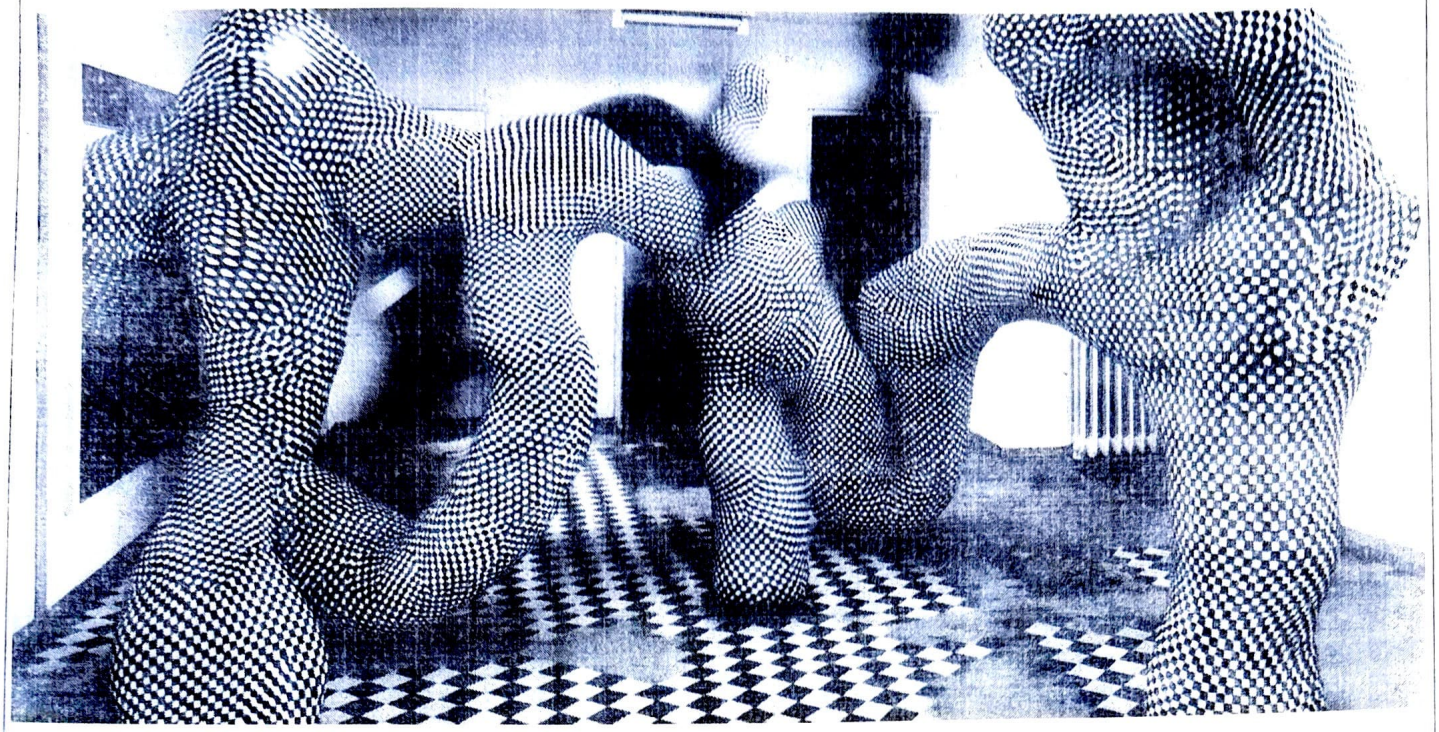
Summary of the mathematical component



A Formalized Framework for Finite, Resource-Bound Mathematics

Abstract

We present a formalized mathematical framework that reorients mathematics away from infinity-based abstractions toward a finite, resource-bound approach grounded in observer capacity. The framework is constructed in four integrated phases and implemented in the Coq proof assistant. This approach provides a constructive demonstration that mathematics can function coherently without assuming infinite sets, infinite precision, or perfect continuity. Instead, mathematical structures emerge from the fundamental constraints of finite observation and thermodynamic limitations on information processing.

Introduction

Traditional mathematics relies heavily on infinity-based abstractions: infinite sets, infinitely divisible continua, and infinite precision. Yet these abstractions stand in tension with the finite reality of physical observation, computation, and cognition. Our framework addresses this tension by constructively building a mathematical system that embraces finitude as a fundamental constraint rather than an approximation of an ideal infinite reality.

The four phases of our framework progressively build from basic ontological constraints to emergent geometric structures:

1. **Core Observer-Environment Framework**: Establishes finite state spaces and capacity-bound observation

When an observer transitions to a state with higher capacity, we

2. **Group Structure and Capacity**: Constructs number systems without assuming natural numbers or infinite sets
3. **Approximate Knowledge**: Formalizes probability and pattern recognition within capacity constraints
4. **Emergent Geometry**: Demonstrates how geometric structures emerge from stable probability patterns

Each phase has been formally implemented in the Coq proof assistant, ensuring logical consistency and constructive validity.

Phase 1: Core Observer-Environment Framework

I. Ontological Foundations

The framework begins with two basic types representing observer states and environment states:

```
Parameter ObsState : Type.
Parameter EnvState : Type.
```

We axiomatically assert the finiteness of both state spaces:

```
Axiom finite_ObsState : exists l : list ObsState,
  NoDup l /\ (forall a : ObsState, In a l).

Axiom finite_EnvState : exists l : list EnvState,
  NoDup l /\ (forall e : EnvState, In e l).
```

This finiteness axiom rejects the concept of infinite state spaces, encoding a fundamental constraint on possible observations.

II. Capacity Function

Each observer state has an associated capacity - a rational number representing its measurement resolution:

```
Parameter capacity : ObsState -> Q.
Axiom capacity_lower_bound : forall a : ObsState, Qle 1 (capacity a).
```

This capacity is bounded from above by a global parameter Ω:

```
Parameter Omega : Q.
Axiom Omega_positive : Qlt 0 Omega.
Axiom capacity_upper_bound : forall a : ObsState, Qle (capacity a) Omega.
```

These constraints encode the principle that observation always has finite resolution.

III. Capacity Refinement and Thermodynamic Constraints

When an observer transitions to a state with higher capacity, we model this as a multiplicative refinement:

```
coq
Copy
Axiom capacity_refinement_axiom :
  forall a a' : ObsState,
    Qlt (capacity a) (capacity a') ->
    capacity a' = capacity a * (capacity_refinement_factor a a') /\
    Qle 1 (capacity_refinement_factor a a').
```

This refinement process consumes resources:

```
coq
Copy
Parameter consume : ObsState -> Q -> ObsState.

Axiom consumption_for_refinement : forall (a : ObsState) (r : Q),
  Qlt 0 r ->
  Qlt (capacity a) (capacity (consume a r)) ->
  exists (cost : Q),
    Qlt 0 cost /\
    Qle cost r.
```

These axioms encode thermodynamic constraints directly into the mathematical framework: increasing measurement precision always comes at a resource cost.

Phase 2: Group Structure and Capacity

IV. Exponent Representation

Instead of assuming natural numbers as primitives, we define a mapping from integers to the bounded interval (0,1):

```
Parameter X : Z -> Q.
Axiom X_in_interval : forall r : Z, Qlt 0 (X r) /\ Qlt (X r) 1.
Axiom X_zero : X 0 = x0.
```

This mapping is defined recursively through successor and predecessor relations:

```
Axiom X_succ : forall r : Z, X (r+1) = X r + f * (1 - X r).
Axiom X_pred : forall r : Z, X (r-1) = 1 - ((1 - X r) / (1 - f)).
```

The parameter f (between 0 and 1) controls the rate at which successor steps approach 1, encoding the principle of diminishing returns for increased precision.

