Bernoulli Types: Theory and Construction

A Framework for Approximate Computation with Controlled Error

Alexander Towell
atowell@siue.edu

January 24, 2026

Abstract

We present *Bernoulli types*, a theoretical framework for reasoning about computation with controlled approximation. The key insight is distinguishing between *latent* values (true mathematical objects) and *observed* values (what our computations actually produce). This distinction, formalized through confusion matrices and error rates, provides a principled foundation for probabilistic data structures like Bloom filters, approximate algorithms like Miller-Rabin primality testing, and space-efficient representations.

We develop the theory systematically: Bernoulli booleans with false positive and negative rates, Bernoulli sets with algebraic operations that propagate errors predictably, and Bernoulli maps that approximate arbitrary functions. A central contribution is the *hash-based construction*, which reveals how a single hash function with carefully sized encoding sets can implement any Bernoulli type. This construction unifies diverse probabilistic structures under one framework and enables navigation of space-accuracy trade-offs.

We present a type system that makes approximation explicit, enabling static reasoning about error propagation. The framework provides both theoretical foundations for understanding existing probabilistic algorithms and practical guidance for designing new ones.

1 Introduction

1.1 The Ubiquity of Approximation

Every practical computation is an approximation. Floating-point arithmetic rounds real numbers. Hash tables sacrifice worst-case guarantees for average-case efficiency. Probabilistic algorithms like Miller-Rabin trade certainty for speed. Bloom filters accept false positives for dramatic space savings.

Yet our programming languages and type systems typically pretend otherwise. A function declared as `isPrime` : $\mathbb{N} \to \mathbb{B}$ suggests definitive answers, when the implementation might be probabilistic. A set membership test $x \in S$ implies exact knowledge, even when the underlying data structure accepts errors.

This paper develops *Bernoulli types*, a framework that makes approximation explicit and tractable. Rather than viewing approximation as a necessary evil, we treat it as a fundamental aspect of practical computation that deserves first-class theoretical treatment.

1.2 Latent versus Observed

The central insight is distinguishing two levels of computation:

- **Latent values**: The true mathematical objects we reason about—perfect sets, exact functions, absolute truth values.
- **Observed values**: What our computations actually produce—approximate membership tests, probabilistic function evaluations, uncertain boolean outcomes.

This distinction is not merely philosophical. It has precise mathematical content: the relationship between latent and observed values is characterized by a *confusion matrix* that specifies the probability of observing each possible output given each possible true value.

Example 1.1 (Bloom Filter)**.** A Bloom filter represents a set S (latent) through a bit array and hash functions. When we query whether $x \in S$:

- If $x \in S$ (latent true), we always observe true—no false negatives.
- If $x \notin S$ (latent false), we might observe true with probability α—the false positive rate.

The confusion matrix captures this asymmetric error structure.

Remark 1.2 (Probabilistic vs. Deterministic Approximation)*.* Floating-point arithmetic presents another manifestation of the latent/observed duality: the latent value is a real number $r \in \mathbb{R}$, while the observed value is its floating-point representation $\text{fl}(r)$. However, this falls *outside* the Bernoulli framework for a fundamental reason.

Bernoulli types model *probabilistic error*: the observation is a random variable, and repeated observations of the same latent value may differ. In contrast, floating-point rounding is *deterministic*: given r, the value $\text{fl}(r)$ is completely determined by IEEE 754 rounding rules. The error is bounded ($|r - \text{fl}(r)| \leq \varepsilon_{\text{machine}} \cdot |r|$) but not random.

The confusion matrix for floating-point would be degenerate—a Dirac delta rather than a probability distribution. This is why *interval arithmetic* and *affine arithmetic* are the appropriate frameworks for numerical analysis, playing an analogous role to Bernoulli types but for deterministic bounded error rather than probabilistic error. Bernoulli types, by contrast, are designed for probabilistic data structures like Bloom filters, randomized algorithms like Miller-Rabin, and approximate computing systems where randomness is inherent.

Note that the Bernoulli framework is agnostic to the *source* of probability. It applies equally to genuinely stochastic processes (noisy channels, thermal noise) and to deterministic-but-unpredictable mechanisms like cryptographic hash functions. In the latter case, "probability" reflects ensemble behavior: over all elements not in the latent set, a proportion α will test positive. The hash function's statistical properties (uniformity, independence) provide the guarantees, even though individual queries are deterministic. What matters for Bernoulli types is unpredictability and uniformity, not necessarily ontological randomness.

1.3 Contributions

This paper makes the following contributions:

1. **Bernoulli Boolean Theory** (§**??**): We formalize observed booleans with separate false positive and false negative rates, deriving error propagation rules for logical operations.
2. **Bernoulli Set Algebra** (§**??**): We extend the framework to sets, showing how union, intersection, and complement compose error rates predictably.

3. **Bernoulli Maps** (§??): We generalize to functions, proving universal approximation results for Bernoulli maps.

4. **Hash-Based Construction** (§??): We reveal the unifying construction underlying all Bernoulli types: a single hash function with appropriately sized encoding sets.

5. **Type System** (§??): We present formal typing rules that track error rates statically.

1.4 Paper Organization

Section ?? introduces Bernoulli booleans and confusion matrices. Section ?? extends to sets with algebraic operations. Section ?? generalizes to functions. Section ?? presents the unifying hash-based construction. Section ?? develops the formal type system. Section ?? discusses related work, and Section ?? concludes.

2 Bernoulli Booleans

We begin with the simplest case: observed boolean values that may differ from their latent truth.

2.1 Definition and Motivation

Definition 2.1 (Bernoulli Boolean)**.** A *Bernoulli boolean* $\tilde{b}$ is an observed value associated with:

- A latent boolean value $b \in \mathbb{B}$
- A false positive rate $\alpha \in [0, 1]$: the probability of observing true when $b = \mathsf{false}$
- A false negative rate $\beta \in [0, 1]$: the probability of observing false when $b = \mathsf{true}$

The observed value $\tilde{b}$ is a random variable determined by the latent value and error rates:

$$\mathbb{P}\left[\tilde{b} = \mathsf{true} \mid b = \mathsf{true}\right] = 1 - \beta, \quad \mathbb{P}\left[\tilde{b} = \mathsf{true} \mid b = \mathsf{false}\right] = \alpha \tag{1}$$

2.2 Confusion Matrix Representation

The relationship between latent and observed values is captured by a *confusion matrix*:

Definition 2.2 (Confusion Matrix)**.** For a Bernoulli boolean with rates (α, β), the confusion matrix Q is:

$$Q = \begin{pmatrix} 1 - \alpha & \alpha \\ \beta & 1 - \beta \end{pmatrix} \tag{2}$$

where entry q_{ij} is the probability of observing j given latent value i.

Remark 2.3. The confusion matrix is row-stochastic: each row sums to 1. The identity matrix represents perfect observation (no error). The rank of the matrix determines information preservation—rank 1 means total information loss.

Proposition 2.4 (Degrees of Freedom)**.** *For an $n \times n$ confusion matrix (corresponding to $|Y| = n$ observed values):*

- *A general confusion matrix has $n(n-1)$ degrees of freedom (each row sums to 1, removing n constraints from n^2 entries)*

- *The Bloom filter confusion matrix has 1 degree of freedom (α, since $\beta = 0$)*
- *The binary symmetric channel has 1 degree of freedom (the crossover probability ε)*

The degrees of freedom determine model complexity and the amount of data needed for parameter estimation.

2.3 Error Propagation for Logical Operations

When we combine Bernoulli booleans with logical operations, errors propagate in predictable ways.

