Regular Types in the Bernoulli Model: When Equality Isn't Equal

Alexander Towell
atowell@siue.edu

January 24, 2026

Abstract

We explore the fundamental tension between regular types—which assume perfect observation of equality—and Bernoulli types, where we can only observe latent equality through a noisy channel. Regular types require equality to be reflexive, symmetric, and transitive, but these axioms assume perfect observation. In reality, equality comparison is itself a computation that *observes* the latent mathematical fact of equality.

Bernoulli types make this explicit: when we write `a == b`, we are not accessing the latent truth but rather observing it through potentially faulty comparison operations. This paper analyzes how the three axioms of equality—reflexivity, symmetry, and transitivity—behave under noisy observation. We show that while latent equality always satisfies these axioms, observed equality satisfies them only probabilistically.

This shift from assuming perfect equality to observing approximate equality enables new programming paradigms for distributed systems (where different nodes may observe different equalities), privacy-preserving computation (where equality is deliberately obscured), and fault-tolerant algorithms (where hardware errors affect observations). We present design patterns for safe integration of Bernoulli types with traditional programming, and analyze the fundamental limits imposed by rank-deficient confusion matrices.

1 Introduction

1.1 The Regular Type Foundation

Alexander Stepanov's concept of regular types forms the foundation of generic programming. A regular type supports:

- **Copy construction**: `T a(b);` creates `a` as a copy of `b`
- **Assignment**: `a = b;` makes `a` equal to `b`
- **Equality**: `a == b` returns `true` iff `a` and `b` are equal
- **Destruction**: Objects can be destroyed

Most critically, equality must satisfy the equivalence relation axioms:

- **Reflexive**: `a == a`
- **Symmetric**: `a == b` implies `b == a`

- **Transitive**: `a == b` and `b == c` implies `a == c`

These axioms enable generic programming: containers, sorting algorithms, and equational reasoning all depend on equality being a true equivalence relation.

1.2 The Latent/Observed Gap

Bernoulli types [**?**] reveal that these constraints assume perfect observation of equality. The distinction is fundamental:

- **Latent**: Two values are either mathematically equal or not
- **Observed**: We can only observe this equality through computation

When `operator==` returns `bernoulli<bool>`, it acknowledges this gap:

```
bernoulli<int> a(42);
assert(a == a);  // Observing latent equality; observation might be incorrect!
```

This paper argues that recognizing the latent/observed distinction in equality is not a deficiency but a more honest model of computation.

1.3 Contributions

This paper makes the following contributions:

1. **Axiom Analysis** (§**??**): We analyze how reflexivity, symmetry, and transitivity behave under noisy observation.
2. **Compositional vs. Direct Approximation** (§**??**): We examine the structural differences between approximating components independently versus approximating composite types as units.
3. **Programming Paradigms** (§**??**): We show how Bernoulli types enable eventually-consistent data structures and Byzantine fault tolerance.
4. **Implementation Patterns** (§**??**): We provide practical guidance for integrating Bernoulli types with traditional programming.
5. **Fundamental Limits** (§**??**): We analyze rank-based indistinguishability limits.

1.4 Connection to Companion Papers

This paper extends the Bernoulli types framework [**? ?**]. Paper 1 establishes the latent/observed duality and confusion matrix formalism. Paper 2 connects approximation to privacy through uniform distributions. This paper examines how these foundations affect the regular type concept.

2 Equality Axioms Under Noisy Observation

2.1 Probabilistic Equality

For Bernoulli types, equality returns a Bernoulli boolean:

Definition 2.1 (Observed Equality)**.** For Bernoulli values $\tilde{a}, \tilde{b}$, the observed equality $\tilde{a} == \tilde{b}$ returns a Bernoulli boolean $(a \tilde{=} b)$ with:

$$\mathbb{P}\left[(a \tilde{=} b) = \mathsf{true} \mid a = b\right] = 1 - \alpha \tag{1}$$

$$\mathbb{P}\left[(a \tilde{=} b) = \mathsf{true} \mid a \neq b\right] = \beta \tag{2}$$

where α is the false negative rate (failing to observe true equality) and β is the false positive rate (observing equality when none exists).