V. Connection to Observer Capacity

Each exponent r is associated with an observer state:

```
Parameter state_of_exponent : Z -> ObsState.
Definition C (r : Z) : Q := capacity (state_of_exponent r).
```

The capacity increases monotonically with the exponent, and each successor step increases capacity by a factor related to f:

```
Axiom capacity_monotonic : forall r s : Z,
  (r < s)%Z -> Qlt (C r) (C s).

Axiom succ_capacity_relation : forall r : Z,
  C (r+1) = Qmult (C r) (1 / (1 - f)).
```

This establishes a direct connection between the successor function and capacity refinement.

VI. Group Operations

We define addition and inverse operations that respect the exponent representation:

```
Definition phi (x : Q) : Q := (1 - x) / c.
Definition S_add (x y : Q) : Q := 1 - c * (phi x * phi y).
Definition S_inv (x : Q) : Q := 1 - c / (phi x).

Axiom S_add_exponents : forall r s : Z, S_add (X r) (X s) = X (r + s)%Z.
Axiom S_inv_exponent : forall r : Z, S_inv (X r) = X (-r)%Z.
```

These operations form a group structure without assuming infinite sets or natural numbers.

VI. Thermodynamic Constraints

Each successor operation consumes resources:

```
Axiom succ_consumes_resource : forall r : Z,
  exists cost : Q,
    Qlt 0 cost /\
    state_of_exponent (r+1)%Z = consume (state_of_exponent r) cost.
```

Group operations preserve bounded capacity:

```
Axiom addition_preserves_bounds_r : forall r s : Z,
  Qle (C (r + s)%Z) (C r).

Axiom addition_preserves_bounds_s : forall r s : Z,
  Qle (C (r + s)%Z) (C s).
```

These axioms ensure that mathematical operations respect thermodynamic constraints.

Definition feature_distance (a : ObsState) (e : EnvState) (x y : Pattern) : Q :=

Phase 3: Approximate Knowledge

VII. Patterns and Features

We introduce finite sets of patterns and features that observers can recognize:

```
Parameter Pattern : Type.
Parameter Feature : Type.
Parameter features : Pattern -> list Feature.

Axiom finite_Pattern : exists l : list Pattern,
  NoDup l /\ (forall x : Pattern, In x l).

Axiom finite_Features : forall x : Pattern,
  NoDup (features x) /\ (features x <> nil).
```

VIII. Probability Assignments

Observers assign probabilities to patterns and features, bounded by their capacity:

```
Parameter mu : ObsState -> EnvState -> Pattern -> Q.
Parameter p : Pattern -> ObsState -> EnvState -> Feature -> Q.

Axiom mu_range : forall a e x,
  Qle 0 (mu a e x) /\ Qle (mu a e x) (capacity a).

Axiom p_sum_1 : forall x a e,
  fold_right Qplus 0 (map (p x a e) (features x)) = 1.
```

The precision of probability assignments is limited by capacity:

```
Axiom probability_precision : forall a e x,
  exists n d : Z,
    Qlt 0 (inject_Z d) /\
    inject_Z d = capacity a /\
    mu a e x = (inject_Z n) / (inject_Z d).
```

This axiom enforces that probabilities have finite precision proportional to observer capacity.

IX. Information Processing Costs

When observer states change, probability assignments are updated through functions that preserve the appropriate ranges:

```
Parameter U_mu : ObsState -> EnvState -> ObsState -> EnvState -> Q -> Q.
Parameter U_p : Pattern -> ObsState -> EnvState -> ObsState -> EnvState -> Feature -> Q -> Q.
```

```
Axiom update_consistency_mu : forall a e a' e' x,
 mu a' e' x = U_mu a e a' e' (mu a e x).
```

These updates consume resources, especially when increasing precision:

```
Axiom transition_info_cost : forall a e a' e',
 Qlt 0 (info_cost a e a' e') ->
 a' = consume a (info_cost a e a' e').

Axiom precision_cost : forall a e a' e' x,
 Qlt (capacity a) (capacity a') ->
 Qlt (mu a e x) (mu a' e' x) ->
 Qlt 0 (info_cost a e a' e').
```

These axioms enforce thermodynamic constraints on information processing.

X. Stability and Approximate Knowledge

We define stability as approximate consistency of probability assignments within a region:

```
Definition Stable (x : Pattern) : Prop :=
 exists a0 : ObsState, exists e0 : EnvState,
  InRegion (a0, e0) /\
  (forall a e, InRegion (a, e) ->
   (Qle (mu a e x - mu a0 e0 x) eps_mu \/ Qle (mu a0 e0 x - mu a e x) eps_mu) /\
   (forall g : Feature, In g (features x) ->
    (Qle (p x a e g - p x a0 e0 g) eps_p \/ Qle (p x a0 e0 g - p x a e g) eps_p))).
```

Observers with finite capacity can only distinguish probabilities above a certain threshold:

```
Axiom approximate_knowledge : forall a x,
 exists delta : Q,
  Qlt 0 delta /\
  Qle delta (1 / (capacity a)) /\
  forall e e',
   mu a e x = mu a e' x \/
   Qle delta (mu a e x - mu a e' x) \/
   Qle delta (mu a e' x - mu a e x).
```

This enforces the principle that all knowledge is approximate within finite capacity constraints.

Phase 4: Emergent Geometry

XI. Probabilistic Distance Measures

We define distance measures based on probability differences:

```
Definition feature_distance (a : ObsState) (e : EnvState) (x y : Pattern) : Q :=
  fold_right Qplus 0
    (map (fun g =>
      match (In_dec Feature g (features x)), (In_dec Feature g (features y)) with
      | left proof_x, left proof_y => q_abs (p x a e g - p y a e g)
      | left proof_x, right _ => p x a e g
      | right _, left proof_y => p y a e g
      | right _, right _ => 0
      end)
    (app (features x) (features y))).
```

We associate patterns with geometric points and define Euclidean distance:

```
Parameter geometric_points : Pattern -> list (Q * Q).

Definition euclidean_distance (p1 p2 : Q * Q) : Q :=
  let '(x1, y1) := p1 in
  let '(x2, y2) := p2 in
  sqrt ((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2)).
```

XII. Geometric Granularity and Distinguishability

Each observer state has a granularity threshold below which points are indistinguishable:

```
Parameter delta_geometric : ObsState -> Q.

Axiom granularity_capacity : forall a : ObsState,
  delta_geometric a = 1 / (capacity a).

Definition distinguishable (a : ObsState) (p1 p2 : Q * Q) : Prop :=
  euclidean_distance p1 p2 > delta_geometric a.
```

Higher capacity leads to finer geometric resolution:

```
Axiom capacity_resolution : forall a1 a2 : ObsState,
  Qlt (capacity a1) (capacity a2) ->
  Qlt (delta_geometric a2) (delta_geometric a1).
```

