Chapter 1: Preliminaries

Probabilistic Minds Consortium (voids.blog)

2024/2025

1 Introduction

This section lays out the foundation of a finite, capacity-based system:

- We introduce observer states and environment states, each of which is strictly finite.
- We define a capacity function that determines how fine-grained probabilities can be.
- We describe patterns and features, each of which is also finite, and show how to assign discrete probability increments.
- We specify consistency conditions (the axioms) that keep the system in a well-defined finite realm.
- Finally, we show how update mechanisms allow us to move from one observer–environment joint state to another, again staying within finite increments.

Explanation:
Think of this section as setting up the "data types" and basic operations for your program. In a computer implementation, each finite set (e.g., observer states, environment states, patterns, features) corresponds to arrays or lists of fixed size. The capacity function acts as a parameter controlling resolution—similar to how you might set a "precision" variable in numerical code. This foundational layer ensures that all further operations are computed using only finite, well-defined data structures.

2 Finite Sets and Basic Objects

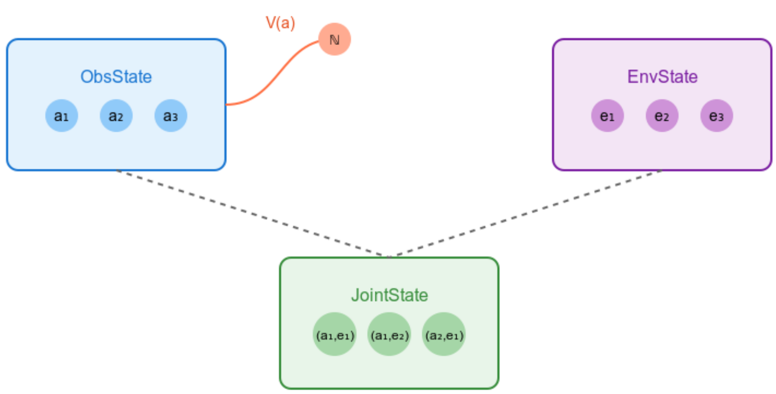
2.1 Observer States

- **Sort:** $\mathrm{ObsState}$.
- **Interpretation:** Each element $a \in \mathrm{ObsState}$ is a possible "observer state."
- **Finite Requirement:** $\mathrm{ObsState}$ must be a finite set in any concrete model:

$$\textbf{Definition:} \quad a \in \mathrm{ObsState} \;\Rightarrow\; \text{"}a\text{ is an observer state."}$$

In code:
You would typically implement `ObsState` as an array or list with a fixed number of entries. This guarantees that loops over observer states terminate after a known number of iterations, ensuring all operations remain finite.

IT Analogy:
If you're coding a system with various user contexts or modes, then $\mathrm{ObsState}$ might store these modes in a short array or dictionary (no infinite expansions needed).



2.2 Environment States

- **Sort:** $\mathrm{EnvState}$.
- **Interpretation:** Each element $e \in \mathrm{EnvState}$ is a discrete environment condition.
- **Finite Requirement:** $\mathrm{EnvState}$ is also finite.
- **Definition:** $e \in \mathrm{EnvState}$ "e is an environment state."

In code:
Environment states should be stored in a fixed-size data structure (like a list or an enumerated type) so that any operations (such as state transitions) stay within known limits.

IT Analogy:
Think "deployment environment" or "server config," each enumerated as a finite set. No unbounded environment states are permitted.

2.3 Joint System States

- **Pairing:** If $a \in \mathrm{ObsState}$ and $e \in \mathrm{EnvState}$, we can speak of the pair (a, e).
- Optionally, define a notational sort:

$$\mathrm{JointState} = \mathrm{ObsState} \times \mathrm{EnvState}.$$

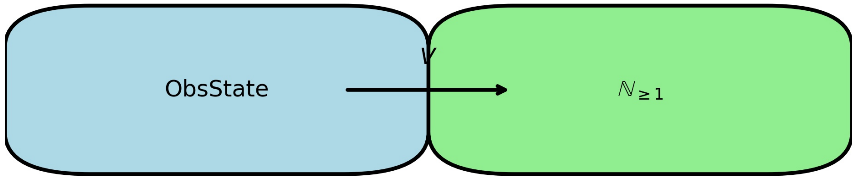
  Since it is a Cartesian product of two finite sets, $\mathrm{JointState}$ is also finite.