Note the reversal from the standard Bernoulli set convention: for equality, a "false negative" means failing to see that equal things are equal.

2.2 Reflexivity Under Observation

Theorem 2.2 (Observed Reflexivity)**.** *Latent reflexivity always holds: a value is mathematically equal to itself. But observed reflexivity fails with probability α:*

$$\mathbb{P}\left[(a \tilde{=} a) = \mathsf{true}\right] = 1 - \alpha \tag{3}$$

Proof. The latent fact $a = a$ always holds. The observation process introduces error with rate α, yielding the stated probability. ■

Remark 2.3 (Sources of Reflexivity Failure). The gap between latent and observed reflexivity arises from:

- **Measurement uncertainty**: Physical sensors observing the same value twice
- **Transmission errors**: Network corrupting equality checks
- **Privacy mechanisms**: Deliberately noisy observations for security
- **Hardware faults**: Bit flips during comparison

2.3 Symmetry Under Observation

Theorem 2.4 (Observed Symmetry)**.** *Latent symmetry always holds: if $a = b$ then $b = a$. However, each observation is independent:*

$$\mathbb{P}\left[(a \tilde{=} b) = \mathsf{true} \mid a = b\right] = \mathbb{P}\left[(b \tilde{=} a) = \mathsf{true} \mid b = a\right] \tag{4}$$

but individual observations may differ:

$$\mathbb{P}\left[(a \tilde{=} b) = \mathsf{true} \wedge (b \tilde{=} a) = \mathsf{false} \mid a = b\right] = \alpha(1 - \alpha) > 0 \tag{5}$$

Proof. Each observation is an independent Bernoulli trial with success probability $1 - \alpha$ given latent equality. The probability of observing true then false is $\alpha(1 - \alpha)$. ■

Example 2.5 (Symmetry Violation in Practice)**.** Consider the code:

```
if (a == b) {      // First observation of latent equality
    assert(b == a);  // Second independent observation - might fail!
}
```

This reflects reality: asking "is A equal to B?" twice may yield different answers due to measurement noise, timing, or observer differences.

2.4 Transitivity Under Observation

Transitivity suffers most severely from the latent/observed gap.

Theorem 2.6 (Observed Transitivity)**.** *Even when latent transitivity holds ($a = b \wedge b = c \implies a = c$), observations compound errors:*

$$\mathbb{P}\left[(a \tilde{=} c) = \mathsf{true} \mid (a \tilde{=} b) = \mathsf{true} \wedge (b \tilde{=} c) = \mathsf{true} \wedge a = b = c\right] = 1 - \alpha \tag{6}$$

The third observation is independent of the first two.

Proof. Given that $a = b = c$ latently holds, each observation is an independent trial. Observing $(a \tilde{=} b) = \mathsf{true}$ and $(b \tilde{=} c) = \mathsf{true}$ does not change the probability of the third observation $(a \tilde{=} c)$. ■

Corollary 2.7 (Transitivity Chain Degradation)**.** *For a chain of n equalities $a_1 = a_2 = \cdots = a_{n+1}$ (all latently true), the probability of observing all n equalities correctly is:*

$$\mathbb{P}\left[\bigwedge_{i=1}^{n} (a_i \tilde{=} a_{i+1}) = \mathsf{true}\right] = (1 - \alpha)^n \tag{7}$$

which degrades exponentially in n.

The key insight: we're not observing "transitivity" but rather three independent observations of latent equalities. Each observation has its own error, and they don't compound in a way that preserves the logical relationship.

2.5 Cascading Effects

The probabilistic nature of equality affects all derived operations:

Proposition 2.8 (Container Effects)**.** *For containers using Bernoulli equality:*

- *`std::set<bernoulli<T>>` may contain "duplicates" (latently equal elements that weren't observed as equal during insertion)*
- *`std::sort` may not produce a total order (transitivity failures)*
- *Class invariants become probabilistic guarantees*

3 Compositional vs. Direct Approximation

A fundamental design choice emerges when applying Bernoulli approximation to composite types.