XIII. Emergent Lines and Shapes

Lines emerge as sequences of distinguishable points:

```
Definition is_line (pts : list (Q * Q)) (a : ObsState) : Prop :=
  nat_to_Q (length pts) >= 2 /\
  forall i, (i < length pts - 1)%nat ->
    let p1 := nth i pts (0, 0) in
    let p2 := nth (i+1)%nat pts (0, 0) in
    distinguishable a p1 p2 /\
    forall p,
```

```
    ~(In p pts) ->
    ~(euclidean_distance p1 p < euclidean_distance p1 p2 /\
      euclidean_distance p p2 < euclidean_distance p1 p2).

Polygons emerge as closed loops of lines:
Definition is_polygon (pts : list (Q * Q)) (a : ObsState) : Prop :=
  nat_to_Q (length pts) >= 3 /\
  is_line (pts ++ (firstn 1 pts)) a.

Higher capacity observers can perceive more detailed geometric structures:

Axiom geometric_precision : forall a : ObsState, forall pts : list (Q * Q),
  is_line pts a ->
  forall a', Qlt (capacity a) (capacity a') ->
    exists pts', is_line pts' a' /\
      (length pts' >= length pts)%nat.
```

XIV. Geometry from Stability

```
Stable patterns give rise to geometric structures:
Axiom geometry_from_stability : forall x : Pattern,
  Stable x ->
  exists pts : list (Q * Q),
    forall a e, InRegion (a, e) ->
      fold_right Qplus 0 (map (point_probability a e) pts) > 1/2 * (capacity a).
```

This axiom establishes that geometric structures emerge from stable probabilistic patterns rather than being fundamental.

Philosophical Significance

This framework represents a paradigm shift in mathematical foundations:

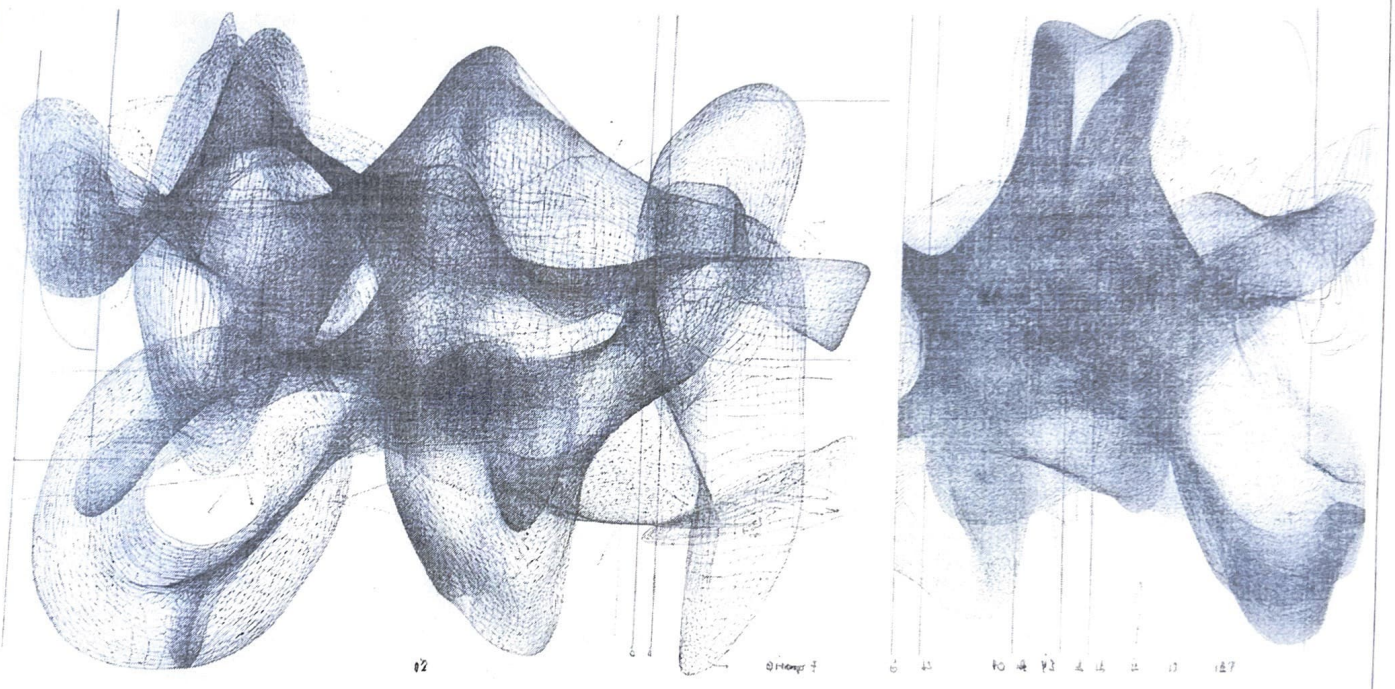
1. **Rejection of Infinity**: The entire system operates without assuming infinite sets, infinite precision, or perfect continuity.
2. **Observer Dependence**: Mathematical structures are defined relative to observer capacity, not as absolute abstractions.
3. **Thermodynamic Constraints**: Information processing and increased precision always consume finite resources.
4. **Approximate Knowledge**: All knowledge is probabilistic and limited by capacity constraints.
5. **Emergent Structures**: Higher-level mathematical concepts (like geometry) emerge from stable patterns rather than being fundamental.

Conclusion

Our formalized framework demonstrates that mathematics can function coherently without assuming infinity as a primitive concept. By embracing the constraints of finite observation and thermodynamic limitations, we construct a mathematical system that may better align with physical reality while avoiding the paradoxes that arise from assuming infinity.

The implementation in Coq provides a constructive demonstration that this approach is logically consistent and can support the essential structures of mathematics: group operations, probability theory, and geometry all emerge naturally from the basic constraints of finite observation.