Implementation:
A typical grains-coded system might store the current state as `(current_observer, current_environment)`—both small integer IDs, for instance.
Explanation:
The joint state (a, e) can be implemented as a tuple or record. In code, combining two finite arrays (for observers and environments) yields a finite Cartesian product, much like nested loops over finite sets.

3 Observer Capacity and Associated Parameters

3.1 Capacity Function

- **Symbol:** $V(a)$.
- **Domain:** $V : \mathrm{ObsState} \to \mathbb{N}_{\geq 1}$.
- **Meaning:** For each observer state a, we have a "capacity" $V(a)$ — akin to bits of precision or levels of detail.



$V(a)$: Capacity of observer state a (bits of precision / levels of detail)

Explanation:
The capacity function $V(a)$ acts like a lookup table for each observer state, controlling how fine-grained probability increments can be. In an object-oriented setting, each observer might have a `capacity` field that influences subsequent calculations.

IT Perspective:
Capacity might represent how finely observer state a can store grains-coded fractions or distances. Higher capacity implies smaller increments.

3.2 Capacity and Fineness of Probability

We typically define an integer function $N(a)$ tied to $V(a)$. For instance, we might set

$$N(a) = 2^{V(a)} \quad \text{or} \quad N(a) = 10^{V(a)}.$$

The system demands a non-decreasing property:

$$\forall\, a, a' : \quad \big(V(a') > V(a)\big) \implies N(a') \geq N(a).$$

Hence, if the observer's capacity vantage grows from a to a', the denominator $N(a')$ for grains-coded fractions also grows (or at least doesn't shrink).
Explanation:
This ensures that an observer with higher capacity can represent probabilities with finer increments. In code, if capacity is upgraded, probability values are recalculated using a larger denominator $N(a)$—akin to moving from a 10-bit to a 16-bit fixed-point scheme.

Practical Implementation:
One might generate an array of discrete values from 0 to 1 in steps of $1/N(a)$, then use it to validate or discretize probabilities throughout the system.

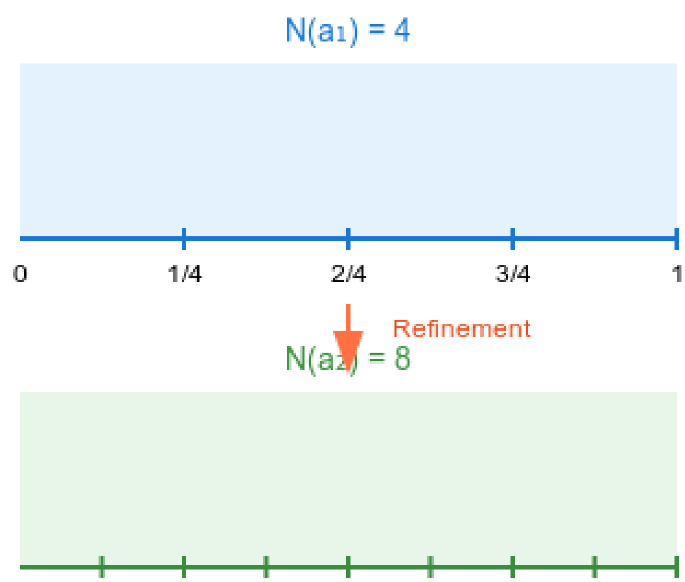
3.3 Defining $N(a)$

- **Function Symbol:** $N : \mathrm{ObsState} \to \mathbb{N}_{\geq 1}$.
- **Interpretation:** For example, if $N(a) = 10$, then valid increments are

$$\left\{\frac{0}{10}, \frac{1}{10}, \ldots, 1\right\}.$$

- The larger $N(a)$ is, the smaller each grains-coded step $\frac{1}{N(a)}$ becomes.

Practical Implementation:
You might store `ncap[a] = 100` to indicate that observer state a has a denominator of 100 grains. The code then interprets all probabilities as integer counts out of 100.



