The random approximate values over algebraic types

Alexander Towell
atowell@siue.edu

Abstract

We define the semantics of *random approximate values* over algebraic data types.

Contents

1 Introduction

The primary mechanism by which a value is an *approximation* is given by the random approximate map model. If a first-order random approximate map of type $X \overset{\varepsilon}{\underset{\omega}{\mapsto}} Y$ takes in an exact value X then it maps to a random approximate value y_{ε}^{ω} of type Y.

We denote that the distribution of values over a type X take on random approximations with $X^{\pm}$. The type is still the same, only the *values* are different with respect to some objective standard, e.g., if f maps a value a to b, then ...

A *type* is a set and the elements of the set are called the *values* of the type.

These *values* are approximate values if, according to some objective function they should be x but take on a range of possible values according to the random approximate value model.

An *abstract data type* is a type and a set of operations on values of the type. For example, the *integer* abstract data type is defined by the set of integers and standard operations like addition and subtraction. A *data structure* is a particular way of organizing data and may implement one or more abstract data types.

2 Random approximate algebraic data types

Suppose we have a function $\mathrm{g}\colon \mathcal{B} \mapsto \mathcal{B}$.

The random approximate value type $\mathcal{B}_\varepsilon^\omega$ is a *monad*. If we provide it as input to g, we git a random approximate value type as output.

That is, the function g is lifted to the function type $\mathcal{B}_\varepsilon^\omega \mapsto \mathcal{B}^\pm$, denoted by $\mathrm{g}^\pm$. The output $\mathcal{B}^\pm$ is a random variable, in particular, it is a first-order random approximate value type.

To compute its false positive and false negative rates, we simply have to compute the following.

Suppose it is given the true value is 0. Then, by definition, $\mathcal{B}_\varepsilon^\omega$ realizes 0 with probability $1-\omega$ and 1 with probability ω. Since g is a function of a random variable, it is also a random variable (unless it is constant). So, the output is $\mathrm{g}(0)$ with probability $1-\omega$ and $\mathrm{g}(1)$ with probability ω. That is, it has a conditional probability mass function given by

$$\mathrm{p}(b|0) = \begin{cases} 1-\omega & \text{if } b = \mathrm{g}(0) \\ \omega & \text{if } b = \mathrm{g}(1)\,. \end{cases} \tag{1}$$

Alternatively, assume it is given true value is 1. Then, by definition, $\mathcal{B}_\varepsilon^\omega$ realizes 0 with probability ε and 1 with probability $1-\varepsilon$ and therefore the output is a random variable O that models $\mathcal{B}^\pm$ that realizes $\mathrm{g}(0)$ with probability $1-\omega$ and $\mathrm{g}(1)$ with probability ω with the conditional probability mass function given by

$$\mathrm{p}_\mathrm{O}(b|1) = \begin{cases} \varepsilon & \text{if } b = \mathrm{g}(0) \\ 1-\varepsilon & \text{if } b = \mathrm{g}(1)\,. \end{cases} \tag{2}$$

The random approximate map is the *exponential type*.

Other types may be generated from this type.

A *type* is a set and the elements of the set are called the *values* of the type. In programming languages, *composite* types are typically composed in two ways, the *sum type* and the *product type*. The product type is a Cartesian product of types (sets).

For instance, a product type may the Cartesian product of integer and Boolean types, $\mathbb{Z} \times \{0,1\}$. In the C++ family of programming language, this product type may be represented by *pair<int,bool>*.

Since it may be inconvenient to refer to member types by their respective tuple indices, programming languages typically allow the indices to be *labeled*. In C++, keywords like *struct* and *class* are commonly used to provide named product types.

A data type may be thought of as a product type with *invariants*, or *constraints* on which values member types may be assigned. Thus, it may be thought of as a *relation* or *correspondence* over the product type. In many cases, this may make approximate relations unsuitable for representing data types, e.g., a tuple that violates one or more of the invariants may be generated from an objective value of the data type.

However, if an approximate relation does violate the invariants, it can simply be considered invalid. In this case, we may pose questions like, what is the probability that an approximate data type generates an invalid result?

$$\mathrm{P}\left[\mathsf{X}_1{}^\pm\right] \tag{3}$$

An *abstract data type* is a type and a set of operations on values of the type. For example, the *integer* abstract data type is defined by the set of integers and standard operations like addition and subtraction. A *data structure* is a particular way of organizing data and may implement one or more abstract data types.

