how variance affects computations (which is not to say that every program using the uncertain type is robust to error).

If programs reason correctly about error, we can exploit their robustness to trade accuracy for efficiency, as approximate computing does for computations. In particular, we can use lower-power sensors for measurements like GPS. The majority of applications do not require sub-metre accuracy from the GPS, and the uncertain type provides a practical way to use less accurate GPS readings while still considering the effect of the accuracy degradation. The uncertain type runtime could even be connected further into the hardware, to vary power usage depending on the required accuracy of the specific consuming computation.

A programming model for uncertainty. The case studies in Chapter 6 are small examples of the promise of the uncertain type as a programming model. The use of the uncertain type as an interface to approximate computing deserves further exploration, as it provides the missing link between the hardware and the end result that programmers otherwise must try to reason about implicitly. Potentially even more promising is applying the uncertain type to machine learning. Many popular machine learning algorithms have Bayesian formulations that provide predictive distributions rather than singular predictions. The uncertain type makes reasoning about the error in machine learning an online rather than offline exercise. Rather than testing a predictor on one test set and measuring its error offline, the uncertain type binds predictions to their error at run time, forcing those who consume the predictions to also consider that they are imperfect and may be wrong.

Improving estimates. By encapsulating distributions, the uncertain type unlocks the power of Bayesian inference, allowing programmers to improve the quality of an estimate using prior information. But rather than being static, this prior information can be learned over time. For example, a GPS sensor can learn a user's location history over time, and use these past results as an indicator of future behaviour. There are also more abstract sources of prior information to explore. For example, calendar information on a smartphone can be used to improve GPS fixes, by noticing that if a user has an appointment at a certain time, they are likely to be at the location of that appointment at that time. The uncertain type provides a simple, principled way to reason about prior knowledge, and the sampling function representation provides a tractable way to specify even complex distributions that have no closed form.

Optimising the uncertain type. Chapter 5 focused on performance, in order to make the uncertain type not only a theory but a practical reality. But more work is necessary to bring it closer to the performance of current abstractions, especially in the low-resource smartphone environment where efficiency is critical. One trivial optimisation is to provide a family of closed-form distributions which short-circuit the lazy evaluation and hypothesis testing semantics when they are not necessary. For example, the mean of the sum of two Gaussian distributions can be computed in constant time in closed form. There is also considerable scope to optimise the lazy evaluation infrastructure, noticing its similarities to both probabilistic graphical models (from machine learning) and lazy evaluation in other programming languages. For example, it may be possible to apply a series of compiler transformations to lazy evaluation trees, such as optimising loop-carried dependencies that otherwise would produce a deep expression tree.

7.2 Conclusion

The problems programmers are solving are increasingly ambitious and ambiguous, reasoning about data from the smallest smartphone sensor to overwhelming databases. What these data sources and the programs that compute with them have in common is that they introduce uncertainty. This problem is not just abstract; this thesis shows that programs experience three types of uncertainty bugs when they fail to consider uncertainty.

Programming languages encourage programmers to abrogate their responsibilities with respect to uncertainty by providing insufficient abstractions and tools. Most programming languages represent uncertain data in the same fashion as other data, using simple discrete types that ignore error. This incongruity drives most programmers to ignore uncertainty. Other abstractions for uncertain data also fall short in one or more of accessibility, efficiency, or expressivity. For example, probabilistic programming languages are inaccessible to the increasing audience of non-expert programmers that deal with uncertainty. Current abstractions, such as for GPS data, are not expressive enough to ask the questions that uncertainty demands, because they mistakenly abstract uncertainty away as if it were only an implementation detail. The challenge this thesis confronts is whether an abstraction for uncertainty can be suitably accessible, efficient, and expressive as to be practical for everyday programmers.

I contend that the uncertain type answers the challenge. The uncertain type, or $\mathit{Uncertain}\langle T \rangle$, is a programming language abstraction for uncertain data that focuses on an accessible interface for non-expert programmers.