Explanation:
This relationship between $V(a)$ and $N(a)$ is crucial for representing higher precision in a finite manner. As $V(a)$ increases, $N(a)$ increases, providing finer increments for probability assignments, while still remaining countable and finite.

4 Probability Assignments and Patterns

4.1 Patterns

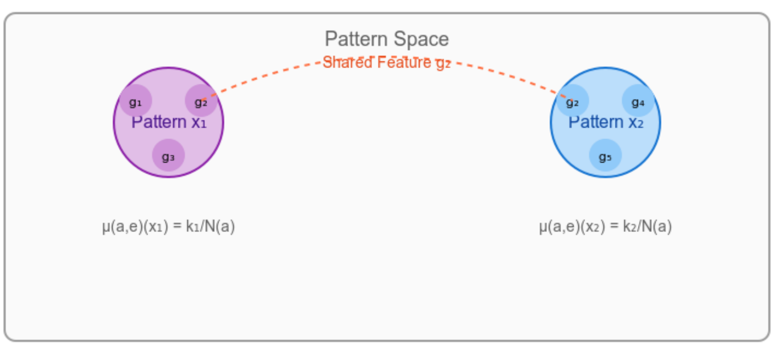
- **Sort:** $\mathrm{Pattern}$.
- The set $\mathrm{Pattern}$ is finite.
- **Example:** A "pattern" might be an event label or a recognized shape.

Explanation:
Patterns function as labels or categories. In a program, they could be implemented as keys in a dictionary or as small enumerations suitable for classification tasks.

4.2 Features of a Pattern

- **Sort:** $\mathrm{Feature}$.
- For each pattern $x \in \mathrm{Pattern}$, define a finite subset $\mathrm{Feat}(x) \subseteq \mathrm{Feature}$.
- The predicate $\mathrm{FeatureOf}(x, g)$ means "feature g is associated with pattern x."

Explanation:
Each pattern has a finite list of features. For instance, you could map a pattern ID to an array of feature IDs in an object or dictionary, ensuring finite looping over features.



4.3 Probability Measure $\mu(a, e)(x)$ at a Joint State

The grains-coded measure $\mu(a, e)(x)$ provides a discrete probability representation adapted to the observer's current capacity. As capacity increases, the measure can handle finer probability increments while remaining discrete.

- **Predicate:** $\mathrm{Mu}(a, e, x, \alpha)$.
- **Meaning:** "At joint state (a, e), the grains-coded measure of pattern x is α."
- **Domain:** $\alpha \in \left\{0, \frac{1}{N(a)}, \ldots, 1\right\}$.
- Every measure is stored as $\frac{k}{N(a)}$.

Implementation:
One could use a 3D data structure keyed by (a, e, x), storing an integer for each fraction $\frac{k}{N(a)}$. This avoids floating-point errors.

Code Snippet (grain_probability.py):

```
from collections import defaultdict
Mu = defaultdict(int)


def set_probability(a, e, x, k, capacity):
    """
    Mu[a,e,x] = k, meaning grains-coded fraction k/capacity.
    """
    Mu[(a,e,x)] = k
```

Hence, "the measure of x at (a, e)" is an integer k out of capacity $N(a)$. See `grain_probability.py` for a complete demonstration.

4.4 Feature Distributions $p_x(a, e)$

- **Predicate:** $\mathrm{Px}(a, e, x, g, \beta)$.
- **Domain:** $\beta \in \left\{0, \frac{1}{N(a)}, \ldots, 1\right\}$.
- Interpreted as "the probability of feature g within pattern x is β."

Once a pattern's probability is set, sub-probabilities can be assigned to each feature in grains-coded fractions. This ensures arithmetic remains predictable and discrete, akin to fixed-point logic.

Hierarchical Probability:
One could define $\mu(a, e)(x)$ as the total measure for x, then decompose it among features $g \in \mathrm{Feat}(x)$ with grains-coded sub-fractions summing exactly to $\mu(a, e)(x)$. No continuous distribution is needed — purely grains-coded.