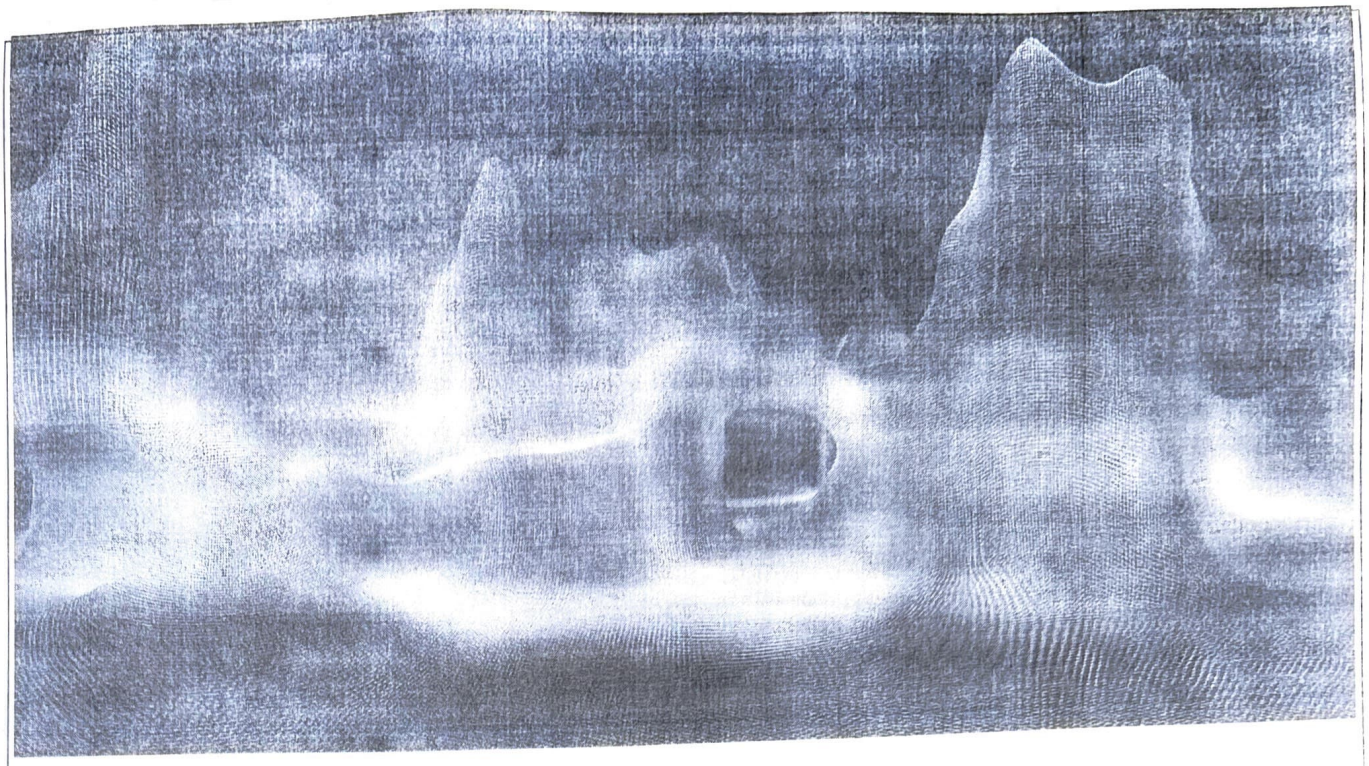
This work opens possibilities for recasting other areas of mathematics within a finite, resource-bound framework, potentially offering new perspectives on long-standing mathematical and philosophical questions about the nature of infinity, continuity, and mathematical truth.



```
(*****************************************************************************)
(* phase1_core_framework.v                                                   *)
(* Core Observer-Environment Framework                                       *)
(*                                                                           *)
(* This module establishes the foundational axioms for a finite,             *)
(* resource-bound mathematical framework. It defines observer states,        *)
(* environment states, capacity functions, and the constraints that          *)
(* govern their interactions.                                                *)
(*****************************************************************************)

Require Import Coq.QArith.QArith.
Require Import Coq.Lists.List.
Import ListNotations.
```

```
(* We work in the rational number scope for capacity values *)
Open Scope Q_scope.

Module Phase1_Core_Framework.

(*****************************************************************************)
(* BASIC TYPE DEFINITIONS                                                    *)
(*****************************************************************************)

(* Observer states and environment states are defined as parameters *)
Parameter ObsState : Type.
Parameter EnvState : Type.

(* We define the joint state as a product type *)
Definition JointState := (ObsState * EnvState)%type.

(*****************************************************************************)
(* FINITENESS AXIOMS                                                         *)
(*****************************************************************************)

(* Axioms asserting that ObsState and EnvState are finite sets *)
Axiom finite_ObsState : exists (l : list ObsState),
  NoDup l /\ (forall x : ObsState, In x l).

Axiom finite_EnvState : exists (l : list EnvState),
  NoDup l /\ (forall e : EnvState, In e l).

(*****************************************************************************)
(* CAPACITY FUNCTION                                                         *)
(*****************************************************************************)

(* Each observer state has a capacity - a measure of its resolution *)
Parameter capacity : ObsState -> Q.

(* Global upper bound on capacity *)
Parameter Omega : Q.

(* Capacity is always positive and bounded *)
Axiom capacity_positive : forall a : ObsState, Qlt 0 (capacity a).
Axiom capacity_lower_bound : forall a : ObsState, Qle 1 (capacity a).
Axiom capacity_upper_bound : forall a : ObsState, Qle (capacity a) Omega.
Axiom Omega_positive : Qlt 0 Omega.
```

```
(*****************************************************************************)
(* STATE TRANSITIONS AND CAPACITY REFINEMENT                                 *)
(*****************************************************************************)

(* Capacity refinement: if capacity increases, it's by a multiplicative factor *)
Axiom capacity_refinement : forall (a a' : ObsState) (M : Q),
  capacity a' = Qmult (capacity a) M ->
  Qlt (capacity a) (capacity a') ->
  Qle 1 M.

(* Capacity and state transitions *)
Parameter transition : JointState -> JointState -> Prop.

(* Transition consumes resources - models thermodynamic constraints *)
Axiom transition_cost : forall (js js' : JointState),
  transition js js' ->
  Qle (capacity (fst js')) (capacity (fst js)).

(* Special case: capacity can increase, but requires consumption *)
Parameter consume : ObsState -> Q -> ObsState.

(* Consuming resources to increase capacity *)
Axiom consumption_for_refinement : forall (a : ObsState) (r : Q),
  Qlt 0 r ->
  Qlt (capacity a) (capacity (consume a r)) ->
  exists (cost : Q),
    Qlt 0 cost /\
    Qle cost r.

(*****************************************************************************)
(* REGION AND STABILITY                                                      *)
(*****************************************************************************)

(* A region is a subset of the joint state space *)
Parameter InRegion : JointState -> Prop.

(* Stability parameters - small positive rationals *)
Parameter eps : Q.
Axiom eps_positive : Qlt 0 eps.

(* Stability notion will be expanded in later phases *)

End Phase1_Core_Framework.
```