3 Primitive approximate *values*

Here are a basic list of primitive algebraic data types and operations on types.

These are *first-order* approximate types since t

3.1 Void type

The most primitive type is the *empty set* type, denoted by Void. There are no elements in the empty set and therefore it is not possible to construct values of this type. There is only one function in the set $\mathsf{Void} \mapsto \mathsf{X}$, known as the *absurd* function since it can never be invoked. Since Void has no values, $\mathsf{Void}^{\pm}$ is equivalent to Void.

$\mathsf{Void}^{\pm}$ is necessary to complete the algebra of algebraic data types, but serves only a theoretical purpose.

3.2 Unit type

The *unit* type, denoted by Unit, is isomorphic to any set with only a single element, i.e., any *singleton set*. The set $\mathsf{Unit} \mapsto \mathsf{X}$ has a cardinality $|\mathsf{X}|$ and the set $\mathsf{X} \mapsto \mathsf{Unit}$ has a cardinality 1. If we are talking about *partial* functions, then $\mathsf{X} \twoheadrightarrow \mathsf{Unit}$ has cardinality 2^{X}.

The set $\mathsf{Unit} \mapsto \mathsf{Unit}$ has a cardinality of 1, which is the identity function $\mathtt{id}\colon \mathsf{Unit} \mapsto \mathsf{Unit}$.

The approximate unit type $\mathsf{Unit}^{\pm}$ is necessary to complete the algebra, but given a void Unit type, similar to the Void type there is no uncertainty about its value, i.e., $\mathsf{Unit}^{\pm}$ is equivalent to Unit.

In addition, if there is some function $\mathrm{f}\colon \mathsf{Unit} \mapsto \mathsf{X}$ its approximate analog is $\mathrm{f}^{\pm}\colon \mathsf{Unit} \mapsto \mathsf{X}$, which models an *approximate* constant.

3.3 Sum types

A sum type $X + Y$ is the disjoint union of types X and Y. The *first-order* approximate sum type $(X + Y)^{\pm}$ is an approximate of the type $X + Y$.

The *higher-order* sum type is different, e.g., $X^{\pm} + Y^{\pm}$ is a higher order sum type, as is $X^{\pm} + Y$ and $\left(X^{\pm} + Y^{\pm}\right)^{\pm}$ is even a higher order.

If we replace X and Y by $X^{\pm}$ and $Y^{\pm}$, we have a sum type $X^{\pm} + Y^{\pm}$.

Values of these types are naturally constructed from approximate maps that map to the type $X + Y$.

3.4 Product types

A product type $X \times Y$ is the Cartesian product of types X and Y. If we replace X and Y by $X^{\pm}$ and $Y^{\pm}$, we have a product type $X^{\pm} \times Y^{\pm}$.

Values of these types are naturally constructed from approximate maps that map to the type $X + Y$ and form a random approximate set over the type $X \times Y$.

3.5 Exponential types (functions)

These have already been discussed. Random approximate maps are the same thing as random approximate exponential types.

When we generate a set of approximate value types and a set of approximate functions over those value types and compose them together to generate a *program*, we may consider this composition to be an *approximate* program.

TODO: A *partial function* is not defined on entire domain. We allow elements not in the preimage, i.e., not defined by the partial function, to either correctly map to *nothing* or, map to some other element in the codomain. The *false mapping rate* ε_y for element y may be specified explicitly, i.e.,

$$\mathrm{P}\big[\mathrm{f}(x) = y \,\big|\, x \notin \mathtt{dom}(\mathrm{f})\big] = \varepsilon_y \,, \tag{4}$$

and the *total* false mapping rate is

$$\sum_{y \in \mathtt{codom}(\mathrm{f})} \mathrm{P}\big[\mathrm{f}(x) = y \,\big|\, x \notin \mathtt{dom}(\mathrm{f})\big] = \varepsilon \,, \tag{5}$$

by the fact that each outcome is mutually exclusive (a function f maps to only one element). Alternatively,

$$\sum_{y \in \mathtt{codom}(\mathrm{f})} \mathrm{P}\big[\mathrm{f}(x) \neq y \,\big|\, x \notin \mathtt{dom}(\mathrm{f})\big] = 1 - \varepsilon = \eta \,. \tag{6}$$