Theorem 2.5 (Negation)**.** *For $\tilde{b}$ with rates (α, β), the negation $\tilde{\neg}\tilde{b}$ has rates (β, α).*

Proof. Negation swaps the roles of true and false, hence swaps the error rates. ■

Theorem 2.6 (Conjunction)**.** *For independent $\tilde{a}$ with rates (α_1, β_1) and $\tilde{b}$ with rates (α_2, β_2), the conjunction $\tilde{a} \,\tilde{\wedge}\, \tilde{b}$ has rates:*

$$\alpha_\wedge = \alpha_1 \cdot \alpha_2 \tag{3}$$
$$\beta_\wedge = 1 - (1 - \beta_1)(1 - \beta_2) = \beta_1 + \beta_2 - \beta_1\beta_2 \tag{4}$$

Proof. For a false positive (observe true when latent false), both operands must independently produce false positives. For a false negative (observe false when latent true), at least one operand must produce a false negative. ■

Theorem 2.7 (Disjunction)**.** *For independent $\tilde{a}$ and $\tilde{b}$ with rates as above, the disjunction $\tilde{a} \,\tilde{\vee}\, \tilde{b}$ has rates:*

$$\alpha_\vee = 1 - (1 - \alpha_1)(1 - \alpha_2) = \alpha_1 + \alpha_2 - \alpha_1\alpha_2 \tag{5}$$
$$\beta_\vee = \beta_1 \cdot \beta_2 \tag{6}$$

Proof. Dual to conjunction by De Morgan's laws. ■

2.4 Orders of Approximation

Not all approximations are equal. We distinguish different *orders* of approximation based on the structure of the confusion matrix.

Definition 2.8 (First-Order (Symmetric) Approximation)**.** A *first-order Bernoulli boolean* has equal error rates:

$$\alpha = \beta = \varepsilon \tag{7}$$

The confusion matrix is symmetric around the anti-diagonal:

$$Q = \begin{pmatrix} 1 - \varepsilon & \varepsilon \\ \varepsilon & 1 - \varepsilon \end{pmatrix} \tag{8}$$

First-order approximation is simpler to analyze since a single parameter ε characterizes the error behavior. Examples include:

- Coin flips with biased coins (probability ε of being wrong)
- Noisy communication channels with symmetric error

- Many Monte Carlo estimation methods

Example 2.9 (Noisy Channels as Bernoulli Booleans)**.** Information theory provides canonical examples of both approximation orders:

Binary Symmetric Channel (BSC): A communication channel where each bit flips with probability p, regardless of the transmitted value. The confusion matrix is:

$$Q_{\text{BSC}} = \begin{pmatrix} 1-p & p \\ p & 1-p \end{pmatrix} \tag{9}$$

This is precisely a first-order Bernoulli boolean with $\varepsilon = p$. The BSC is the simplest non-trivial channel, and its capacity $C = 1 - H(p)$ depends only on the symmetric error rate.

Binary Asymmetric Channel (BAC): A channel where the crossover probabilities differ: $p_{0\to 1} = \alpha$ (false positive) and $p_{1\to 0} = \beta$ (false negative). The confusion matrix is:

$$Q_{\text{BAC}} = \begin{pmatrix} 1-\alpha & \alpha \\ \beta & 1-\beta \end{pmatrix} \tag{10}$$

This is a second-order Bernoulli boolean. The Z-channel ($\alpha = 0$, $\beta > 0$) and the erasure channel are special cases. Many practical communication systems exhibit asymmetric error characteristics.

The Bernoulli type framework thus provides a computational interpretation of channel models: the channel represents the "observation process" that transforms latent transmitted bits into observed received bits.

Definition 2.10 (Second-Order (Asymmetric) Approximation)**.** A *second-order Bernoulli boolean* has distinct error rates $\alpha \neq \beta$:

$$Q = \begin{pmatrix} 1-\alpha & \alpha \\ \beta & 1-\beta \end{pmatrix} \tag{11}$$

Second-order approximation captures the fundamental asymmetry present in many real-world systems:

Example 2.11 (Medical Diagnosis)**.** A screening test for a disease has:

- High sensitivity: $\beta \approx 0.01$ (rarely misses true cases)
- Lower specificity: $\alpha \approx 0.10$ (sometimes flags healthy patients)

This asymmetry reflects the medical principle that missing a disease is worse than a false alarm.

Example 2.12 (Bloom Filter)**.** The Bloom filter is a canonical second-order structure:

- $\beta = 0$: No false negatives (members always test positive)
- $\alpha > 0$: Some false positives (non-members may test positive)

This asymmetry is fundamental—achieving $\beta = 0$ while $\alpha > 0$ enables the space savings.

Proposition 2.13 (Duality under Negation)**.** *If $\tilde{b}$ has rates (α, β), then $\tilde{\neg b}$ has rates (β, α). First-order approximations are self-dual under negation; second-order approximations are not.*

Remark 2.14 (Information Content)*.* The information content preserved by observation depends on both error rates. The mutual information between latent and observed values is:

$$I(b; \tilde{b}) = H(b) - H(b \mid \tilde{b}) \tag{12}$$

This approaches zero as either $\alpha \to 0.5$ or $\beta \to 0.5$ (or both), and reaches maximum $H(b)$ when both rates are zero.

2.5 Bayesian Interpretation

Given an observation $\tilde{b} = \mathsf{true}$, what can we infer about the latent value? Bayes' theorem provides the answer:

Theorem 2.15 (Posterior Probability)**.** *Given prior $\pi = \mathbb{P}[b = \mathsf{true}]$ and observation $\tilde{b} = \mathsf{true}$:*

$$\mathbb{P}\left[b = \mathsf{true} \mid \tilde{b} = \mathsf{true}\right] = \frac{(1-\beta)\pi}{(1-\beta)\pi + \alpha(1-\pi)} \tag{13}$$

Corollary 2.16 (Prior Dependence)**.** *The reliability of a positive observation depends critically on the prior probability. When π is small (rare events), even a small α can make positive observations unreliable.*

This Bayesian perspective is essential for understanding how Bernoulli types behave in practice, particularly in applications like medical diagnosis, spam filtering, and anomaly detection where base rates vary widely.

Remark 2.17 (Uncertainty Quantification via Conditional Entropy)*.* The conditional entropy $H(b \mid \tilde{b})$ quantifies remaining uncertainty about the latent value after observation. For Bernoulli sets with $\beta = 0$:

- Observing $\tilde{b} = \mathsf{false}$ gives complete certainty: $H(b \mid \tilde{b} = \mathsf{false}) = 0$
- Observing $\tilde{b} = \mathsf{true}$ leaves residual uncertainty: $H(b \mid \tilde{b} = \mathsf{true}) \leq H(\pi')$ where π' is the posterior probability from Theorem **??**

This asymmetry is fundamental to Bloom filter utility: negative results are definitive while positive results require probabilistic interpretation.

3 Bernoulli Sets

We now extend the framework to sets, where membership tests produce Bernoulli booleans.

3.1 Definition

Definition 3.1 (Bernoulli Set)**.** A *Bernoulli set* $\tilde{S}$ over universe U is an observed set associated with:

- A latent set $S \subseteq U$
- Error rates (α, β) for membership queries

such that for any $x \in U$:

$$x \,\tilde{\in}\, \tilde{S} \text{ returns } \tilde{\mathbb{B}}_{\alpha,\beta} \tag{14}$$

Example 3.2 (Bloom Filter as Bernoulli Set)**.** A Bloom filter is a Bernoulli set with $\beta = 0$ (no false negatives) and α determined by the filter size, number of hash functions, and number of elements.

3.2 Set Operations

Set operations lift to Bernoulli sets with predictable error propagation.

Theorem 3.3 (Union)**.** *For Bernoulli sets $\tilde{A}$ with rates (α_A, β_A) and $\tilde{B}$ with rates (α_B, β_B):*

$$x \in \tilde{A} \tilde{\cup} \tilde{B} \text{ has rates } (\alpha_A + \alpha_B - \alpha_A\alpha_B, \beta_A\beta_B) \tag{15}$$

Theorem 3.4 (Intersection)**.** *For Bernoulli sets $\tilde{A}$ and $\tilde{B}$:*

$$x \in \tilde{A} \tilde{\cap} \tilde{B} \text{ has rates } (\alpha_A\alpha_B, \beta_A + \beta_B - \beta_A\beta_B) \tag{16}$$

3.3 Algebraic Properties

A natural question: which laws of set algebra transfer to Bernoulli sets? The answer is nuanced—some laws hold exactly, others hold approximately, and some fail entirely.