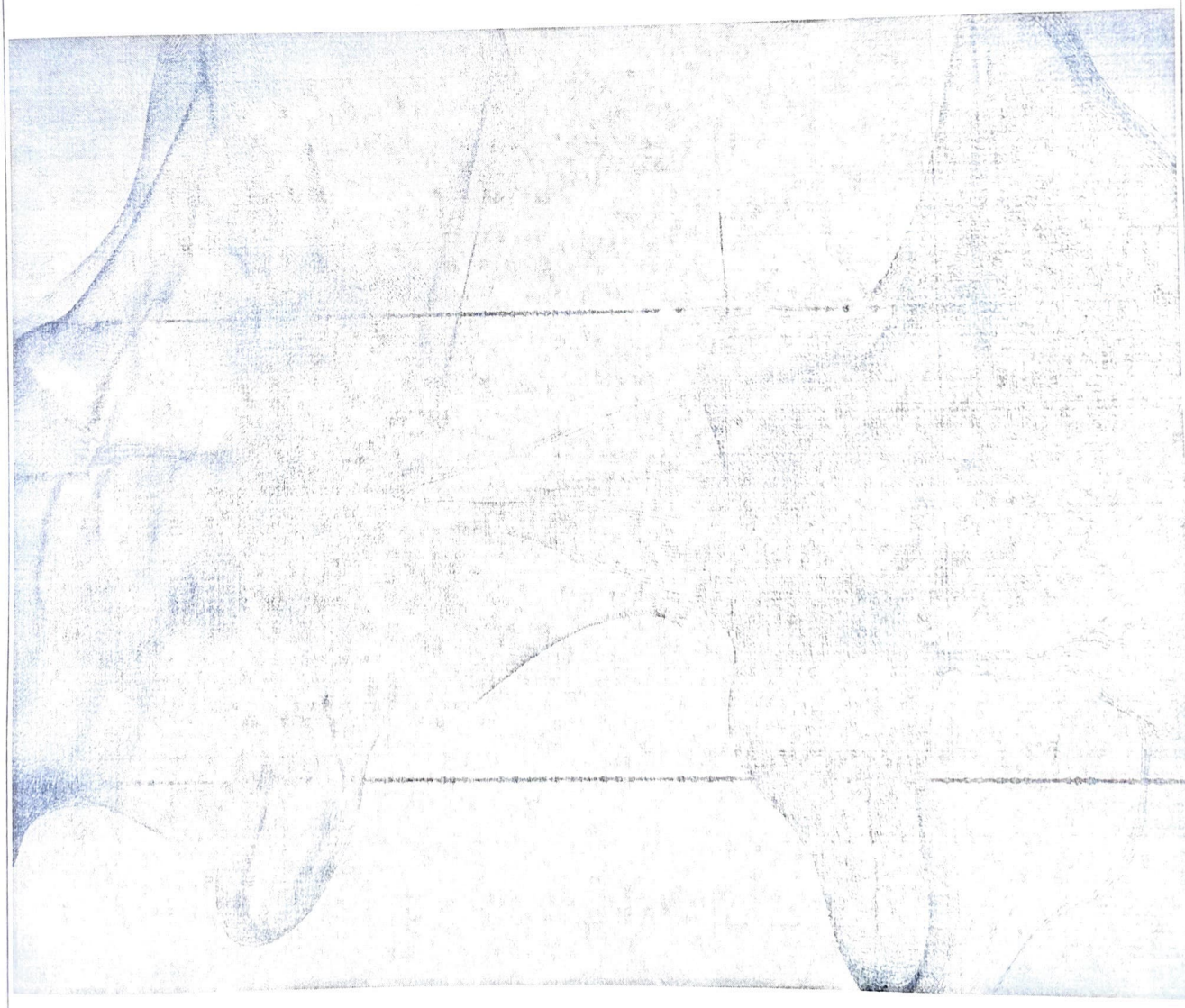
```
(*****************************************************************************)
(* phase2_group_capacity.v                                                   *)
(* Connecting Group Structure to Observer Capacity                           *)
(*                                                                           *)
(* This module establishes the connection between the exponent-based         *)
(* group structure and the observer capacity framework.                      *)
(* It shows how successor operations relate to capacity consumption and      *)
(* how the group structure preserves the finite, bounded nature of the system *)
(*****************************************************************************)

Require Import Coq.QArith.QArith.
Require Import Coq.ZArith.ZArith.
Require Import Coq.Lists.List.
Import ListNotations.

Open Scope Q_scope.

Module Phase2_Group_Capacity.

(*****************************************************************************)
(* CORE FRAMEWORK FROM PHASE 1 (Included to make this file self-contained)   *)
(*****************************************************************************)

(* Observer states and environment states *)
Parameter ObsState : Type.
Parameter EnvState : Type.
```

```
(* Joint state as a product type *)
Definition JointState := (ObsState * EnvState)%type.

(* Finiteness axioms *)
Axiom finite_ObsState : exists (l : list ObsState),
  NoDup l /\ (forall x : ObsState, In x l).

Axiom finite_EnvState : exists (l : list EnvState),
  NoDup l /\ (forall e : EnvState, In e l).

(* Capacity function and bounds *)
Parameter capacity : ObsState -> Q.
Parameter Omega : Q.

Axiom capacity_positive : forall a : ObsState, Qlt 0 (capacity a).
Axiom capacity_lower_bound : forall a : ObsState, Qle 1 (capacity a).
Axiom capacity_upper_bound : forall a : ObsState, Qle (capacity a) Omega.
Axiom Omega_positive : Qlt 0 Omega.

(* Resource consumption *)
Parameter consume : ObsState -> Q -> ObsState.

Axiom consumption_for_refinement : forall (a : ObsState) (r : Q),
  Qlt 0 r ->
  Qlt (capacity a) (capacity (consume a r)) ->
  exists (cost : Q),
    Qlt 0 cost /\
    Qle cost r.

(*****************************************************************************)
(* EXPONENT REPRESENTATION PARAMETERS                                        *)
(*****************************************************************************)

(* Fixed increment fraction f, with 0 < f < 1 *)
Parameter f : Q.
Axiom f_range : Qlt 0 f /\ Qlt f 1.

(* Initial value x0 in (0,1), which serves as our neutral element *)
Parameter x0 : Q.
Axiom x0_range : Qlt 0 x0 /\ Qlt x0 1.

(× Define c = 1 - x0 *)
Definition c : Q := 1 - x0.
Axiom c_positive : Qlt 0 c.
```

```
Axiom c_less_than_1 : Qlt c 1.

(*****************************************************************************)
(* EXPONENT REPRESENTATION FUNCTION                                          *)
(*****************************************************************************)

(* Map from integers (exponents) to rationals in (0,1) *)
Parameter X : Z -> Q.

(* X(r) is always in the open interval (0,1) *)
Axiom X_in_interval : forall r : Z, Qlt 0 (X r) /\ Qlt (X r) 1.

(× X(0) = x0 - our identity element *)
Axiom X_zero : X 0 = x0.

(* Recursive definition of X via successor relation *)
Axiom X_succ : forall r : Z, X (r+1) = X r + f * (1 - X r).

(* Recursive definition of X via predecessor relation *)
Axiom X_pred : forall r : Z, X (r-1) = 1 - ((1 - X r) / (1 - f)).

(*****************************************************************************)
(* MAPPING BETWEEN EXPONENTS AND OBSERVER STATES                             *)
(*****************************************************************************)

(* Function mapping exponents to observer states *)
Parameter state_of_exponent : Z -> ObsState.

(* Capacity function on exponents - shorthand for capacity(state_of_exponent(r)) *)
Definition C (r : Z) : Q := capacity (state_of_exponent r).

(* Capacity increases with exponent value *)
Axiom capacity_monotonic : forall r s : Z,
  (r < s)%Z -> Qlt (C r) (C s).

(* Capacity increases by a factor related to f when exponent increases by 1 *)
Axiom succ_capacity_relation : forall r : Z,
  C (r+1) = Qmult (C r) (1 / (1 - f)).

(* This relates our parameter f directly to capacity refinement *)

(*****************************************************************************)
(* GROUP OPERATIONS ON EXPONENTS                                             *)
(*****************************************************************************)
```

```
(* Phi transformation for addition *)
Definition phi (x : Q) : Q := (1 - x) / c.

(* Addition operation *)
Definition S_add (x y : Q) : Q := 1 - c * (phi x * phi y).

(* Addition corresponds to adding exponents *)
Axiom S_add_exponents : forall r s : Z, S_add (X r) (X s) = X (r + s)%Z.

(* Inverse operation *)
Definition S_inv (x : Q) : Q := 1 - c / (phi x).

(* Inverse corresponds to negating exponent *)
Axiom S_inv_exponent : forall r : Z, S_inv (X r) = X (-r)%Z.

(*****************************************************************************)
(* RESOURCE CONSUMPTION AND THERMODYNAMIC CONSTRAINTS                        *)
(*****************************************************************************)

(* Successor operation consumes capacity - implements thermodynamic constraint *)
Axiom succ_consumes_resource : forall r : Z,
 exists cost : Q,
  Qlt 0 cost /\
  state_of_exponent (r+1)%Z = consume (state_of_exponent r) cost.

(* Group operations preserve bounded capacity *)
Axiom addition_preserves_bounds_r : forall r s : Z,
 Qle (C (r + s)%Z) (C r).

Axiom addition_preserves_bounds_s : forall r s : Z,
 Qle (C (r + s)%Z) (C s).

(* These two axioms together replace the Qmax operation - showing that the
  combined capacity is bounded by either of the individual capacities *)

(*****************************************************************************)
(* PRECISION AND CAPACITY CONNECTION                                         *)
(*****************************************************************************)

(* The precision of discrimination between successive X values
  is related to capacity *)
Axiom precision_capacity : forall r : Z,
 (* X values always increase with exponents, so X(r+1) - X(r) is positive *)
```

```
Qle 0 (p x a e g) /\ Qle (p x a e g) 1.

X (r+1)%Z - X r = f × (1 - X r) /\
Qle (f * (1 - X r)) (1 / (C r)).

(* This axiom connects the discriminatory power of the observer to their capacity *)

(*****************************************************************************)
(* FINITE APPROXIMATION PROPERTIES                                           *)
(*****************************************************************************)

(* For any finite capacity, there is a maximum exponent that can be represented *)
Axiom finite_representation : forall a : ObsState,
 exists r_max : Z,
  forall s : Z,
   state_of_exponent s = a -> (s <= r_max)%Z.

(* This enforces that finite capacity implies bounded exponent range *)

End Phase2_Group_Capacity.
```



Qle 0 (p x a e g) /\ Qle (p x a e g) 1.