Example 1 *Suppose we have a predicate function* $\mathrm{f} \colon \mathsf{X} \mapsto \mathsf{Bool}$*, i.e., set indicator. Then, we may construct a partial function from* f *with an* approximate map *over those elements that return* **true***. If we specify a false mapping rate* $\varepsilon_{\mathbf{false}}$*, this predicate is isomorphic to an approximate set over* X *with a false positive rate* $\varepsilon = \varepsilon_{\mathbf{false}}$*.*

An optimal approximate map obtains the information-theoretic lower-bound on the expected space complexity, $-1.44 \log_2 \varepsilon$ *bits per positive element.*

While a random approximate set is in practice simpler to implement, theoretically an optimal approximate map is both fully generalized (for any function over any domain and codomain) and obtains the same space efficiency.

The obfuscated approximate map has the same characteristics, and the optimal implementation is a black box that is able to increase obfuscation power for less space efficiency. WORK THIS OUT EXACTly, and move all of this out of the example env of course.

Remark. *NOTE: Clarify: we call a value an approximate value when it should be* x *but there is a probability it is something else due to the approximate map or some other noisy process.* △

Most approximate maps are black boxes. They introduce an approximation error, and in addition, the inputs may also have an approximation error. At that point, the values being mapped to are higher-order random approximate values.

4 Random approximate Boolean algebra

A general purpose The primitive operations in the Boolean algebra, `and`,`or`, and `negate`,

The approximate value type $\mathsf{Bool}^{\pm}$ discussed in **??** can be composed with algebraic types like the product type to construct any other value type.

We consider a generalization of this Boolean algebra given by six-tuple

$$\left(\overset{+}{-}\mathcal{B}^n, \mathtt{and}^{\pm}, \mathtt{or}^{\pm}, \mathtt{negate}^{\pm}, \overset{+}{-}1^n, \overset{+}{-}0^n \right) , \tag{7}$$

where the operators are *bit-wise* operators.

The values of n bits are *isomorphic* to any value type that has a cardinality of 2^n and as a Boolean algebra. For instance, we could implement approximate sets with a complete implementation of set-theoretic operations on them over any universe of n elements.

An exponential type $X \mapsto Y$ is the set of functions from domain X to codomain Y. If we replace X and Y by $X^{\pm}$ and $Y^{\pm}$, we have an approximate exponential type $X^{\pm} \mapsto Y^{\pm}$, e.g., if $\mathrm{f}\colon X \mapsto Y$, then an approximate representation of f is $\mathrm{f}^{\pm}\colon X^{\pm} \mapsto Y^{\pm}$.

TODO: make an approximate value monad! Carry the approximation error information, make it a simple wrapper with some additional info.

If it is not important that X or Y be themselves oblivious types, then we have the *represntation* of the functions as oblivious, but the inputs and outputs can be *plain*.

NOTE: this is the case for many things not just exponential types. Still need to grapple with this, maybe still dealing with the approximation over elements rather than the approximation of universe thing.

Maps, also known as *partial functions*, are *rules* that map inputs to outputs. Let $\mathrm{f}\colon \mathsf{X} \mapsto \mathsf{Y}$ be a partial function that maps inputs from the domain X to outputs from the codomain Y.

There are three *orthogonal* ways in which f may leak information.

Let the *computational basis* (a minimal set of functions) for values of type X be denoted by the overload set F, where *any* other function that depends on X is some composition of the elements of F and elements from other dependent computational bases.

As a function of X, f depends on a subset L of F. If we *substitute* X by some object type that *models* X, to be compatible with f, at minimum it must overload the set of functions in L.

Consider a partial function $\mathrm{f}\colon X_1 \times X_2 \cdots \times X_n \mapsto Y_1 \times \cdots \times Y_m$ and suppose we replace X_1 by $X_1^{\pm}$, an approximate value type. Then, we denote this function by $\mathrm{f}^{\pm}\colon {X_1}^{\pm} \times X_2 \cdots \times X_n \mapsto {Y_1}^{\pm} \times \cdots \times {Y_m}^{\pm}$.

There are two approaches to this.

5 C++ implementation

In programming languages, *composite* types are typically composed in two ways, the *sum type* and the *product type*. The product type is a Cartesian product of types (sets).

For instance, a product type may the Cartesian product of integer and Boolean types, $\mathbb{Z} \times \{0, 1\}$. In the C++ family of programming language, this product type may be represented by *pair<int,bool>*.

