A Mechanized Formalization of the ϱ-Calculus and the Spice Calculus in Lean 4

Zar Goertzel and Claude Opus
Mettapedia Project
February 2026

Abstract

We present a mechanized formalization in Lean 4 of the ϱ-calculus (reflective higher-order calculus) and its temporal extension, the *spice calculus*, as described by Meredith and collaborators. All theorems are **fully verified** with zero sorries and zero axioms. Key results include: structural congruence and reduction semantics for the ϱ-calculus; semantic characterization of values as normal forms, with the equivalence $\mathsf{Value}(p) \iff \mathsf{presentMoment}(p) = \emptyset$; the correctness of the spice calculus reduction—proving that the standard ϱ-calculus is recovered at horizon $n = 0$; and the formalization of the agent's *present moment* including surface channels, internal channels, and the connection between irreducibility and empty interaction sets.

1 Introduction

The ϱ-calculus, introduced by Meredith and Radestock [1], is a reflective higher-order process calculus that extends the π-calculus with a quote-dereference mechanism: any *process* can be quoted to produce a *name*, and any name can be dereferenced back into a process. This reflexive structure—names are codes for processes—gives the calculus a remarkable economy of concepts while retaining the full expressiveness of higher-order process calculi.

Meredith's recent work on the *spice calculus* [2] extends this framework with n-step lookahead, giving agents temporal structure: a past (trace of reductions), a present (1-step reachability), and a future (n-step lookahead). When $n = 0$, the spice calculus recovers the standard ϱ-calculus exactly.

This paper reports on a mechanized formalization of these ideas in Lean 4 with Mathlib, resulting in a library that is:

- **Fully verified**: all theorems machine-checked with no axioms or gaps.
- **Constructive where possible**: the reduction relation `Reduces` is `Type`-valued, enabling extraction of reduction witnesses.
- **Built on MeTTaIL syntax**: the process term language is formalized as MeTTaIL `Patterns`, matching the Rust implementation of RChain/MeTTa.

2 The ϱ-Calculus

2.1 Syntax

The ϱ-calculus has two mutually defined syntactic categories:

Definition 2.1 (ϱ-Calculus Syntax)**.**

$$\begin{array}{rcll} P, Q & ::= & \mathbf{0} & \text{(nil process)} \\ & | & n\,!\,(Q) & \text{(output)} \\ & | & \mathsf{for}(x \leftarrow n)\{P\} & \text{(input)} \\ & | & \{P \mid Q\} & \text{(parallel)} \\ & | & *\,(n) & \text{(dereference)} \\ n, m & ::= & @(P) & \text{(quote)} \end{array}$$

In our formalization, these are represented as MeTTaIL `Pattern` constructors: `.apply "PZero" []`, `.apply "POutput" [n, q]`, `.apply "PInput" [n, .lambda x p]`, `.collection .hashBag ps none`, `.apply "PDrop" [n]`, and `.apply "NQuote" [p]`, respectively.

2.2 Structural Congruence

Definition 2.2 (Structural Congruence)**.** The structural congruence $\equiv$ is the least congruence containing $\equiv_\alpha$ and satisfying:

$$P \mid \mathbf{0} \equiv P \equiv \mathbf{0} \mid P$$
$$P \mid Q \equiv Q \mid P$$
$$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$$
$$@(*(n)) = n \quad \text{(quote-drop)}$$

Our Lean formalization includes:

- `AlphaEquiv`: α-equivalence with bound variable renaming, congruence under all constructors, and explicit symmetry/transitivity.
- `StructuralCongruence`: 20 constructors covering α-equivalence, parallel composition laws, list permutation, flattening, congruence under all term formers, and quote-drop.
- `NameEquiv`: Name equivalence respecting structural congruence, formalizing the STRUCT-EQUIV rule from [1].

A key formalization insight: quote *respects* structural congruence (including α-equivalence). If $P \equiv Q$ then $@(P) \equiv_N @(Q)$. This is the STRUCT-EQUIV rule on page 7 of [1].

2.3 Reduction Semantics

Definition 2.3 (Reduction)**.** The one-step reduction $\rightsquigarrow$ is defined by:

$$\frac{}{\{n\,!\,(Q) \mid \mathsf{for}(x \leftarrow n)\{P\}, \ldots \mathrm{rest}\} \rightsquigarrow \{P[@(Q)/x], \ldots \mathrm{rest} \mid\}}\ \textsc{Comm} \qquad \frac{}{*(@(P)) \rightsquigarrow P}\ \textsc{Drop}$$

$$\frac{P \equiv P' \qquad P' \rightsquigarrow Q' \qquad Q' \equiv Q}{P \rightsquigarrow Q}\ \textsc{Equiv} \qquad \frac{P \rightsquigarrow Q}{\{P \mid \mathrm{rest}\} \rightsquigarrow \{Q \mid \mathrm{rest}\}}\ \textsc{Par}$$

A crucial design decision: `Reduces` is `Type`-valued rather than `Prop`-valued. This means reduction derivations are computational objects encoding *how* a reduction happens, not just *that* it happens. Wrapping in `Nonempty` recovers the propositional version where needed.

The Lean formalization includes 7 constructors: `comm`, `drop`, `equiv`, `par`, `par_any`, `par_set`, and `par_set_any`. Parallel composition is encoded as ordered lists (`.collection .hashBag`) with permutation congruence in structural congruence; `par_set` and `par_set_any` are the analogous rules for set collections used in the spice calculus.