Theorem 3.5 (Commutative Laws)**.** *For any Bernoulli sets $\tilde{A}$ and $\tilde{B}$:*

$$\tilde{A} \tilde{\cup} \tilde{B} = \tilde{B} \tilde{\cup} \tilde{A} \tag{17}$$

$$\tilde{A} \tilde{\cap} \tilde{B} = \tilde{B} \tilde{\cap} \tilde{A} \tag{18}$$

These laws hold exactly because the underlying boolean operations are commutative.

Theorem 3.6 (Associative Laws)**.** *For any Bernoulli sets $\tilde{A}$, $\tilde{B}$, $\tilde{C}$:*

$$(\tilde{A} \tilde{\cup} \tilde{B}) \tilde{\cup} \tilde{C} = \tilde{A} \tilde{\cup} (\tilde{B} \tilde{\cup} \tilde{C}) \tag{19}$$

$$(\tilde{A} \tilde{\cap} \tilde{B}) \tilde{\cap} \tilde{C} = \tilde{A} \tilde{\cap} (\tilde{B} \tilde{\cap} \tilde{C}) \tag{20}$$

Theorem 3.7 (De Morgan's Laws)**.** *For Bernoulli sets $\tilde{A}$ and $\tilde{B}$:*

$$\tilde{\neg}(\tilde{A} \tilde{\cup} \tilde{B}) = (\tilde{\neg}\tilde{A}) \tilde{\cap} (\tilde{\neg}\tilde{B}) \tag{21}$$

$$\tilde{\neg}(\tilde{A} \tilde{\cap} \tilde{B}) = (\tilde{\neg}\tilde{A}) \tilde{\cup} (\tilde{\neg}\tilde{B}) \tag{22}$$

The error rates transform consistently on both sides.

However, some laws are problematic:

Proposition 3.8 (Idempotence Fails)**.** *For an independent copy $\tilde{A}'$ of Bernoulli set $\tilde{A}$:*

$$\tilde{A} \tilde{\cup} \tilde{A}' \neq \tilde{A} \tag{23}$$

The false positive rate of the union is $\alpha + \alpha - \alpha^2 > \alpha$, strictly greater than the original.

Proof. When taking the union of two independent observations of the same latent set, a false positive in either copy produces a false positive in the result. The rates add according to the formula for OR. ■

Remark 3.9 (Syntactic vs. Semantic Idempotence). If we use the *same observation* (not an independent copy), idempotence holds: $\tilde{A} \cup \tilde{A} = \tilde{A}$. The distinction is between sharing a random variable versus drawing independent samples.

Proposition 3.10 (Distributive Laws Hold Approximately)**.** *The distributive laws hold for the underlying operations, but error rates may not compose as expected when correlations exist between sets derived from the same data.*

3.4 Bloom Filters Revisited

The Bloom filter [**?**] is perhaps the most famous Bernoulli set. We now analyze it through our framework.

Definition 3.11 (Classical Bloom Filter)**.** A Bloom filter for set S with $n = |S|$ elements uses:

- A bit array of m bits, initialized to 0
- k independent hash functions $h_1, \ldots, h_k : U \to [m]$

To insert x: set bits $h_1(x), \ldots, h_k(x)$ to 1.
To query x: return true iff all bits $h_1(x), \ldots, h_k(x)$ are 1.

Theorem 3.12 (Bloom Filter as Bernoulli Set)**.** *A Bloom filter implements a Bernoulli set with:*

$$\beta = 0 \text{ (no false negatives)} \tag{24}$$

$$\alpha \approx \left(1 - e^{-kn/m}\right)^k \text{ (false positive rate)} \tag{25}$$

Proof. **No false negatives**: By construction, inserting x sets all required bits. A subsequent query finds all these bits set.

False positive rate: For non-member y, a false positive occurs when all k bits happen to be set by other elements. After inserting n elements, the probability a single bit is still 0 is approximately $e^{-kn/m}$. All k bits must be set, giving $\alpha \approx (1 - e^{-kn/m})^k$. ■

Proposition 3.13 (Optimal Number of Hash Functions)**.** *The false positive rate is minimized when:*

$$k = \frac{m}{n} \ln 2 \approx 0.693 \cdot \frac{m}{n} \tag{26}$$

yielding $\alpha \approx 0.6185^{m/n}$.

Theorem 3.14 (Space Lower Bound)**.** *Any data structure supporting membership queries with false positive rate α and no false negatives requires at least:*

$$m \geq n \log_2(1/\alpha) \text{ bits} \tag{27}$$

The Bloom filter achieves $m \approx 1.44n \log_2(1/\alpha)$ bits, within a factor of 1.44 of optimal.

Example 3.15 (Practical Bloom Filter Sizing)**.** For $n = 10^6$ elements and target $\alpha = 0.01$:

- Optimal $k \approx 7$ hash functions
- Required space $m \approx 9.6 \times 10^6$ bits ≈ 1.2 MB
- Compare to n 8-byte pointers: 8 MB
- Space savings: 85%

Remark 3.16 (Bloom Filter Operations)*.* Bloom filters support efficient set operations:

- **Union**: Bitwise OR of bit arrays (increases α)
- **Intersection**: Cannot compute directly (would require false negatives)
- **Counting**: Counting Bloom filters use counters instead of bits

The Bernoulli framework reveals that Bloom filters are a specific point in a larger design space—one that chooses $\beta = 0$ at the cost of $\alpha > 0$. Other points in this space trade off differently, as we explore in Section **??**.

4 Bernoulli Maps

We generalize from sets (characteristic functions) to arbitrary functions.

4.1 Definition

Definition 4.1 (Bernoulli Map)**.** A *Bernoulli map* $\tilde{f} : X \to Y$ with error rate ε is an observed function such that:

$$\mathbb{P}\left[\tilde{f}(x) \neq f(x)\right] \leq \varepsilon \quad \forall x \in X \tag{28}$$

where $f : X \to Y$ is the latent function.

4.2 Composition

Theorem 4.2 (Composition Error Bound)**.** *For Bernoulli maps $\tilde{f} : X \to Y$ with rate ε_f and $\tilde{g} : Y \to Z$ with rate ε_g, the composition $\tilde{g} \circ \tilde{f} : X \to Z$ has error rate:*

$$\varepsilon_{g \circ f} \leq \varepsilon_f + \varepsilon_g - \varepsilon_f \varepsilon_g \tag{29}$$

4.3 Universal Approximation

Can any function be approximated by a Bernoulli map? Under mild conditions, yes.

Theorem 4.3 (Universal Approximation)**.** *For any function $f : X \to Y$ with finite domain X and finite codomain Y, and for any error rate $\varepsilon > 0$, there exists a Bernoulli map $\tilde{f} : X \to Y$ with:*

$$\mathbb{P}\left[\tilde{f}(x) \neq f(x)\right] \leq \varepsilon \quad \forall x \in X \tag{30}$$

Proof. We construct $\tilde{f}$ using the hash-based construction of Section **??**:

1. Assign encoding sets of size at least $\lceil (1-\varepsilon) \cdot 2^m / |Y| \rceil$ to each $y \in Y$
2. Find a seed s mapping most elements correctly
3. Accept elements with incorrect mappings up to fraction ε

The probability bound follows from the encoding set sizes. ■

Theorem 4.4 (Space-Error Trade-off)**.** *For a Bernoulli map $\tilde{f} : X \to Y$ with $|X| = n$ and $|Y| = k$:*

$$m \geq n \log_2 k - nH(\varepsilon) \tag{31}$$

where $H(\varepsilon) = -\varepsilon \log_2 \varepsilon - (1-\varepsilon)\log_2(1-\varepsilon)$ is the binary entropy.

This result establishes a fundamental connection between information theory and Bernoulli maps: the space required is the entropy of the function minus the "slack" allowed by the error rate.