Since it may be inconvenient to refer to member types by their respective tuple indices, programming languages typically allow the indices to be *labeled*. In C++, keywords like *struct* and *class* are commonly used to provide named product types.

A data type may be thought of as a product type with *invariants*, or *constraints* on which values member types may be assigned. Thus, it may be thought of as a *relation* or *correspondence* over the product type. In many cases, this may make approximate relations unsuitable for representing data types, e.g., a tuple that violates one or more of the invariants may be generated from an objective value of the data type.

However, if an approximate relation does violate the invariants, it can simply be considered invalid. In this case, we may pose questions like, what is the probability that an approximate data type generates an invalid result?

Viewing a *type* as a set, most programming languages have primitive types like *integers*, *Booleans*, and *characters*. In C++, these are respectively denoted by `int`, `bool`, and `char` with cardinalities given respectively by 2^{32}, 2^8, and 2^1. Any type needs one or more *value constructors* to construct

objects that model values in that type. For instance, in C++ the value that denotes the Boolean value of *true* is constructed with the syntax `true`.

The *unit* type is special singleton set with a single value, i.e., a cardinality of 2^0. In C++, the confusing notation of `void` denotes the unit type (and the single value).

Remark. *The* absurd *type is a special type with zero values, i.e., the* empty set. *Since there are no values in the absurd type, no values of this type can be constructed. There is no primitive absurd type in C++.* △

A *product type* is the n-fold Cartesian product of zero or more types where the zero-th product is the unit type. For instance, in C++, `struct { char y, bool z }` is a *named* product type and `tuple<char,bool>` is the *unamed* counterpart, where both are product types `char` × `bool`. The cardinality of this product type is $2^8 \cdot 2^1 = 512$. One way a particular value of this product type may be constructed is `tuple<char,bool>('a', true)`.

The values of a sum type are typically grouped into several classes, called variants. The set of all possible values of a sum type is the disjoint union of the sets of all possible values of its variants. For instance, in C++, `variant<char,bool>` is the sum type `char` + `bool`, which has a cardinality of $2^8 + 2^1 = 258$. One way a particular value of this sum type may be constructed is `variant<char,bool>('1', true)`. A particularly useful type in C++ is `optional<X>`, which conceptually models the sum type `X` + `void` where `void` denotes the variant "not a value of type `X`."

Remark. *This is not valid C++ syntax, even though the unit type value should be first-class.* △

The cardinality of the `optional<X>` is the cardinality of `X` plus 1. We label the value in this singleton *nothing* and may test whether a particular value is either *nothing* or alternatively a value in `X`.

Exponential types are *functions*. In C++, `[](tuple<char,bool>) -> bool` is the set of functions `char` × `bool` ↦ `bool`, which has a cardinality of 2^{512}. Usually, a more convenient syntax is used, like `[](char,bool) -> bool`. The constant `true` function of the exponential type `char` × `bool` ↦ `bool` may be constructed with the definition `[](char,bool) -> bool { return true; }`.

Remark. *The exponential type `[](X x) -> void` is of little practical value and, in C++, usually denotes a* procedure *that causes* side-effects *like writing to IO.* △

Recall that any subset of a set corresponds to a *relation*. Types are *subsets* of the algebraic types where subsets are defined by *invariants*. We may compose primitive types to specify a variety of *compound* types.

Example 2 Rationals *may be implemented as a product type of two integers,*

```
using Rational = tuple<int,int>,
```

where the first and last elements of the tuple represent the numerator and denominator respectively. If the invariant is that the denominator is not 0*, then a* value constructor *`rational` : `int` × `int` ↦ `optional<Rational>` that takes a numerator and denominator and outputs either a rational or, if the invariants are violated,* nothing, *is given by* `??`.[1]

```
optional<Rational> rational(int num, int denom)
{
```

[1] Alternatively, the value constructor can be a *partial function* `rational` : `int` × `int` ↠ `Rational` that is undefined for input $\langle x, 0 \rangle$ for any $x \in$ `int`.

```
        if (denom == 0)
        return {}; // Return the value that denotes nothing.
        return tuple<int,int>(num, denom);
}.
```

The expected operators on rationals, like addition `operator+(Rational,Rational) -> Rational`*, may be implemented so that* `Rational` *models the* concept *of rationals.*

The `Rational` *type is a subset of the product type* `tuple<int,int>` *and is thus a* binary relation *on* $\mathbb{Z} \times \mathbb{Z}$.