3.1 Two Approaches

Definition 3.1 (Compositional Approximation)**.** For a product type $A \times B$, *compositional approximation* creates:

$$\text{Pair of Bernoulli values} : (\mathcal{B}\langle A\rangle, \mathcal{B}\langle B\rangle) \tag{8}$$

where each component is independently approximated.

Definition 3.2 (Direct Approximation)**.** For a product type $A \times B$, *direct approximation* creates:

$$\text{Bernoulli pair} : \mathcal{B}\langle A \times B\rangle \tag{9}$$

where the pair itself is approximated as a single unit.

3.2 Structural Differences

Theorem 3.3 (Compositional vs. Direct Structure)**.** *Compositional and direct approximation exhibit fundamentally different properties:*

***Compositional**:*

- *Confusion matrix is Kronecker product of component matrices*
- *Independent error structure between components*
- *Higher-order due to independent parameters*

***Direct**:*

- *Single unified confusion matrix over product space*
- *Can exhibit correlation between component errors*
- *May have different error patterns than component-wise errors*

Example 3.4 (Compositional Equality)**.** For a struct with multiple fields using compositional approximation:

```
struct Person {
    bernoulli<string> name;  // Independent approximation
    bernoulli<int> age;      // Independent approximation
};

bernoulli<bool> operator==(const Person& p1, const Person& p2) {
    return (p1.name == p2.name) && (p1.age == p2.age);
    // Error compounds: 1 - (1-eps_name)(1-eps_age)
}
```

3.3 Design Guidance

Proposition 3.5 (When to Use Each Approach)**.** ***Compositional approximation** is preferred when:*

- *Components are logically independent*

- *Implementing separate approximations is straightforward*
- *Error budgets can be allocated per-component*

Direct approximation *is preferred when:*

- *Components have natural correlation structure*
- *A single approximation algorithm handles the entire type*
- *Lower overall error rates are critical*

4 New Programming Paradigms

4.1 Probabilistic Contracts

Traditional contracts assume deterministic predicates. With Bernoulli types, contracts become probabilistic:

```
void insert(bernoulli_set<T>& s, const T& value) {
    // Postcondition: P[value in s] >= 1 - epsilon
}
```

4.2 Eventually-Consistent Data Structures

Bernoulli types naturally model eventually-consistent systems:

Example 4.1 (Distributed Set)**.** In a distributed system with network partitions:

```
class distributed_set {
    bernoulli_set<T> local_view;

    bernoulli<bool> contains(const T& value) {
        // Returns probabilistic result based on:
        // - Local information
        // - Network reliability
        // - Synchronization lag
    }
};
```

Different nodes may observe different set memberships during partition healing.

4.3 Byzantine Fault Tolerance

Bernoulli types can model Byzantine failures where replicas may return inconsistent results:

Example 4.2 (Byzantine Value)**.** `template<typename T>`

```
class byzantine_value {
    std::vector<bernoulli<T>> replicas;

    bernoulli<T> read() {
        // Majority voting with probabilistic agreement
```

```
        return majority_vote(replicas);
    }
};
```

The error rate reflects the probability of Byzantine consensus failure.

4.4 Privacy-Preserving Equality

Differential privacy requires adding noise to queries, naturally fitting the Bernoulli model:

Example 4.3 (Private Database)**.**
```
class private_database {
    double privacy_epsilon;

    bernoulli<bool> equals(const record& a, const record& b) {
        bool actual = (a.id == b.id);
        // Add noise for differential privacy
        double noise = laplace_noise(1.0 / privacy_epsilon);
        return bernoulli<bool>::from_probability(sigmoid(actual ? 1+noise : noise));
    }
};
```