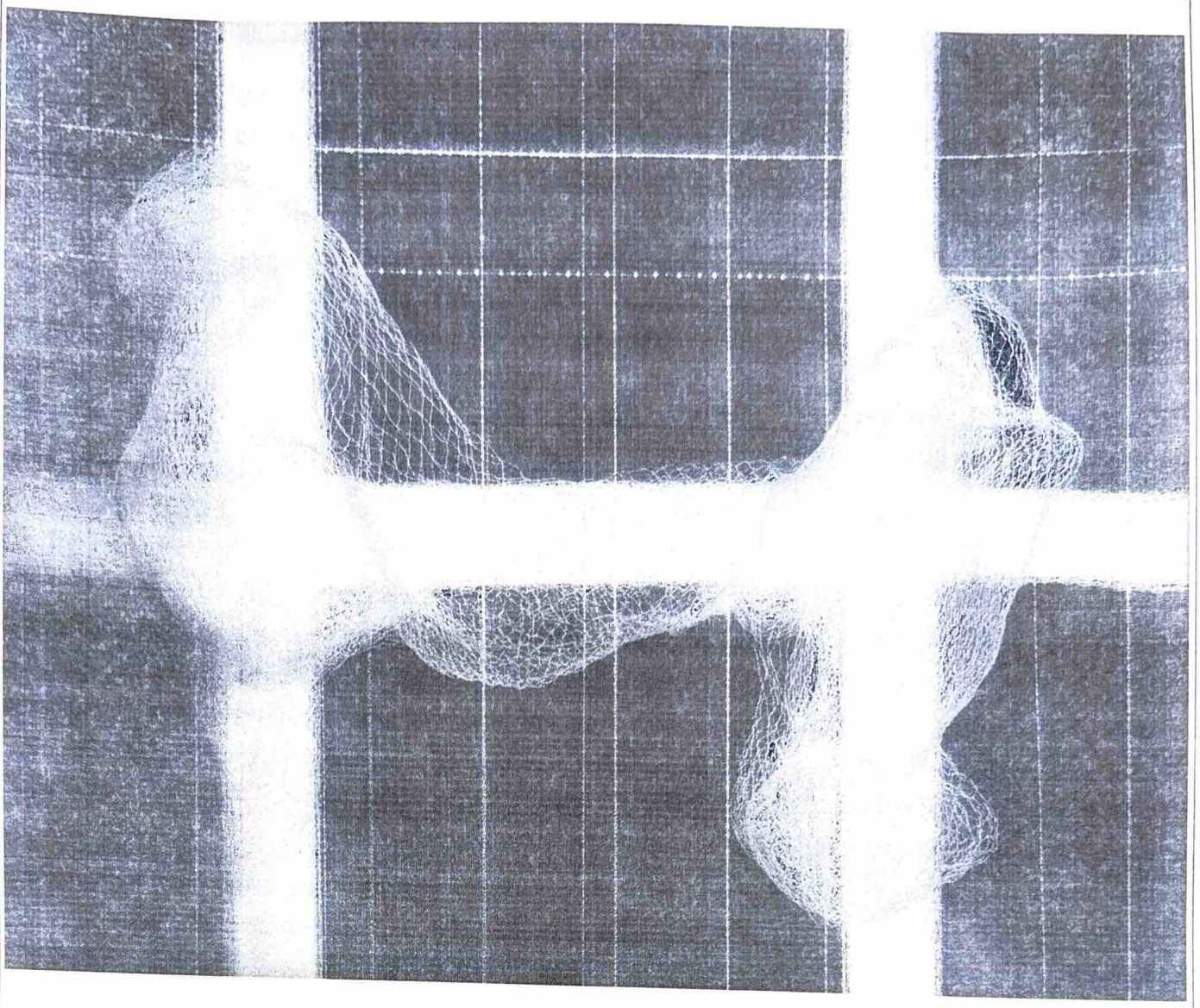
```
(*****************************************************************************)
(* phase3_approximate_knowledge.v                                            *)
(* Approximate Knowledge Through Patterns and Probabilities                  *)
(*                                                                           *)
(* This module formalizes the notion of approximate knowledge through        *)
(* patterns, features, and capacity-dependent probability assignments.       *)
(* It models how observers with finite capacity can only make approximate    *)
(* measurements, with information processing having resource costs.          *)
(*****************************************************************************)

Require Import Coq.QArith.QArith.
Require Import Coq.ZArith.ZArith.
Require Import Coq.Lists.List.
Import ListNotations.

Open Scope Q_scope.

Module Phase3_Approximate_Knowledge.

(*****************************************************************************)
(* CORE FRAMEWORK AND GROUP STRUCTURE (From Phases 1-2)                      *)
(*****************************************************************************)

(* Core types and capacity *)
Parameter ObsState : Type.
Parameter EnvState : Type.
Definition JointState := (ObsState * EnvState)%type.

Parameter capacity : ObsState -> Q.
Parameter Omega : Q.
Axiom capacity_positive : forall a : ObsState, Qlt 0 (capacity a).
Axiom capacity_lower_bound : forall a : ObsState, Qle 1 (capacity a).
Axiom capacity_upper_bound : forall a : ObsState, Qle (capacity a) Omega.

(* Resource consumption *)
Parameter consume : ObsState -> Q -> ObsState.

(* Group structure parameters *)
Parameter f : Q.
Axiom f_range : Qlt 0 f /\ Qlt f 1.
Parameter x0 : Q.
Axiom x0_range : Qlt 0 x0 /\ Qlt x0 1.
Definition c : Q := 1 - x0.
```

```
(* Exponent representation *)
Parameter X : Z -> Q.
Parameter state_of_exponent : Z -> ObsState.
Definition C (r : Z) : Q := capacity (state_of_exponent r).

(*****************************************************************************)
(* PATTERNS AND FEATURES                                                     *)
(*****************************************************************************)

(* Patterns are things the observer can recognize *)
Parameter Pattern : Type.

(* Each pattern has a set of features *)
Parameter Feature : Type.
Parameter features : Pattern -> list Feature.

(* Finiteness of patterns and features *)
Axiom finite_Pattern : exists l : list Pattern,
  NoDup l /\ (forall x : Pattern, In x l).

Axiom finite_Features : forall x : Pattern,
  NoDup (features x) /\ (features x <> nil).

(*****************************************************************************)
(* PROBABILITY ASSIGNMENTS                                                   *)
(*****************************************************************************)

(* Probability that a pattern is recognized in a given joint state *)
Parameter mu : ObsState -> EnvState -> Pattern -> Q.

(* Probability of a feature given a pattern in a joint state *)
Parameter p : Pattern -> ObsState -> EnvState -> Feature -> Q.

(* Probability assignments are bounded by capacity *)
Axiom mu_range : forall a e x,
  Qle 0 (mu a e x) /\ Qle (mu a e x) (capacity a).

(* Feature probabilities sum to 1 for each pattern *)
Axiom p_sum_1 : forall x a e,
  fold_right Qplus 0 (map (p x a e) (features x)) = 1.

(* Feature probabilities respect capacity limits *)
Axiom p_range : forall x a e g,
  In g (features x) ->
```

```
(* Exponent representation *)
Parameter X : Z -> Q.
Parameter state_of_exponent : Z -> ObsState.
Definition C (r : Z) : Q := capacity (state_of_exponent r).

(*****************************************************************************)
(* PATTERNS AND FEATURES                                                     *)
(*****************************************************************************)

(* Patterns are things the observer can recognize *)
Parameter Pattern : Type.

(* Each pattern has a set of features *)
Parameter Feature : Type.
Parameter features : Pattern -> list Feature.

(* Finiteness of patterns and features *)
Axiom finite_Pattern : exists l : list Pattern,
  NoDup l /\ (forall x : Pattern, In x l).

Axiom finite_Features : forall x : Pattern,
  NoDup (features x) /\ (features x <> nil).

(*****************************************************************************)
(* PROBABILITY ASSIGNMENTS                                                   *)
(*****************************************************************************)

(* Probability that a pattern is recognized in a given joint state *)
Parameter mu : ObsState -> EnvState -> Pattern -> Q.

(* Probability of a feature given a pattern in a joint state *)
Parameter p : Pattern -> ObsState -> EnvState -> Feature -> Q.

(* Probability assignments are bounded by capacity *)
Axiom mu_range : forall a e x,
  Qle 0 (mu a e x) /\ Qle (mu a e x) (capacity a).

(* Feature probabilities sum to 1 for each pattern *)
Axiom p_sum_1 : forall x a e,
  fold_right Qplus 0 (map (p x a e) (features x)) = 1.

(* Feature probabilities respect capacity limits *)
Axiom p_range : forall x a e g,
  In g (features x) ->
```

```
  Qle 0 (p x a e g) /\ Qle (p x a e g) 1.

(* Precision of probability assignments depends on capacity *)
Axiom probability_precision : forall a e x,
  exists n d : Z,
    Qlt 0 (inject_Z d) /\
    inject_Z d = capacity a /\
    mu a e x = (inject_Z n) / (inject_Z d).

(* This axiom states that probabilities are expressible as fractions
   with denominator equal to the capacity, limiting their precision *)

(*****************************************************************************)
(* UPDATE FUNCTIONS AND INFORMATION COSTS                                    *)
(*****************************************************************************)

(* Update function for pattern probability when state changes *)