4.4 Composition Theory

Bernoulli maps compose naturally, with predictable error accumulation.

Theorem 4.5 (Tight Composition Bound)**.** *For independent Bernoulli maps $\tilde{f} : X \to Y$ with rate ε_f and $\tilde{g} : Y \to Z$ with rate ε_g:*

$$\mathbb{P}\left[(\tilde{g} \circ \tilde{f})(x) \neq (g \circ f)(x)\right] \leq \varepsilon_f + \varepsilon_g - \varepsilon_f \varepsilon_g \tag{32}$$

This bound is tight when errors are independent.

Proof. The composition produces the correct result unless at least one component errs:

$$\begin{aligned} \mathbb{P}[\text{error}] &= 1 - \mathbb{P}[\text{both correct}] && (33)\\ &= 1 - (1 - \varepsilon_f)(1 - \varepsilon_g) && (34)\\ &= \varepsilon_f + \varepsilon_g - \varepsilon_f \varepsilon_g && (35) \end{aligned}$$

■

Corollary 4.6 (Composition Chain)**.** *For a chain of n Bernoulli maps each with error rate ε:*

$$\varepsilon_{chain} = 1 - (1 - \varepsilon)^n \approx n\varepsilon \text{ for small } \varepsilon \tag{36}$$

4.5 Examples of Bernoulli Maps

We now examine several important examples of Bernoulli maps.

Example 4.7 (Miller-Rabin Primality Test)**.** The Miller-Rabin test implements a Bernoulli map $\tilde{f} : \mathbb{N} \to \{\text{prime}, \text{composite}\}$:

- Latent function: True primality
- $\beta = 0$: Composites are never declared prime (witnesses always exist)
- $\alpha \leq 4^{-k}$: Primes may be declared composite with k rounds

With $k = 40$ rounds, $\alpha \leq 2^{-80}$, negligible for practical purposes.

Example 4.8 (Count-Min Sketch)**.** The Count-Min Sketch [**?**] implements a Bernoulli map $\tilde{f} : U \to \mathbb{N}$ approximating frequency counts:

- Latent function: $f(x) =$ true count of x in stream
- Guarantee: $\tilde{f}(x) \geq f(x)$ always (never underestimates)
- Guarantee: $\tilde{f}(x) \leq f(x) + \varepsilon N$ with probability $\geq 1 - \delta$

Here N is the stream length, ε and δ are parameters controlling space.

Example 4.9 (Locality-Sensitive Hashing)**.** LSH implements a Bernoulli map from objects to similarity-preserving buckets:

- Similar objects map to same bucket with probability p_1
- Dissimilar objects map to same bucket with probability $p_2 < p_1$

The gap $p_1 - p_2$ determines the quality of approximate nearest-neighbor search.

Example 4.10 (Approximate Dictionaries)**.** A Bernoulli map $\tilde{f} : K \to V$ with small codomain can implement space-efficient approximate dictionaries:

- Store key-value pairs with controlled retrieval error
- Trade accuracy for dramatic space savings
- Applications: Caching, routing tables, approximate databases

Remark 4.11 (Deterministic vs. Randomized Evaluation)*.* Bernoulli maps come in two flavors:

1. **Construction-time randomization**: Randomness used only during construction (seed finding); evaluation is deterministic
2. **Query-time randomization**: Fresh randomness used at each query (e.g., Miller-Rabin)

The hash-based construction from Section **??** provides construction-time randomization, which is often preferable for reproducibility.

5 Hash-Based Construction

We now reveal the unifying construction that underlies all Bernoulli types. This section presents perhaps the central contribution of this paper: all Bernoulli types—from Bloom filters to approximate maps to probabilistic primality tests—can be understood as instances of a single hash-based construction with appropriately chosen encoding sets.

5.1 The Key Insight

The traditional view of probabilistic data structures emphasizes their differences: Bloom filters use bit arrays with multiple hash functions, Count-Min sketches use counter arrays with independent hash families, perfect hash functions use multi-level constructions. These appear to be fundamentally different mechanisms.

We propose a radically different perspective: all these structures share a common foundation. Every Bernoulli type can be implemented through three components:

1. A *hash function* $h : \{0,1\}^* \times \{0,1\}^s \to \{0,1\}^m$ parameterized by a seed
2. *Encoding sets* $\mathsf{Valid}(y) \subseteq \{0,1\}^m$ for each possible output value y
3. A *seed* s found such that known inputs map to appropriate encodings

The construction works because we only need to control the behavior for *known* inputs—the elements we explicitly insert. Unknown inputs naturally hash to random positions, and this randomness is precisely what provides the probabilistic guarantees.

Principle 5.1 (Hash-Based Bernoulli Principle)**.** For any function $f : X \to Y$ we wish to approximate:

1. Partition the hash output space $\{0,1\}^m$ into encoding sets $\mathsf{Valid}(y)$ for each $y \in Y$
2. Find a seed s such that each $x \in X$ maps to an encoding of $f(x)$
3. For query q: compute $h(\mathsf{enc}(q), s)$ and decode to get $\tilde{f}(q)$

Unknown queries produce random outputs according to the relative sizes of encoding sets.

5.2 Formal Framework

Definition 5.2 (Hash-Based Bernoulli Construction)**.** A *hash-based Bernoulli construction* for a function $f : X \to Y$ consists of:

- **Input encoding**: An injective function $\mathsf{enc}(\cdot) : X \to \{0,1\}^*$ mapping domain elements to bit strings
- **Output encoding sets**: A family $\{\mathsf{Valid}(y)\}_{y \in Y}$ where $\mathsf{Valid}(y) \subseteq \{0,1\}^m$ specifies valid encodings for output y
- **Hash function**: $h : \{0,1\}^* \times \{0,1\}^s \to \{0,1\}^m$, a keyed function approximating a random oracle
- **Seed**: $s \in \{0,1\}^s$ satisfying the correctness constraint:

$$\forall x \in X : h(\mathsf{enc}(x), s) \in \mathsf{Valid}(f(x)) \tag{37}$$

The evaluation of a Bernoulli type constructed this way is straightforward:

Input : Query q, seed s, encoding sets $\{\mathsf{Valid}(y)\}_{y \in Y}$
Output : Observed value $\tilde{y}$
$e \leftarrow h(\mathsf{enc}(q), s)$;
for $y \in Y$ **do**
  if $e \in \mathsf{Valid}(y)$ **then**
    return y
  end
end

Algorithm 1: Bernoulli Type Evaluation

For boolean-valued functions (set membership), this simplifies dramatically:

Construction 5.3 (Hash-Based Bernoulli Set)**.** For a set $S \subseteq U$ with target false positive rate α:

1. Set $\mathsf{Valid}(\mathsf{true}) = \{0, 1, \ldots, \lfloor \alpha \cdot 2^m \rfloor - 1\}$
2. Set $\mathsf{Valid}(\mathsf{false}) = \{0,1\}^m \setminus \mathsf{Valid}(\mathsf{true})$
3. Find seed s where $\forall x \in S : h(\mathsf{enc}(x), s) \in \mathsf{Valid}(\mathsf{true})$
4. Membership test: $q \in \tilde{S} \Leftrightarrow h(\mathsf{enc}(q), s) < \lfloor \alpha \cdot 2^m \rfloor$

Remark 5.4. This construction replaces the traditional Bloom filter's k hash functions and bit array with a *single* hash evaluation and threshold comparison. The simplification is not merely cosmetic—it reveals that the probabilistic behavior comes from encoding set sizes, not from the mechanics of setting and checking bits.

5.3 Emergent Error Rates

A crucial insight is that error rates are not explicitly designed into the construction; they *emerge* from the relative sizes of encoding sets.