5 Consistency Conditions for Probability

5.1 Discrete Probability Values (Axiom A1)

$$\forall\, a, e, x, \alpha : \quad \mathrm{Mu}(a, e, x, \alpha) \implies \alpha \in \left\{ \tfrac{k}{N(a)} \mid k = 0, \ldots, N(a) \right\}.$$

No real continuum is permitted; only grains-coded fractions at capacity $N(a)$.
In code:
When assigning probabilities, a validation function should confirm that each value is one of these discrete increments, preventing floating-point contamination.

5.2 Feature Distribution Increments (Axiom A2)

$$\forall\, a, e, x, g, \beta : \quad \mathrm{Px}(a, e, x, g, \beta) \implies \beta \in \left\{ \tfrac{k}{N(a)} \mid k = 0, \ldots, N(a) \right\}.$$

All feature probabilities are grains-coded. Generally, we also require:

$$\sum_{g \in \mathrm{Feat}(x)} \beta_g = \mu(a, e)(x).$$

Explanation:
This ensures all feature-level probabilities share the same discrete granularity. Functions that alter feature probabilities must re-scale or normalize to remain within the set.

5.3 Capacity Monotonicity (Axiom A3)

$$\forall\, a, a' : \quad \big(V(a') > V(a)\big) \implies N(a') \geq N(a).$$

Higher capacity vantage means at least the same or greater denominator—no infinite leaps or real values appear.
Explanation:
In practice, upgrading to a higher resolution mode triggers integer arithmetic for re-scaling. This can be crucial in preventing conflicts or floating-range expansions.

6 Environment Signals and Constraints

6.1 Signal Sets

- **Symbol:** $\mathrm{SignalSet}(e)$.
- Each environment state e returns a finite set of signals or constraints.

Implementation:

```
SignalSet = {
    "envA": {"signalA", "signalB"},
    "envB": {"signalX", "signalY", "signalZ"}
}
```

Explanation:
This lookup returns a finite set of constraints for each environment. In code, it is used to determine which patterns remain valid.

6.2 Compatibility of Patterns

- **Predicate:** $\mathrm{Compat}(x, e)$.
- **Meaning:** "Pattern x is consistent with environment e."

Often coded as:

$$\mathrm{Compat}(x, e) \;\leftrightarrow\; \Big(\forall g : \mathrm{FeatureOf}(x, g) \rightarrow g \in \mathrm{SignalSet}(e)\Big).$$

Explanation:
In code, this function checks if every feature of x is within $\mathrm{SignalSet}(e)$. It's effectively a filter for deciding which patterns apply in the current environment.

6.3 State Changes

- If environment transitions from e to e', $\mathrm{SignalSet}(e)$ changes to $\mathrm{SignalSet}(e')$.
- Patterns with nonzero measure re-check $\mathrm{Compat}(x, e')$.

Implementation:

```
def update_environment(old_e, new_e, Mu, capacity):
    for x in all_patterns:
        k = Mu[(current_a, old_e, x)]
        if not Compat(x, new_e, feature_map):
            Mu[(current_a, new_e, x)] = 0
        else:
            Mu[(current_a, new_e, x)] = k
```

Explanation:
Whenever the environment changes, we re-validate patterns. This mirrors event-driven approaches, re-checking conditions upon each state update.

7 Update Mechanisms

7.1 Update Functions U_μ and U_p

We define functions that map probabilities or feature distributions from one joint state (a, e) to another (a', e'). For instance:

$$\mathbf{U}_\mu : \{0, \tfrac{1}{N(a)}, \ldots, 1\} \times \mathrm{ObsState} \times \mathrm{EnvState} \times \mathrm{ObsState} \times \mathrm{EnvState} \to \{0, \tfrac{1}{N(a')}, \ldots, 1\}.$$

These functions describe how probabilities transform as both observer and environment change, preserving grains-coded increments rather than continuous intervals.

Explanation:
When environment or observer states change, the system re-checks patterns. This parallels event-driven logic.

IT Implementation:
For instance, if (a, e) transitions to (a', e'), one might do:

```
def grains_transition(old_val, old_a, new_a):
    if new_a_capacity % old_a_capacity != 0:
        raise ValueError("Incompatible capacities")
    factor = new_a_capacity // old_a_capacity
    new_val = old_val * factor
    return new_val
```

By performing integer arithmetic, we scale probabilities from one capacity to the next while staying discrete.