5 Implementation Patterns

5.1 Mixing Deterministic and Probabilistic Code

Safe integration requires careful boundaries:

```
// Convert Bernoulli to deterministic with threshold
template<typename T>
std::optional<T> to_deterministic(const bernoulli<T>& b, double confidence) {
    if (b.confidence() >= confidence) {
        return b.expected_value();
    }
    return std::nullopt;
}

// Lift deterministic to Bernoulli
template<typename T>
bernoulli<T> to_bernoulli(const T& value) {
    return bernoulli<T>(value, 0.0);  // Zero error rate
}
```

5.2 Type System Extensions

```
// Type-level error bounds
template<typename T, double MaxError>
class bounded_bernoulli {
    static_assert(MaxError >= 0.0 && MaxError <= 1.0);
    // Compile-time error bound guarantee
};
```

```
// Concepts for Bernoulli-aware algorithms
template<typename T>
concept BernoulliComparable = requires(T a, T b) {
    { a == b } -> std::convertible_to<bernoulli<bool>>;
};
```

5.3 Interoperability Guidelines

Proposition 5.1 (Design Guidelines)**.**

- ***Surface latent vs. observed****: Expose both a latent-spec notion of equality and an observed comparator returning a probability.*
- ***Stabilize by repetition****: For critical checks, aggregate multiple observations (majority vote) and bound error via concentration inequalities.*
- ***Contain scope****: Restrict observed equality to local algorithmic steps; publish only stabilized results across module boundaries.*
- ***Document assumptions****: Record independence/memoryless assumptions and expected error rates at API boundaries.*

6 Fundamental Limits

Even with perfect knowledge of confusion matrix parameters, some latent values remain asymptotically indistinguishable due to rank structure.

Theorem 6.1 (Asymptotic Indistinguishability)**.** *Consider the confusion matrix Q for observing equality between regular type values. If rank$(Q) < |T|$ where T is the type's value space, then there exist latent configurations that remain indistinguishable even with unlimited observations.*

Proof. If Q has rank $r < |T|$, its row space has dimension r. Values whose confusion matrix rows are identical produce identically distributed observations. No statistical test can distinguish them. ■

Corollary 6.2 (Privacy from Rank Deficiency)**.** *The rank deficiency creates inherent privacy—some information remains protected regardless of computational resources or observation count.*

Remark 6.3 (Connection to Oblivious Computing)*.* This analysis connects to Paper 2's oblivious computing framework [**?**]: by designing observation processes with intentional rank deficiency, we achieve unconditional privacy where private inputs remain indistinguishable even with unlimited computational power.

7 Related Work

7.1 Regular Types and Generic Programming

Stepanov's work on generic programming establishes the regular type concept. Our contribution is examining how these foundations behave under the latent/observed duality.

7.2 Probabilistic Programming

Languages like Church and Stan support probabilistic computation but don't address the regular type concept directly. Bernoulli types focus specifically on the observation gap.

7.3 Eventual Consistency

CRDTs achieve eventual consistency through commutative operations. Bernoulli types model the uncertainty during convergence and provide explicit error bounds.

7.4 Approximate Computing

Work on approximate hardware and quality-of-service programming shares goals but operates at different abstraction levels.

8 Conclusion

Regular types assume perfect observation of equality—that when we write `a == b`, we access the latent mathematical truth. Bernoulli types acknowledge that equality comparison is itself an observation through a potentially noisy channel.

Key findings:

- **Reflexivity**: $\mathbb{P}\left[(a \mathrel{\tilde{=}} a) = \mathsf{true}\right] = 1 - \alpha$ (not guaranteed)
- **Symmetry**: Individual observations may differ even for latently equal values
- **Transitivity**: Three independent observations, each with its own error
- **Compositional vs. Direct**: Different error structures for composite types

Implications:

- **Honesty**: Acknowledges that all computation observes latent truth imperfectly
- **Robustness**: Systems expecting observation errors handle real-world failures gracefully
- **Scalability**: Accepting approximate observations enables better scaling
- **Privacy**: The gap between latent and observed can protect sensitive information
- **Distributed systems**: Different nodes may observe different equalities—a feature, not a bug

By making the latent/observed distinction explicit, Bernoulli types don't break programming—they make it more honest about the nature of computation.