Parameter U_mu : ObsState -> EnvState -> ObsState -> EnvState -> Q -> Q.

(* Update function for feature probability when state changes *)
Parameter U_p : Pattern -> ObsState -> EnvState -> ObsState -> EnvState -> Feature -> Q -> Q.

(* Updates preserve the probability ranges *)
Axiom U_mu_range : forall a e a' e' q,
  Qle 0 q -> Qle q (capacity a) ->
  Qle 0 (U_mu a e a' e' q) /\ Qle (U_mu a e a' e' q) (capacity a').

Axiom U_p_range : forall x a e a' e' g q,
  Qle 0 q -> Qle q 1 ->
  Qle 0 (U_p x a e a' e' g q) /\ Qle (U_p x a e a' e' g q) 1.

(* Update consistency - probabilities after transition *)
Axiom update_consistency_mu : forall a e a' e' x,
  mu a' e' x = U_mu a e a' e' (mu a e x).

Axiom update_consistency_p : forall x a e a' e' g,
  In g (features x) ->
  p x a' e' g = U_p x a e a' e' g (p x a e g).

(* Information processing cost *)
Parameter info_cost : ObsState -> EnvState -> ObsState -> EnvState -> Q.

(* State transitions have information processing costs *)
Axiom transition_info_cost : forall a e a' e',
```

```
  Qlt 0 (info_cost a e a' e') ->
  a' = consume a (info_cost a e a' e').

(* Higher information costs for more precise updates *)
Axiom precision_cost : forall a e a' e' x,
  Qlt (capacity a) (capacity a') ->
  Qlt (mu a e x) (mu a' e' x) ->
  Qlt 0 (info_cost a e a' e').

(*****************************************************************************)
(* STABILITY AND APPROXIMATE KNOWLEDGE                                       *)
(*****************************************************************************)

(* A region is a subset of the joint state space *)
Parameter InRegion : JointState -> Prop.

(* Small positive thresholds for stability *)
Parameter eps_mu : Q.
Parameter eps_p : Q.
Axiom eps_positive : Qlt 0 eps_mu /\ Qlt 0 eps_p.

(* Stability means probabilities stay approximately constant within a region *)
Definition Stable (x : Pattern) : Prop :=
  exists a0 : ObsState, exists e0 : EnvState,
    InRegion (a0, e0) /\
    (forall a e, InRegion (a, e) ->
      (Qle (mu a e x - mu a0 e0 x) eps_mu \/ Qle (mu a0 e0 x - mu a e x) eps_mu) /\
      (forall g : Feature, In g (features x) ->
        (Qle (p x a e g - p x a0 e0 g) eps_p \/ Qle (p x a0 e0 g - p x a e g) eps_p))).

(* Approximate knowledge: limited precision on probability judgments *)
Axiom approximate_knowledge : forall a x,
  exists delta : Q,
    Qlt 0 delta /\
    Qle delta (1 / (capacity a)) /\
    forall e e',
      mu a e x = mu a e' x \/
      Qle delta (mu a e x - mu a e' x) \/
      Qle delta (mu a e' x - mu a e x).

(* This axiom formalizes that observers with finite capacity can only
   distinguish probabilities above a certain threshold delta *)

End Phase3_Approximate_Knowledge.
```



```
(*****************************************************************************)
(* phase4_emergent_geometry.v                                                *)
(* Emergent Geometry from Probabilistic Patterns                             *)
(*                                                                           *)
(* This module shows how geometric concepts emerge from the probabilistic    *)
(* framework established in previous phases. It defines distance measures    *)
(* based on probability differences and establishes stability conditions     *)
(* that lead to coherent geometric structures.                               *)
(*****************************************************************************)

Require Import Coq.QArith.QArith.
Require Import Coq.ZArith.ZArith.
Require Import Coq.Lists.List.
Require Import Coq.Arith.Arith.
Import ListNotations.

Open Scope Q_scope.
```

```
Module Phase4_Emergent_Geometry.

(*****************************************************************************)
(* CORE FRAMEWORK FROM PREVIOUS PHASES                                       *)
(*****************************************************************************)

(* Core types and capacity *)
Parameter ObsState : Type.
Parameter EnvState : Type.
Definition JointState := (ObsState * EnvState)%type.

Parameter capacity : ObsState -> Q.
Parameter Omega : Q.
Axiom capacity_positive : forall a : ObsState, Qlt 0 (capacity a).

(* Patterns and features *)
Parameter Pattern : Type.
Parameter Feature : Type.
Parameter features : Pattern -> list Feature.

(* Probability assignments *)
Parameter mu : ObsState -> EnvState -> Pattern -> Q.
Parameter p : Pattern -> ObsState -> EnvState -> Feature -> Q.

(* Stability region *)
Parameter InRegion : JointState -> Prop.
Parameter eps_mu : Q.
Parameter eps_p : Q.
Axiom eps_positive : Qlt 0 eps_mu /\ Qlt 0 eps_p.

(* Stability definition from Phase 3 *)
Definition Stable (x : Pattern) : Prop :=
  exists a0 : ObsState, exists e0 : EnvState,
    InRegion (a0, e0) /\
    (forall a e, InRegion (a, e) ->
      (Qle (mu a e x - mu a0 e0 x) eps_mu \/ Qle (mu a0 e0 x - mu a e x) eps_mu) /\
      (forall g : Feature, In g (features x) ->
        (Qle (p x a e g - p x a0 e0 g) eps_p \/ Qle (p x a0 e0 g - p x a e g) eps_p))).

(*****************************************************************************)
(* ABSOLUTE VALUE AND UTILITY FUNCTIONS                                      *)
(*****************************************************************************)

(* Define our own absolute value function since Qabs is not available *)
```

```
Definition q_abs (q : Q) : Q :=
  if Qlt_le_dec q 0 then -q else q.

(* Function to get all patterns - we'll define it as a parameter *)
Parameter get_all_patterns : list Pattern.
Axiom get_all_patterns_complete : forall x : Pattern, In x get_all_patterns.

(* Helper function for feature membership test *)
Parameter In_dec : forall (A : Type), forall (a : A) (l : list A), {In a l} + {~ In a l}.

(* Define sqrt as a parameter since we don't have direct access to it *)
Parameter sqrt : Q -> Q.
Axiom sqrt_properties : forall q : Q,
  Qle 0 q -> Qle 0 (sqrt q) /\ Qmult (sqrt q) (sqrt q) = q.