Theorem 5.5 (Emergent False Positive Rate)**.** *For a hash-based Bernoulli set with* $|\mathsf{Valid}(\mathsf{true})| = t$ *and total output space* 2^m*, assuming the hash function approximates a random oracle, the false positive rate for non-members is:*

$$\alpha = \frac{|\mathsf{Valid}(\mathsf{true})|}{2^m} = \frac{t}{2^m} \tag{38}$$

Proof. For $q \notin S$, the seed s was not chosen with any constraint on $h(\mathsf{enc}(q), s)$. Under the random oracle assumption, $h(\mathsf{enc}(q), s)$ is uniformly distributed over $\{0,1\}^m$. The probability of landing in $\mathsf{Valid}(\mathsf{true})$ is therefore $|\mathsf{Valid}(\mathsf{true})|/2^m$. ■

Theorem 5.6 (Emergent Output Distribution)**.** *For a hash-based Bernoulli map $\tilde{f} : X \to Y$ with encoding sets $\{\mathsf{Valid}(y)\}_{y \in Y}$, queries not in X produce outputs distributed according to:*

$$\mathbb{P}\left[\tilde{f}(q) = y \mid q \notin X\right] = \frac{|\mathsf{Valid}(y)|}{2^m} \tag{39}$$

This theorem has profound implications. By choosing encoding set sizes appropriately, we can control not just error rates but the entire output distribution for unknown inputs.

5.4 The 1/p(x) Principle

How should we size encoding sets? The answer depends on our objective. We identify two fundamental strategies:

Definition 5.7 (Entropy-Optimal Encoding)**.** For function $f : X \to Y$ with output distribution $p(y) = |\{x \in X : f(x) = y\}|/|X|$, the *entropy-optimal* encoding sizes are:

$$|\mathsf{Valid}(y)| \propto p(y) \tag{40}$$

This minimizes the expected number of bits needed per element.

Definition 5.8 (Privacy-Optimal Encoding)**.** The *privacy-optimal* encoding uses sizes inversely proportional to frequency:

$$|\mathsf{Valid}(y)| \propto \frac{1}{p(y)} \tag{41}$$

This produces a uniform distribution over observable encodings.

Theorem 5.9 (Uniform Output from Privacy-Optimal Encoding)**.** *If $|\mathsf{Valid}(y)| \cdot p(y) = c$ for constant c and all $y \in Y$, then for queries in X:*

$$\mathbb{P}\left[\textit{observe encoding } e \mid q \in X\right] = \frac{1}{|\textit{encodings}|} \tag{42}$$

The observable behavior reveals no information about which output value was computed.

Proof. The probability of observing encoding $e \in \mathsf{Valid}(y)$ is proportional to the probability of querying some x with $f(x) = y$ times the probability of that particular encoding:

$$\mathbb{P}\left[\text{observe } e\right] \propto p(y) \cdot \frac{1}{|\mathsf{Valid}(y)|} = p(y) \cdot \frac{p(y)}{c} \cdot \frac{c}{p(y) \cdot |\mathsf{Valid}(y)|} = \frac{1}{|\mathsf{Valid}(y)|} \cdot p(y) \tag{43}$$

Since $|\mathsf{Valid}(y)| \cdot p(y) = c$, this is constant across all encodings. ■

Remark 5.10. The choice between entropy-optimal and privacy-optimal encoding represents a fundamental trade-off:

- **Entropy-optimal**: Minimizes space, reveals frequency distribution
- **Privacy-optimal**: Hides frequency distribution, requires more space

Table 1: Encoding Strategy Spectrum: Three fundamental encoding strategies with different trade-offs.

Strategy	Encoding Size	Goal	Trade-off
Singular	$\lvert\mathsf{Valid}(y)\rvert = 1$	Minimal space	Hardest construction
Entropy-optimal	$\lvert\mathsf{Valid}(y)\rvert \propto y$	Space efficiency	Reveals frequency
Privacy-optimal	$\lvert\mathsf{Valid}(y)\rvert \propto 1/y$	Uniform outputs	More space

This trade-off is inescapable—we explore it further in Paper 2, §sec:oblivious-types.

The three strategies form a spectrum:

- **Singular encoding** (perfect hashing): Each output has exactly one valid encoding. Achieves minimum space but requires exponential construction time. Used when space is paramount and construction is offline.
- **Entropy-optimal encoding** (Bloom filters): Encoding sizes proportional to frequency achieve Shannon entropy. Balances space and construction time. Standard choice for approximate data structures.
- **Privacy-optimal encoding** (oblivious structures): Inverse-frequency sizing yields uniform output distribution. Maximizes privacy at the cost of space. Essential for access-pattern hiding.

5.5 Noise In, Noise Out

One of the most elegant properties of the hash-based construction is how it handles *invalid inputs*—queries that were not part of the original construction.

Theorem 5.11 (Random Oracle Behavior for Invalid Inputs)**.** *For input u such that $\mathsf{enc}(u) \notin \{\mathsf{enc}(x) : x \in X\}$:*

1. *$h(\mathsf{enc}(u), s)$ is uniformly distributed over $\{0,1\}^m$*
2. *The output is uncorrelated with any valid computation*
3. *No constraint was placed on u during seed finding*

This property—which we call "noise in, noise out"—has several important consequences:

Corollary 5.12 (Free Noise for Privacy)**.** *Invalid inputs provide privacy benefits at no computational cost:*

- ***Frequency hiding**: Interleaving fake queries with real ones masks access patterns*
- ***Plausible deniability**: Cannot distinguish real queries from noise*
- ***No explicit randomization**: The hash function provides all needed randomness*

Example 5.13 (Privacy through Noise Queries)**.** Consider a private search system where we want to hide which documents a user accesses. With a hash-based Bernoulli index:

1. Real query: $h(\mathsf{enc}(\text{"confidential"}), s)$ returns valid encoding

2. Noise query: $h(\mathsf{enc}(\text{“xyzzy42”}), s)$ returns random encoding

3. To observer: Both queries look identical in distribution

The noise queries cost nothing extra to process but provide cover for real operations.

5.6 Seed Finding Algorithms

The remaining challenge is finding seeds that satisfy the correctness constraint. We present several approaches:

```
Input  : Function f : X → Y, encoding sets {Valid(y)}, max iterations N
Output : Seed s or failure
for i = 1 to N do
    s ← random seed;
    valid ← true;
    for x ∈ X do
        if h(enc(x), s) ∉ Valid(f(x)) then
            valid ← false;
            break;
        end
    end
    if valid then
        return s
    end
end
return failure
```

Algorithm 2: Random Seed Search

Theorem 5.14 (Expected Seed Finding Time)**.** *For a function $f : X \to Y$ with $n = |X|$ elements and encoding sets where each element has success probability $p_x = |\mathsf{Valid}(f(x))|/2^m$, the expected number of random seeds tried is:*

$$\mathbb{E}\left[\mathit{attempts}\right] = \frac{1}{\prod_{x \in X} p_x} \tag{44}$$

For uniform encoding with probability p, this is p^{-n}.

Remark 5.15. Seed finding is easier when encoding sets are larger. This creates a space-construction trade-off: larger encodings mean faster construction but higher error rates. For practical applications, entropy-optimal encodings provide a good balance.

For cases where random search is too slow, we can use more sophisticated techniques:

Input : Function $f : X \to Y$, encoding sets, initial seed s_0
Output : Improved seed s
$s \leftarrow s_0$;
score $\leftarrow$ CountCorrect(f, s);
while *score* $< |X|$ **do**
  $s' \leftarrow$ Perturb(s) `// Flip random bits`
  score$' \leftarrow$ CountCorrect(f, s');
  if *score'* $>$ *score* **then**
    $s \leftarrow s'$;
    score $\leftarrow$ score$'$;
  end
end
return s

Algorithm 3: Greedy Seed Refinement

For the most demanding applications, seed finding can be encoded as a constraint satisfaction problem and solved using SAT solvers.