7.2 Mapping Between Increments

$$\forall \alpha \in \left\{ \frac{k}{N(a)} \right\} : \mathbf{U}_\mu(\alpha, a, e, a', e') \ \in \ \left\{ \frac{k'}{N(a')} : k' = 0, \ldots, N(a') \right\}.$$

No real-number or partial fraction issues—only integer transformations.

7.3 Finite Step Updates

Given that $\mathrm{ObsState}, \mathrm{EnvState}, \mathrm{Pattern}, \mathrm{Feature}$ are all finite, each update step is local and discrete—no infinite recursion or limit approach. The grains-coded logic ensures every operation remains integer-based.

Every update is a finite, local operation. In code, it means no risk of non-termination; each update runs over a known, bounded set of values.

8 Summary of Section 1

By defining:

1. Finite sorts $\mathrm{ObsState}, \mathrm{EnvState}, \mathrm{Pattern}, \mathrm{Feature}$,
2. Function symbols $\mu, p_x, N(a)$,
3. Predicates $\mathrm{Compat}(x, e), \mathrm{FeatureOf}(x, g)$,
4. Axiom schemas (A1–A3) restricting probabilities and features to grains-coded increments,

we create a finite framework for storing states, signals, patterns, and discrete probabilities. No infinite sets or real numbers—only integer-based denominators that can expand when capacity increases. This provides a blueprint for your data structures and algorithms, with each entity finite and each function operating on discrete values. Code typically uses strict type definitions and bounded loops for guaranteed feasibility.

Practical Code Illustrations:

1. **Storing Discrete Probabilities (grain_probability.py):**

```
Mu = {}

def set_probability(a, e, x, k, capacity):
    """
    Store grains-coded measure for pattern x at (a,e).
    k is the integer grains count out of 'capacity'.
    """
    Mu[(a,e,x)] = k

def get_probability(a, e, x, capacity):
    """
    Return grains-coded fraction k/capacity as a float for inspection
    or usage. Not recommended for critical logic to avoid floating issues,
    but fine for debugging.
    """
    k = Mu.get((a,e,x), 0)
    return k / capacity
```

Hence, "the measure of x at (a, e)" is $\frac{k}{\text{capacity}}$.

2. **Denominator Embedding (embed_grains_fraction(...)):**

```
def embed_grains_fraction(old_k, old_capacity, new_capacity):
    if new_capacity % old_capacity != 0:
        raise ValueError("new_capacity not a multiple of old_capacity")
    factor = new_capacity // old_capacity
    new_k = old_k * factor
    return new_k, new_capacity
```

   Demonstrates exact fraction scaling when refining capacity.

3. **Signals and Compatibility:**

```
SignalSet = {
    "envA": {"sig1", "sig2"},
    "envB": {"sigX"},
}

def is_compatible(x, e, feature_map):
    """
    Check if pattern x's features are in SignalSet[e].
    feature_map[x] = set of features for pattern x.
    """
    return feature_map[x].issubset(SignalSet[e])
```

4. **Local Updates:**

```
def update_distribution(a, old_e, new_e, capacity, grains_map):
    for x in grains_map:
        k = Mu.get((a, old_e, x), 0)
        if not is_compatible(x, new_e, grains_map):
            Mu[(a, new_e, x)] = 0
        else:
            Mu[(a, new_e, x)] = k
```

Takeaways:

- The grains-coded approach ensures that each distribution or feature measure is an integer k out of capacity.
- All environment or observer state transitions remain purely discrete, referencing a known capacity.

Next Steps:

- Section 2 introduces stability and geometric notions (distance, loops) in grains-coded form.
- Section 3 covers capacity refinement from $N(a)$ to $N(a')$ exactly, referencing the embedding snippet.
- Additional sections tackle algebraic structures (fields, vector spaces), topological ideas (loops, polygons), error measures, and finite complexity bounds.

With Section 1 in place, you have a clear definition of states, patterns, capacities, and grains-coded probabilities, as well as a glimpse of how to store and update them in code. This foundation supports all further grains-coded expansions and advanced geometry.