(* Convert nat to Q for comparisons *)
Definition nat_to_Q (n : nat) : Q := inject_Z (Z.of_nat n).

(*****************************************************************************)
(* PROBABILISTIC DISTANCE MEASURES                                           *)
(*****************************************************************************)

(* Feature-based distance between patterns *)
Definition feature_distance (a : ObsState) (e : EnvState) (x y : Pattern) : Q :=
  fold_right Qplus 0
    (map (fun g =>
      match (In_dec Feature g (features x)), (In_dec Feature g (features y)) with
      | left proof_x, left proof_y => q_abs (p x a e g - p y a e g)
      | left proof_x, right _ => p x a e g
      | right _, left proof_y => p y a e g
      | right _, right _ => 0
      end)
    (app (features x) (features y))).

(* Distance between joint states based on pattern probabilities *)
Definition state_distance (js1 js2 : JointState) : Q :=
  let '(a1, e1) := js1 in
  let '(a2, e2) := js2 in
  fold_right Qplus 0
    (map (fun x => q_abs (mu a1 e1 x - mu a2 e2 x))
      get_all_patterns).

(* Point-based geometric distance *)
Parameter geometric_points : Pattern -> list (Q * Q).
```

```
delta_geometric a − 1 / (...
Parameter reference_observer : ObsState.
Parameter reference_environment : EnvState.

(* Euclidean distance between points *)
Definition euclidean_distance (p1 p2 : Q * Q) : Q :=
 let '(x1, y1) := p1 in
 let '(x2, y2) := p2 in
 sqrt ((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2)).

(* Helper function to get minimum of a list with a default value *)
Fixpoint list_min (l : list Q) (default : Q) : Q :=
 match l with
 | nil => default
 | h :: t => if Qlt_le_dec h (list_min t default) then h else list_min t default
 end.

(* Flatten a list of lists *)
Fixpoint flatten {A : Type} (l : list (list A)) : list A :=
 match l with
 | nil => nil
 | h :: t => h ++ flatten t
 end.

(* Map a function over a list and flatten the result *)
Definition flat_map {A B : Type} (f : A -> list B) (l : list A) : list B :=
 flatten (map f l).

(* Emergent distance between patterns *)
Definition emergent_distance (x y : Pattern) : Q :=
 list_min
   (flat_map (fun p1 =>
     map (fun p2 => euclidean_distance p1 p2)
       (geometric_points y))
   (geometric_points x))
   999. (* Large number as default *)

(*****************************************************************************)
(* GEOMETRIC STABILITY AND PRECISION                                         *)
(*****************************************************************************)

(* Granularity threshold - minimum distinguishable distance *)
Parameter delta_geometric : ObsState -> Q.

(* Granularity depends on capacity *)
```

```
Axiom granularity_capacity : forall a : ObsState,
  delta_geometric a = 1 / (capacity a).

(* Points closer than granularity threshold are indistinguishable *)
Definition distinguishable (a : ObsState) (p1 p2 : Q * Q) : Prop :=
  euclidean_distance p1 p2 > delta_geometric a.

(* Stability implies consistent geometric properties *)
Definition geometric_stable (x : Pattern) : Prop :=
  Stable x /\
  forall a1 e1 a2 e2,
    InRegion (a1, e1) /\ InRegion (a2, e2) ->
    forall p1 p2,
      In p1 (geometric_points x) /\ In p2 (geometric_points x) ->
      distinguishable a1 p1 p2 <-> distinguishable a2 p1 p2.

(* Higher capacity leads to finer geometric resolution *)
Axiom capacity_resolution : forall a1 a2 : ObsState,
  Qlt (capacity a1) (capacity a2) ->
  Qlt (delta_geometric a2) (delta_geometric a1).

(*****************************************************************************)
(* EMERGENT LINES AND SHAPES                                                 *)
(*****************************************************************************)

(* A line is a sequence of points with minimal transitions *)
Definition is_line (pts : list (Q * Q)) (a : ObsState) : Prop :=
  nat_to_Q (length pts) >= 2 /\
  forall i, (i < length pts - 1)%nat ->
    let p1 := nth i pts (0, 0) in
    let p2 := nth (i+1)%nat pts (0, 0) in
    distinguishable a p1 p2 /\
    forall p,
      ~(In p pts) ->
      ~(euclidean_distance p1 p < euclidean_distance p1 p2 /\
        euclidean_distance p p2 < euclidean_distance p1 p2).

(* A shape (polygon) is a closed loop of lines *)
Definition is_polygon (pts : list (Q * Q)) (a : ObsState) : Prop :=
  nat_to_Q (length pts) >= 3 /\
  is_line (pts ++ (firstn 1 pts)) a.

(* Emergent geometric properties depend on observer capacity *)
Axiom geometric_precision : forall a : ObsState, forall pts : list (Q * Q),
```

```
 is_line pts a ->
 forall a', Qlt (capacity a) (capacity a') ->
  exists pts', is_line pts' a' /\
    (* New line approximates the original but with higher precision *)
    (length pts' >= length pts)%nat.

(*****************************************************************************)
(* INTEGRATION WITH PROBABILISTIC FRAMEWORK                                  *)
(*****************************************************************************)

(* Probability of a point being part of a geometric structure *)
Parameter point_probability : ObsState -> EnvState -> Q * Q -> Q.

(* Point probability depends on observer capacity *)
Axiom point_probability_capacity : forall a e p,
 point_probability a e p <= capacity a.

(* Line probability derived from point probabilities *)
Definition line_probability (a : ObsState) (e : EnvState) (pts : list (Q * Q)) : Q :=
 fold_right Qmult 1 (map (point_probability a e) pts).

(* Geometric structures emerge from stable patterns *)
Axiom geometry_from_stability : forall x : Pattern,
 Stable x ->
 exists pts : list (Q * Q),
  forall a e, InRegion (a, e) ->
   fold_right Qplus 0 (map (point_probability a e) pts) > 1/2 * (capacity a).

(* This axiom establishes that stable patterns give rise to
  recognizable geometric structures with high probability *)

End Phase4_Emergent_Geometry.
```