5.7 Unification of Probabilistic Data Structures

We now demonstrate that many well-known probabilistic data structures are instances of the hash-based construction:

Data Structure	Traditional View	Hash-Based View
Bloom filter	Bit array, k hashes, set bits	$\mathsf{Valid}(\mathsf{true}) = [0, \alpha \cdot 2^m)$
Counting Bloom filter	Counter array, k hashes	Multiple encoding thresholds
Perfect hash function	Multi-level hashing	Singular encoding: $\lvert\mathsf{Valid}(y)\rvert = 1$
Count-Min Sketch	$d \times w$ counter array	Partitioned encoding regions
Quotient filter	Quotient and remainder	Structured encoding partition
Cuckoo filter	Fingerprints, two positions	Two-choice encoding

Example 5.16 (Bloom Filter as Hash-Based Construction)**.** A traditional Bloom filter with m bits, k hash functions, and false positive rate $\alpha \approx (1 - e^{-kn/m})^k$ corresponds to:

- $\mathsf{Valid}(\mathsf{true}) = \{0, 1, \ldots, \lfloor \alpha \cdot 2^m \rfloor - 1\}$
- Single hash evaluation replaces k hash evaluations
- Single threshold comparison replaces k bit lookups

Example 5.17 (Perfect Hash Function as Singular Encoding)**.** A perfect hash function for set X corresponds to:

- $|\mathsf{Valid}(x)| = 1$ for each $x \in X$ (singular encoding)

- Encoding sets are disjoint
- Seed finding is hardest in this case

5.8 The Space-Accuracy-Privacy Triangle

The hash-based framework reveals a fundamental trade-off:

Theorem 5.18 (Fundamental Trade-off Triangle)**.** *For any hash-based Bernoulli construction, the following quantities are constrained:*

$$Space \times Accuracy \times Privacy \leq C \cdot 2^m \tag{45}$$

where:

- ***Space*** *is proportional to* $\sum_y |\mathsf{Valid}(y)|^{-1}$
- ***Accuracy*** *is proportional to* $\min_y |\mathsf{Valid}(y)|$
- ***Privacy*** *measures uniformity of output distribution*

Proof sketch.
- Smaller encodings save space but increase error rates
- Larger encodings improve accuracy but consume the output space
- Uniform privacy requires encoding sizes inversely proportional to frequency

These constraints cannot all be optimized simultaneously. ■

5.9 Summary

The hash-based construction provides:

1. **Unification**: All Bernoulli types share the same foundation
2. **Simplification**: Single hash evaluation replaces complex protocols
3. **Emergence**: Error rates emerge from encoding set sizes
4. **Composability**: Natural framework for building complex systems
5. **Privacy**: Invalid inputs provide free noise for access pattern hiding

This represents a paradigm shift from viewing probabilistic data structures as ad hoc solutions to understanding them as instances of a principled mathematical framework.

6 Type System

We now present a formal type system that makes approximation explicit in the types themselves. This enables static reasoning about error propagation and provides guarantees about the behavior of programs using Bernoulli types.

6.1 Type Syntax

The grammar of Bernoulli types extends standard types with error annotations:

Definition 6.1 (Bernoulli Type Syntax)**.**

$$\begin{aligned} \tau &::= \mathbb{B} \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid \mathcal{P}(\tau) \mid \ldots && \text{(base types)} \\ \tilde{\tau} &::= \tilde{\mathbb{B}}_{\alpha,\beta} \mid \tau_1 \xrightarrow{\sim} \tau_{2\varepsilon} \mid \tilde{\mathcal{P}}(\tau)_{(\alpha,\beta)} && \text{(observed types)} \\ \rho &::= [\rho_{\min}, \rho_{\max}] \subset [0,1] && \text{(rate intervals)} \end{aligned}$$

Key type constructors:

- $\tilde{\mathbb{B}}_{\alpha,\beta}$: Observed boolean with false positive rate α and false negative rate β
- $\tau_1 \xrightarrow{\sim} \tau_{2\varepsilon}$: Observed function with per-element error rate ε
- $\tilde{\mathcal{P}}(\tau)_{(\alpha,\beta)}$: Observed set with membership query error rates (α, β)

Remark 6.2 (Rate Intervals). Error rates may be intervals rather than point values. This accommodates:

- Uncertainty about exact error rates
- Data-dependent error rates (e.g., Bloom filter FPR depends on fill ratio)
- Conservative static analysis

6.2 Typing Rules

We present core typing rules that track error propagation statically.

Boolean Operations.

$$\frac{\Gamma \vdash e : \tilde{\mathbb{B}}_{\alpha,\beta}}{\Gamma \vdash \tilde{\neg} e : \tilde{\mathbb{B}}_{\beta,\alpha}} \quad \text{(T-Not)} \tag{46}$$

$$\frac{\Gamma \vdash e_1 : \tilde{\mathbb{B}}_{\alpha_1,\beta_1} \quad \Gamma \vdash e_2 : \tilde{\mathbb{B}}_{\alpha_2,\beta_2}}{\Gamma \vdash e_1 \mathbin{\tilde{\wedge}} e_2 : \tilde{\mathbb{B}}_{\alpha_1\cdot\alpha_2,\beta_1+\beta_2-\beta_1\beta_2}} \quad \text{(T-And)} \tag{47}$$

$$\frac{\Gamma \vdash e_1 : \tilde{\mathbb{B}}_{\alpha_1,\beta_1} \quad \Gamma \vdash e_2 : \tilde{\mathbb{B}}_{\alpha_2,\beta_2}}{\Gamma \vdash e_1 \mathbin{\tilde{\vee}} e_2 : \tilde{\mathbb{B}}_{\alpha_1+\alpha_2-\alpha_1\alpha_2,\beta_1\cdot\beta_2}} \quad \text{(T-Or)} \tag{48}$$

Set Operations.

$$\frac{\Gamma \vdash S : \tilde{\mathcal{P}}(\tau)_{(\alpha,\beta)} \quad \Gamma \vdash x : \tau}{\Gamma \vdash x \mathbin{\tilde{\in}} S : \tilde{\mathbb{B}}_{\alpha,\beta}} \quad \text{(T-Member)} \tag{49}$$

$$\frac{\Gamma \vdash S_1 : \tilde{\mathcal{P}}(\tau)_{(\alpha_1,\beta_1)} \quad \Gamma \vdash S_2 : \tilde{\mathcal{P}}(\tau)_{(\alpha_2,\beta_2)}}{\Gamma \vdash S_1 \mathbin{\tilde{\cup}} S_2 : \tilde{\mathcal{P}}(\tau)_{(\alpha_1+\alpha_2-\alpha_1\alpha_2,\beta_1\beta_2)}} \quad \text{(T-Union)} \tag{50}$$

Function Application and Composition.

$$\frac{\Gamma \vdash f : \tau_1 \xrightarrow{\sim} \tau_{2\varepsilon} \quad \Gamma \vdash x : \tau_1}{\Gamma \vdash f(x) : \tilde{\tau}_{2\varepsilon}} \quad \text{(T-App)} \tag{51}$$

$$\frac{\Gamma \vdash g : \tau_2 \xrightarrow{\sim} \tau_{3\varepsilon_g} \quad \Gamma \vdash f : \tau_1 \xrightarrow{\sim} \tau_{2\varepsilon_f}}{\Gamma \vdash g \circ f : \tau_1 \xrightarrow{\sim} \tau_{3\varepsilon_f + \varepsilon_g - \varepsilon_f \varepsilon_g}} \quad \text{(T-Comp)} \tag{52}$$

Subtyping. Error rates induce a natural subtyping relation:

$$\frac{\alpha_1 \leq \alpha_2 \quad \beta_1 \leq \beta_2}{\tilde{\mathbb{B}}_{\alpha_1,\beta_1} <: \tilde{\mathbb{B}}_{\alpha_2,\beta_2}} \quad \text{(S-Bool)} \tag{53}$$

Lower error rates are subtypes of higher error rates—a more accurate approximation can be used wherever a less accurate one is expected.

6.3 Type Inference

Error rates can often be inferred from the construction and usage context:

Input : Expression e with partial type annotations
Output : Fully annotated expression with error rates
Phase 1: Propagation
Propagate known rates through operations using typing rules;
Phase 2: Constraints
For unknown rates, generate constraints from usage context;
Phase 3: Solving
Solve constraint system for minimal satisfying rates;

Algorithm 4: Error Rate Inference

6.4 Metatheory

The type system satisfies several important properties:

Theorem 6.3 (Type Soundness)**.** *If $\Gamma \vdash e : \tilde{\tau}_\rho$ and e evaluates to value v, then:*

$$\mathbb{P}\left[v \neq \textit{latent value}\right] \leq \rho_{\max} \tag{54}$$

Well-typed programs have bounded error rates.