The semantics of the uncertain type provide an accessible and expressive abstraction for identifying probability distributions, computing with them, asking questions using conditionals, and improving estimated data with domain knowledge. Throughout these steps the uncertain type balances correctness with ease of use, a trade-off existing work does not adequately consider. Insights into random sampling, lazy evaluation, data dependencies, and hypothesis testing work in concert to make this abstraction efficient and tractable for use in real-world programs.

Three case studies demonstrate that the uncertain type achieves its goals. The uncertain type improves the accuracy and expressiveness of smartphone GPS libraries, eliminates random noise in a simple digital sensor, and makes approximate computing more accurate and explicit in its trade-offs. The experience of these case studies exposes the potential of the uncertain type as a programming model for thinking about and solving problems involving uncertainty, especially because it compels programmers to consider the effect of error on their programs, while not requiring expert knowledge of probability theory.

Uncertainty is not a malignance to be avoided. It is an important part of many problems, and a mastery of uncertainty gives programmers the power to create programs that ask sophisticated questions about evidence and belief. Just as many other scientific fields have principled mechanisms for making decisions under uncertainty, so too should programming. The uncertain type embraces uncertainty, while recognising that most programmers are not experts in probability theory. By making uncertainty a primitive first-order type, non-expert programmers write programs that are more concise, expressive, and correct.

Bibliography

Baek, W. and Chilimbi, T. M., 2010. Green: A framework for supporting energy-conscious programming using controlled approximation. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, ON, Canada, June 5 - 10, 2010*. (cited on page 21)

Benjelloun, O.; Sarma, A. D.; Halevy, A.; and Widom, J., 2006. ULDBs: Databases with uncertainty and lineage. In *Proceedings of the 32nd International Conference on Very Large Data Bases, VLDB 2006, Seoul, Korea, September 12 - 15, 2006*. ACM. (cited on page 18)

Berlekamp, E. R.; Conway, J. H.; and Guy, R. K., 2004. *Winning Ways for Your Mathematical Plays, Volume 4*. A K Peters, Wellesley, MA. (cited on page 72)

Bishop, C. M., 2006. *Pattern Recognition and Machine Learning*. Springer, New York, NY. (cited on pages 59 and 78)

Blackburn, S. M.; Garner, R.; Hoffmann, C.; Khang, A. M.; McKinley, K. S.; Bentzur, R.; Diwan, A.; Feinberg, D.; Frampton, D.; Guyer, S. Z.; Hirzel, M.; Hosking, A.; Jump, M.; Lee, H.; Moss, J. E. B.; Phansalkar, A.; Stefanovic, D.; VanDrunen, T.; von Dincklage, D.; and Wiedermann, B., 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications, OOPSLA 2006, Portland, OR, USA, October 22 - 26, 2006*. (cited on page 19)

Borgström, J.; Gordon, A. D.; Greenberg, M.; Margetson, J.; and Van Gael, J., 2011. Measure transformer semantics for bayesian machine learning. In *Proceedings of the 20th European Conference on Programming Languages and Systems, ESOP 2011, Saarbrücken, Germany, March 26 - April 3, 2011*. Springer-Verlag. (cited on page 2)

Bornholt, J.; Mytkowicz, T.; and McKinley, K. S., 2013. Uncertain<T>: A First-Order Type for Uncertain Data. Technical Report MSR-TR-2013-46, Microsoft Research. (cited on pages vii and 72)

Carbin, M.; Misailovic, S.; and Rinard, M. C., 2013. Verifying quantitative reliability of programs that execute on unreliable hardware. In *Proceedings of the 28th Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications, OOPSLA 2013, Indianapolis, IN, USA, October 29 - 31, 2013*. (cited on page 21)

Chaganty, A. T.; Nori, A. V.; and Rajamani, S. K., 2013. Efficiently sampling probabilistic programs via program analysis. In *Proceedings of the 16th International Conference on Artificial Intelligence and Statistics, AISTATS 2013, Scottsdale, AZ, USA, April 29 - May 1, 2013*. JMLR. (cited on pages 2, 12, 13, and 16)