Remark. *We implemented* `Rational` *as a product type* `tuple<int,int>` *and a value constructor* `rational` *for pedagogical reasons, but generally programming structures like* class *are utilized since they facilitate important concepts like* encapsulation. △

Each of these types and operators has a corresponding random approximation, e.g., the exponential type is just a random approximate map, and the *relations* that define types are just random approximate relations with *deterministic* properties that model the invariants.

The invariants may not be easy to satisfy, and so a random approximation relation of the corresponding type may not be practical. However, when the invariants can be satisfied, we may implement *random approximate algebraic data types* of the *algebraic data types*, e.g., we can compose random approximate algebraic data types as before to construct compound random approximate data types of the corresponding compound algebraic data type.

This may not seem particularly useful, but it permits space-efficient representations and, moreover, concepts like *oblivious algebraic data types* may be based on it with some notion of closure.

5.1 Approximate value monad

Talk about lifting functions.

Suppose we have a function $\mathrm{f}\colon X \mapsto Y$.

and we apply f to an approximate value of type X, denoted by $\mathrm{a}(X)$,

$\mathrm{f}\ \mathrm{a}(\mathrm{f})\colon X \overset{\varepsilon}{\underset{\omega}{\mapsto}} Y$.

The approximate value parameterized by X, denoted by `approx<X>`, is a monad type similar to the *maybe* monad discussed previously, except significantly more complicated if *fully* implemented.[2]

The general type of the approximate value monad is simply defined as

```
template <typename X> struct approx<X> {}
```

which has a computational basis given by the following overload set.

Given an approximate value of type X, its *false positive rate* is given by

```
template <typename X>
auto fpr(approx<X> x)   { /* implementation */ },
```

its *false negative rate* is given by

```
template <typename X>
auto fnr(approx<X> x)   { /* implementation */ },
```

[2]We may choose the trivial implementation that just tags it as an approximate value and let that bit of information carry through.

its value is given by

```
template <typename X>
auto value(approx<X> x) { /* implementation */ },
```

and its conditional probability mass function is given by

```
template <typename X>
auto pmf(approx<X> x, X true_value) { /* implementation */ }.
```

Note that the false positive and false negative rates are the result of using the distance function $\mathrm{d}(a, a) = 0$ and otherwise $\mathrm{d}(a, b) = 1$, $a \neq b$, to calculate the *expected* loss given respectively the negative (where d evaluates to 1 with respect to some ground truth) and positive (where d evaluates to 0) subsets of X.

If we give it a *loss* function, we may estimate its loss.

For particular value types, we specialize this template.

```
template <> struct approx<bool>
{
        double fpr;
        double fnr;
        bool value;
}
```

which has a computational basis given by

```
auto fpr(approx<bool> x)   { return x.fpr; },
auto fnr(approx<bool> x)   { return x.fnr; },
auto value(approx<bool> x) { return x.value; }.
```

The function `fmap`: $(\texttt{bool} \mapsto \texttt{bool}) \times \texttt{approx<bool>} \mapsto \texttt{approx<bool>}$ is defined as

```
auto amap(function<bool(bool)> f, approx<bool> x)
{
        auto fnr = f(true);
        auto fpr = f(false);

        return approx<bool> { fpr, fnr, f(value(x)) };
}
```

If we compose functions to construct an *algorithm* (or program), then if some of the values (e.g., functions or Boolean values) are replaced by approximate values, the algorithm becomes (in general) an approximate algorithm and we may deduce its false positive and negative rates as we did previously through, say, function composition.

If the algorithm includes *branching*, the algorithm is still an approximate algorithm, but in order to infer the false positive and false negative rate, we must go down *all* branches, which is in general not be tractable. However, we could estimate the result using *Monte-carlo* simulation.

The approximate Boolean value is straightforward since Boolean types are the natural predicate return type. More importantly, there are only two values of this type.

5.2 Type-erasure

The fact that we are dealing with specific approximate types may be erased into some approximate type abstract data type that hides the concrete data type. It may be further erased to just a type, e.g., f^{ω}_{ε} may be erased to $f^{\pm}$ which may be erased to f. TODO: convert to types not particular elements in the type

References