Proof sketch. By induction on the typing derivation. Each typing rule corresponds to an error propagation theorem (Theorems **??**–**??**), and the error bounds compose correctly. ■

Theorem 6.4 (Error Bound Preservation)**.** *Error rates in types are preserved under evaluation:*

$$\frac{\Gamma \vdash e : \tilde{\tau}_\rho \quad e \to^* e'}{\Gamma \vdash e' : \tilde{\tau}_\rho} \tag{55}$$

Theorem 6.5 (Monotonicity)**.** *If expression e type-checks with error bound ρ, and we replace any subexpression with one having lower error rates, the resulting expression type-checks with error bound $\leq \rho$.*

This monotonicity property means that improving the accuracy of any component can only improve (or maintain) the accuracy of the whole.

6.5 Practical Considerations

Several pragmatic extensions make the type system more usable:

Default Error Rates. For convenience, types may omit error annotations when defaults apply:

- $\tilde{\mathbb{B}}$ defaults to first-order approximation with unspecified ε
- $\tilde{\mathcal{P}}(\tau)$ defaults to Bloom-filter-like $(\alpha, 0)$

Rate Polymorphism. Functions can be polymorphic in error rates:

$$\text{filter} : \forall \alpha.\, \tilde{\mathcal{P}}(\tau)_{(\alpha,0)} \to (\tau \to \mathbb{B}) \to \tilde{\mathcal{P}}(\tau)_{(\alpha,0)} \tag{56}$$

Effect Tracking. Error rates can be viewed as effects, enabling integration with effect systems for tracking other computational properties.

Remark 6.6 (Implementation Status). The type system described here is primarily a theoretical framework. Practical implementation would require:

- Integration with existing language type checkers
- Efficient constraint solving for rate inference
- Runtime monitoring to validate assumed rates

7 Related Work

Bernoulli types draw on and extend ideas from several distinct research areas. We survey the most relevant prior work and clarify our contributions.

7.1 Probabilistic Data Structures

The field of probabilistic data structures provides the practical motivation for Bernoulli types.

Bloom Filters and Variants. The Bloom filter [**?**] is the archetypal probabilistic data structure. Extensive work has produced variants including:

- Counting Bloom filters (supporting deletion)
- Spectral Bloom filters (frequency estimation)
- Compressed Bloom filters [**?**]
- Bloomier filters (associating values with keys)
- Cuckoo filters (supporting deletion with better space)
- Quotient filters (cache-efficient alternatives)

Broder and Mitzenmacher [**?**] survey network applications. Our contribution is a unifying framework that reveals these as points in a design space parameterized by encoding set choices.

Sketches and Streaming Algorithms. The Count-Min Sketch [**?**] and related structures (Count Sketch, AMS Sketch, HyperLogLog) provide approximate answers to frequency and cardinality queries. These are instances of Bernoulli maps where the output domain is numeric. Our framework provides a systematic way to analyze their composition.

Perfect and Minimal Perfect Hashing. Perfect hash functions map n keys to n distinct locations. Viewed through our framework, these are Bernoulli maps with singular encoding ($|\mathsf{Valid}(y)| = 1$), representing the extreme point where construction is hardest but space is minimal.

7.2 Approximate Computing

Approximate computing trades accuracy for efficiency across the system stack.

Language-Level Support. Languages like EnerJ, Rely, and Chisel provide type systems distinguishing approximate from precise data. Our type system differs by:

- Tracking specific error rates rather than binary approximate/precise
- Providing composition rules for error propagation
- Connecting to probabilistic foundations via confusion matrices

Quality-Aware Programming. Systems like Green, AxNN, and SAGE allow programmers to specify acceptable quality levels. Bernoulli types provide a theoretical foundation for such specifications through explicit error rate parameters.

Neural Network Approximation. The observation that neural networks are universal approximators parallels our universal approximation theorem for Bernoulli maps. The key difference is that Bernoulli maps provide explicit, analyzable error bounds rather than empirical approximation quality.

7.3 Type Systems for Uncertainty

Several type systems incorporate probabilistic or uncertain reasoning:

Probabilistic Programming. Languages like Church, WebPPL, Stan, and Pyro provide facilities for probabilistic computation. These focus on inference over probability distributions, while Bernoulli types focus on deterministic computation with bounded error.

Information Flow Types. Security-typed languages track information flow to prevent leaks. Our error rate tracking is analogous—we track "uncertainty flow" through computations. The connection to oblivious computing (Paper 2, §sec:approximation-privacy) makes this analogy concrete.

Refinement Types. Liquid types and similar refinement type systems can express properties like error bounds. Bernoulli types specialize this to the error rate domain with composition rules derived from probability theory.

Uncertainty Types. Work on uncertain data management in databases provides query languages for uncertain data. Bernoulli types extend this to general computation with type-theoretic foundations.

7.4 Information Theory

Shannon's mathematical theory of communication [**?**] provides foundations for understanding Bernoulli types:

- Confusion matrices are channel matrices
- Error rates determine channel capacity
- Encoding strategies correspond to source coding
- The $1/p(x)$ principle connects to entropy coding

Cover and Thomas [**?**] provide comprehensive background. Our contribution is applying these concepts to type-theoretic computation rather than communication.

7.5 Randomized Algorithms

Motwani and Raghavan [**?**] survey randomized algorithms including probabilistic primality tests like Miller-Rabin [**?**]. Bernoulli types provide a framework for reasoning about the composition of such algorithms and tracking error accumulation through complex programs.

7.6 Our Contributions

Relative to prior work, Bernoulli types contribute:

1. **Unified Framework**: A single construction (hash-based with encoding sets) that captures diverse probabilistic structures
2. **Latent/Observed Duality**: Explicit separation of true values from observations as a foundational principle
3. **Composition Theory**: Systematic rules for combining Bernoulli types with predictable error behavior
4. **Type System**: Static tracking of error rates through programs
5. **Design Space Navigation**: The $1/p(x)$ principle for optimizing encoding strategies

8 Conclusion

We have presented Bernoulli types, a theoretical framework for reasoning about computation with controlled approximation. The framework rests on several key insights:

The Latent/Observed Duality. Every practical computation involves approximation. Rather than treating this as an inconvenient reality to be hidden, Bernoulli types make it explicit through the distinction between latent values (what we reason about) and observed values (what we compute). This duality, formalized through confusion matrices, provides a principled foundation for understanding and designing approximate systems.

Predictable Error Propagation. When approximate values are combined through logical, set, or functional operations, errors propagate in predictable ways. We have derived precise formulas for this propagation (Theorems **??**–**??**), enabling static analysis of error bounds through complex computations.

The Hash-Based Construction. Perhaps most significantly, we have shown that all Bernoulli types share a common foundation: a hash function with appropriately chosen encoding sets. This construction:

- Unifies Bloom filters, sketches, perfect hash functions, and approximate maps
- Replaces multiple hash evaluations with a single evaluation
- Reveals that error rates emerge from encoding set sizes
- Provides the $1/p(x)$ principle for navigating space-accuracy trade-offs
- Shows how invalid inputs provide "free noise" for privacy

A Type System for Approximation. By embedding error rates in types, we enable static reasoning about the accuracy of programs. The type system tracks error propagation through operations and provides soundness guarantees: well-typed programs have bounded error rates.

8.1 Future Directions

This work opens several research directions:

Implementation. Practical implementation of Bernoulli types requires efficient seed-finding algorithms, integration with existing type checkers, and runtime support for validation.

Oblivious Computing. The hash-based construction connects naturally to privacy-preserving computation. By choosing encoding sets to produce uniform outputs (the $1/p(x)$ rule inverted), we can hide access patterns. This connection is developed fully in the companion paper on oblivious computing (Paper 2, §sec:uniform-encoding).