Chakrapani, L. N.; Akgul, B. E. S.; Cheemalavagu, S.; Korkmaz, P.; Palem, K. V.; and Seshasayee, B., 2006. Ultra-efficient (embedded) SOC architectures based on probabilistic CMOS (PCMOS) technology. In *Proceedings of the Conference on Design, Automation and Test in Europe, DATE 2006, Munich, Germany, March 6 - 10, 2006*. (cited on page 22)

Chaudhuri, S.; Gulwani, S.; Lublinerman, R.; and Navidpour, S., 2011. Proving programs robust. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of software engineering, FSE 2011, Szeged, Hungary, September 5 - 9, 2011*. (cited on page 83)

Dalvi, N. and Suciu, D., 2007. Management of probabilistic data: Foundations and challenges. In *Proceedings of the 26th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS 2007, Beijing, China, June 11 - 13, 2007*. (cited on page 18)

Erwig, M. and Kollmansberger, S., 2006. Probabilistic functional programming in Haskell. *Journal of Functional Programming*, *16*(1), (2006), 21–34. (cited on pages 10 and 11)

Esmaeilzadeh, H.; Sampson, A.; Ceze, L.; and Burger, D., 2012a. Architecture support for disciplined approximate programming. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012*. ACM. (cited on page 22)

Esmaeilzadeh, H.; Sampson, A.; Ceze, L.; and Burger, D., 2012b. Neural acceleration for general-purpose approximate programs. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2012, Vancouver, BC, Canada, December 1 - 5, 2012*. IEEE Computer Society. (cited on pages 21, 77, 78, and 79)

Gardner, M., 1970. Mathematical Games: The fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, *223*, (1970), 120–133. (cited on page 72)

Gartner, 2013. Gartner Says Smartphone Sales Grew 46.5 Percent in Second Quarter of 2013 and Exceeded Feature Phone Sales for First Time. Press Release. Accessed August 2013. URL http://www.gartner.com/newsroom/id/2573415. (cited on page 22)

Georges, A.; Buytaert, D.; and Eeckhout, L., 2007. Statistically Rigorous Java Performance Evaluation. In *Proceedings of the 22nd annual ACM SIGPLAN Conference* *on Object-oriented Programming Systems, Languages, and Applications, OOPSLA 2007, Montreal, QC, Canada, October 21 - 25, 2007*. (cited on page 19)

Gilks, W. R.; Thomas, A.; and Spiegelhalter, D. J., 1994. A Language and Program for Complex Bayesian Modelling. *Journal of the Royal Statistical Society. Series D (The Statistician)*, *43*(1), (1994), 169–177. (cited on pages 2 and 13)

Giry, M., 1982. A categorical approach to probability theory. In *Categorical Aspects of Topology and Analysis* (Ed. B. Banaschewski), vol. 915 of *Lecture Notes in Mathematics*, 68–85. Springer Berlin Heidelberg. (cited on page 10)

Glen, A. G.; Evans, D. L.; and Leemis, L. M., 2001. APPL: A Probability Programming Language. *The American Statistician*, *55*(2), (2001), 156–166. (cited on pages 14, 17, and 58)

Goodman, N. D.; Mansinghka, V. K.; Roy, D. M.; Bonawitz, K.; and Tenenbaum, J. B., 2008. Church: a language for generative models. In *Proceedings of the 24th Conference in Uncertainty in Artificial Intelligence, UAI 2008, Helsinki, Finland, July 9 - 12, 2008*. (cited on pages 2, 12, and 13)

Hastings, W. K., 1970. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, *57*(1), (1970), 97–109. (cited on page 2)

Jaroszewicz, S. and Korzeń, M., 2012. Arithmetic operations on independent random variables: A numerical approach. *SIAM Journal on Scientific Computing*, *34*, (2012), A1241–A1265. (cited on pages 14 and 17)

Kidd, E., 2007. Bayes' rule in Haskell, or why drug tests don't work. Accessed August 2013. URL http://www.randomhacks.net/articles/2007/02/22/bayes-rule-and-drug-tests. (cited on page 11)

Kiselyov, O. and Shan, C.-C., 2009. Embedded probabilistic programming. In *Proceedings of the IFIP TC 2 Working Conference on Domain-Specific Languages, DSL 2009, Oxford, England, July 15 - 17, 2009*. Springer-Verlag. (cited on page 14)

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

Minka, T.; Winn, J.; Guiver, J.; and Knowles, D., 2012. Infer.NET 2.5. Microsoft Research Cambridge. Accessed August 2013. URL http://research.microsoft.com/infernet. (cited on pages 2, 12, and 13)

Moore, R. E., 1966. *Interval analysis*. Prentice-Hall, Englewood Cliffs, NJ, USA. (cited on page 20)

Narayanan, S.; Sartori, J.; Kumar, R.; and Jones, D. L., 2010. Scalable stochastic processors. In *Proceedings of the Conference on Design, Automation and Test in Europe, DATE 2010, Dresden, Germany, March 8 - 12, 2010*. IEEE. (cited on page 22)

Neal, R. M., 1994. *Bayesian learning for neural networks*. Ph.D. thesis, University of Toronto. (cited on page 77)

Newson, P. and Krumm, J., 2009. Hidden Markov map matching through noise and sparseness. In *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, GIS 2009, Seattle, WA, USA, November 4 - 6, 2009*. (cited on page 18)

Park, S.; Pfenning, F.; and Thrun, S., 2005. A probabilistic language based on sampling functions. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, CA, USA, January 12 - 14, 2005*. (cited on pages 10, 13, 17, 35, and 43)

Ramsey, N. and Pfeffer, A., 2002. Stochastic lambda calculus and monads of probability distributions. In *Proceedings of the 29th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2002, Portland, OR, USA, January 16 - 18, 2002*. (cited on pages 2 and 10)

Saheb-Djahromi, N., 1978. Probabilistic LCF. In *Mathematical Foundations of Computer Science 1978* (Ed. J. Winkowski), vol. 64 of *Lecture Notes in Computer Science*, 442–451. Springer Berlin Heidelberg. (cited on page 13)

Sampson, A.; Dietl, W.; Fortuna, E.; Gnanapragasam, D.; Ceze, L.; and Grossman, D., 2011. EnerJ: Approximate data types for safe and general low-power computation. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4 - 8, 2011*. (cited on page 21)

Sankaranarayanan, S.; Chakarov, A.; and Gulwani, S., 2013. Static analysis for probabilistic programs: Inferring whole program properties from finitely many paths. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2013, June 17 - 19, 2013*. (cited on page 16)

Schwarz, J.; Hudson, S.; Mankoff, J.; and Wilson, A. D., 2010. A framework for robust and flexible handling of inputs with uncertainty. In *Proceedings of the 23nd annual ACM Symposium on User Interface Software and Technology, UIST 2010, New York, NY, USA, October 3 - 6, 2010*. (cited on page 19)

Schwarz, J.; Mankoff, J.; and Hudson, S., 2011. Monte carlo methods for managing interactive state, action and feedback under uncertainty. In *Proceedings of the 24th annual ACM Symposium on User Interface Software and Technology, UIST 2011, Santa Barbara, CA, USA, October 16 - 19, 2011*. (cited on page 19)

Thompson, R., 1998. Global positioning system: The mathematics of GPS receivers. *Mathematics Magazine*, *71*(4), (1998), 260–269. (cited on pages 18 and 25)

Thrun, S., 2000. Towards programming tools for robots that integrate probabilistic computation and learning. In *Proceedings of the 2000 IEEE International Conference on Robotics and Automation, ICRA 2000, San Francisco, CA, USA, April 24 - 28, 2000*. (cited on page 18)

Topsøe, F., 1970. On the Glivenko-Cantelli theorem. *Zeitschrift für Wahrscheinlichkeitstheorie und Verwandte Gebiete*, *14*, (1970), 239–250. (cited on page 43)

van Diggelen, F., 2007. GNSS Accuracy: Lies, Damn Lies, and Statistics. *GPS World*, *18*(1), (2007), 26–32. (cited on page 38)

Whitehead, J., 1999. A unified theory for sequential clinical trials. *Statistics in Medicine*, *18*(17-18). (cited on page 63)

Zadeh, L. A., 1965. Fuzzy sets. *Information and Control*, *8*, (1965), 338–353. (cited on page 20)