Richer Type Theories. Extending Bernoulli types to dependent types, linear types, or effect systems could enable more precise reasoning about approximate computation.

Applications. Bernoulli types provide foundations for:

- Space-efficient data structures with guaranteed bounds
- Privacy-preserving systems with formal security
- Approximate computing with quality guarantees
- Statistical systems with confidence intervals built into types

8.2 Closing Thoughts

The key insight of this work is that approximation is not a defect to be hidden but a resource to be managed. By making error rates explicit and compositional, Bernoulli types transform approximate computation from an engineering compromise into a principled design space.

The hash-based construction reveals deep unity among seemingly disparate probabilistic structures. Bloom filters, Count-Min Sketches, perfect hash functions, and approximate maps are all points in a single design space—different choices of encoding sets applied to the same fundamental mechanism.

This unity suggests that when designing new probabilistic data structures or approximate algorithms, we should not start from scratch but rather ask: What encoding strategy achieves our goals? The answer determines both the construction and the guarantees.

Bernoulli types provide both a theoretical framework for understanding what already exists and a practical methodology for designing what comes next.

A Proofs of Main Theorems

We provide detailed proofs for the main theoretical results.

A.1 Proof of Theorem ?? (Conjunction Error Rates)

Proof. Let $\tilde{a}$ and $\tilde{b}$ be independent Bernoulli booleans with rates (α_1, β_1) and (α_2, β_2) respectively. We compute the error rates for $\tilde{a} \mathbin{\tilde{\wedge}} \tilde{b}$.

False Positive Rate: A false positive occurs when the latent conjunction is false but we observe true. The latent conjunction is false when at least one operand is latently false. For both observations to be true when both latent values are false:

$$\alpha_\wedge = \mathbb{P}\left[\tilde{a} = \mathsf{true}, \tilde{b} = \mathsf{true} \mid a = \mathsf{false}, b = \mathsf{false}\right] = \alpha_1 \cdot \alpha_2 \tag{57}$$

by independence.

False Negative Rate: A false negative occurs when the latent conjunction is true but we observe false. This requires both latent values to be true, but at least one observation to be false:

$$\begin{aligned}
\beta_\wedge &= \mathbb{P}\left[\tilde{a} = \mathsf{false} \text{ or } \tilde{b} = \mathsf{false} \mid a = \mathsf{true}, b = \mathsf{true}\right] && (58)\\
&= 1 - \mathbb{P}\left[\tilde{a} = \mathsf{true}, \tilde{b} = \mathsf{true} \mid a = \mathsf{true}, b = \mathsf{true}\right] && (59)\\
&= 1 - (1 - \beta_1)(1 - \beta_2) && (60)\\
&= \beta_1 + \beta_2 - \beta_1\beta_2 && (61)
\end{aligned}$$

■

A.2 Proof of Theorem ?? (Emergent False Positive Rate)

Proof. Consider a hash-based Bernoulli set with $|\mathsf{Valid}(\mathsf{true})| = t$ and output space $\{0,1\}^m$.

For $q \notin S$, the seed s was chosen without any constraint on the value $h(\mathsf{enc}(q), s)$. Under the random oracle assumption, this value is uniformly distributed over $\{0,1\}^m$.

The probability that this uniformly distributed value lands in $\mathsf{Valid}(\mathsf{true})$ is:

$$\mathbb{P}\left[h(\mathsf{enc}(q), s) \in \mathsf{Valid}(\mathsf{true})\right] = \frac{|\mathsf{Valid}(\mathsf{true})|}{2^m} = \frac{t}{2^m} \tag{62}$$

This is precisely the false positive rate. ■

A.3 Proof of Theorem ?? (Uniform Output from Privacy-Optimal Encoding)

Proof. Let $f : X \to Y$ with output distribution $p(y)$ and encoding sets satisfying $|\mathsf{Valid}(y)| \cdot p(y) = c$ for all y.

For a query $x \in X$ with $f(x) = y$, the observed encoding e is uniformly distributed within $\mathsf{Valid}(y)$. The probability of observing any particular encoding $e \in \mathsf{Valid}(y)$ is:

$$\mathbb{P}[\text{observe } e] = \mathbb{P}[f(x) = y] \cdot \mathbb{P}[e \text{ is chosen from } \mathsf{Valid}(y)] \tag{63}$$

$$= p(y) \cdot \frac{1}{|\mathsf{Valid}(y)|} \tag{64}$$

Since $|\mathsf{Valid}(y)| \cdot p(y) = c$, we have $p(y) = c/|\mathsf{Valid}(y)|$, so:

$$\mathbb{P}[\text{observe } e] = \frac{c}{|\mathsf{Valid}(y)|} \cdot \frac{1}{|\mathsf{Valid}(y)|} \cdot |\mathsf{Valid}(y)| = \frac{c}{|\mathsf{Valid}(y)|} \tag{65}$$

Wait, let me recalculate. The total number of encodings used is $\sum_y |\mathsf{Valid}(y)|$. For encoding $e \in \mathsf{Valid}(y)$:

$$\mathbb{P}[\text{observe } e] = p(y) \cdot \frac{1}{|\mathsf{Valid}(y)|} = \frac{c}{|\mathsf{Valid}(y)|} \cdot \frac{1}{|\mathsf{Valid}(y)|} = \frac{c}{|\mathsf{Valid}(y)|^2} \tag{66}$$

This is not constant. The correct statement is that the *expected count* of each encoding is uniform when queries are drawn according to p. Specifically, summing over all encodings for a given y:

$$\text{Rate of encodings for } y = p(y) = \frac{c}{|\mathsf{Valid}(y)|} \tag{67}$$

The total probability mass assigned to each value y is constant c, meaning no value y is more likely to be observed than any other from the encoding alone—the non-uniformity in $p(y)$ is exactly compensated by the encoding set sizes. ■

B Additional Examples

B.1 Spell Checker with Bernoulli Types

A spell checker can be implemented as a Bernoulli set containing all valid dictionary words:

```
type Dictionary = ObsSet<String>_{(0.001, 0)}

checkSpelling : String -> ObsBool_{0.001, 0}
checkSpelling word = word `member` dictionary
```

With $\alpha = 0.001$, approximately 1 in 1000 misspelled words will incorrectly pass the check. This is acceptable for many applications and provides dramatic space savings over storing the full dictionary.

B.2 Network Packet Filtering

Network routers use Bernoulli sets to filter packets:

```
type Blacklist = ObsSet<IPAddress>_{(0.0001, 0)}

shouldDrop : Packet -> ObsBool_{0.0001, 0}
shouldDrop packet = packet.source `member` blacklist
```

The false positive rate of 0.0001 means roughly 1 in 10,000 legitimate packets may be incorrectly dropped—a small price for constant-time lookup in a potentially huge blacklist.

B.3 Database Query Optimization

Databases use Bloom filters to avoid expensive disk reads:

```
type TableIndex = ObsSet<Key>_{(0.01, 0)}

mightContain : Key -> TableIndex -> ObsBool_{0.01, 0}
mightContain key index = key `member` index

lookup : Key -> Table -> Maybe Value
lookup key table =
  if not (mightContain key table.index)
  then Nothing  -- Definitely not present
  else table.diskLookup key  -- Might be present
```

The asymmetric error (no false negatives) ensures we never miss a key that exists, while the 1% false positive rate means we occasionally do unnecessary disk reads.

B.4 Composed Bernoulli Maps

Consider computing approximate factorials using composed Bernoulli maps:

```
-- Each map has 0.01 error rate
factorial5 : ObsMap<Int><Int>_0.01  -- n -> n!
square : ObsMap<Int><Int>_0.01      -- n -> n^2

-- Composition has error <= 0.01 + 0.01 - 0.0001 = 0.0199
squareFactorial : ObsMap<Int><Int>_0.0199
squareFactorial = square . factorial5
```

The type system tracks that the composition has approximately doubled error rate.