3 Values, Normal Forms, and Progress

A *value* (or *normal form*) is a pattern from which no reduction step is possible—the semantic notion of irreducibility:

Definition 3.1 (Value / Normal Form)**.**

$$\begin{aligned} \mathsf{CanStep}(p) &\iff \exists q.\, p \rightsquigarrow q \\ \mathsf{NormalForm}(p) &\iff \neg\, \mathsf{CanStep}(p) \\ \mathsf{Value} &= \mathsf{NormalForm} \end{aligned}$$

This is the correct semantic definition: it automatically respects all reduction rules (COMM, DROP, PAR, EQUIV) without needing to track syntax. Following the standard ϱ-calculus semantics [1], reduction does *not* propagate under input or output guards—guarded processes are blocking until synchronization. The formalization also provides a decidable syntactic approximation `isInertSyntax` that accepts a superset of normal forms—it does not check for COMM-redexes in parallel bags, so some reducible patterns pass the check. The useful direction is the contrapositive: failing the check implies the pattern genuinely reduces (proven as `non_inert_proc_reduces`).

Theorem 3.2 (Progress)**.** *Every pattern is either reducible or a value:* $\mathsf{CanStep}(p) \vee \mathsf{Value}(p)$.

This is an instance of excluded middle (proven as `step_or_normalForm`). The real content lies in canonical-forms lemmas that characterize what normal forms look like:

Lemma 3.3 (Canonical Forms)**.**
1. $\mathsf{NormalForm}(*(@(P)))$ *is false for any* P *(`normalForm_no_drop`).*
2. *In empty context, all names are quotes (`empty_context_name_is_quote`).*
3. *If* p *is syntactically non-inert and well-typed, then* p *reduces (`non_inert_proc_reduces`).*

4 The Spice Calculus

4.1 Temporal Structure

The spice calculus [2] extends the ϱ-calculus with n-step lookahead, giving agents temporal structure.

Definition 4.1 (Future States and Present Moment)**.**

$$\begin{aligned} \mathsf{futureStates}(p, n) &= \{q \mid p \rightsquigarrow^n q\} \\ \mathsf{presentMoment}(p) &= \mathsf{futureStates}(p, 1) \\ \mathsf{reachableStates}(p, n) &= \{q \mid \exists k \le n.\, p \rightsquigarrow^k q\} \\ \mathsf{spice}(p, n) &= \mathsf{reachableStates}(p, n) \end{aligned}$$

Theorem 4.2 (Star = Union of Futures)**.** $\{q \mid p \rightsquigarrow^* q\} = \bigcup_n \mathsf{futureStates}(p, n)$

This fundamental theorem connects the reflexive-transitive closure to n-step reachability: every reachable state is reachable in exactly n steps for some finite n.

4.2 Spice COMM Rule

Definition 4.3 (Spice COMM)**.** Given horizon n and a finiteness proof for $\mathsf{spice}(Q, n)$:

$$\frac{}{\{x\,!\,(Q) \mid \mathsf{for}(y \leftarrow x)\{P\}, \ldots \mathrm{rest}\} \rightsquigarrow_n \{P[@(\mathsf{spice}(Q, n))/y], \ldots \mathrm{rest} \mid\}}\ \textsc{Spice-Comm}$$

The key design: `spiceCommSubst` is parameterized by a finiteness proof `h_fin : (spiceEval q n).Finite`. For $n = 0$, this is provided by `spiceEval_zero_finite` (the singleton $\{q\}$ is trivially finite). For $n > 0$, finiteness requires finite branching modulo structural congruence—a standard assumption in process calculus, but not provable without quotienting by SC.

Theorem 4.4 (Recovery at Horizon 0)**.** *When $n = 0$, the spice COMM rule exactly recovers the standard ϱ-calculus COMM rule.*

Proof. Three key lemmas compose:

1. $\mathsf{spice}(q, 0) = \{q\}$ (`spice_zero_is_current`)
2. $\mathsf{futureSetAsPattern}(\{q\}, h) = q$ (`futureSetAsPattern_singleton`, singleton unwrapping)
3. $\mathsf{spiceCommSubst}(p, x, q, 0, h) = \mathsf{commSubst}(p, x, q)$ (`spiceCommSubst_zero`)

Together, `reactive_is_standard` proves that every `SpiceCommReduction 0` derivation produces a standard `Reduces` derivation, by induction on the spice reduction structure. □

4.3 The Present Moment

Following [2], Section 4.4.1, an agent's *present moment* comprises all interactions it can have immediately:

Definition 4.5 (Present Moment)**.**

$$\begin{aligned}\mathsf{surf}(a, e) &= \{x \in \mathsf{FN}(a) \cap \mathsf{FN}(e) \mid a \mid e \downarrow_x\} \\ \mathsf{int}(a, e) &= \{x \in \mathsf{N}(a) \setminus \mathsf{FN}(e) \mid a \downarrow_x\} \\ \mathsf{PM}(a, e) &= \mathsf{PM}_{\mathsf{ext}}(a, e) \cup \mathsf{PM}_{\mathsf{int}}(a, e)\end{aligned}$$

where $\mathsf{PM}_{\mathsf{ext}}$ collects external interactions (agent with environment) and $\mathsf{PM}_{\mathsf{int}}$ collects internal self-interactions.

Notation. We use $\mathsf{presentMoment}(p)$ (Definition 4.1) for the *successor set*—all 1-step reducts of a state p—and $\mathsf{PM}(a, e)$ (Definition 4.5) for the *interaction-context* present moment of an agent a in an environment e.

Our formalization proves:

- `surf_comm`: Surface channels are symmetric in agent/environment.
- `presentMoment_nonempty_iff`: The present moment is nonempty iff there is some interaction channel.
- `presentMoment_subset_futureStates`: The present moment is a subset of 1-step future states.

Theorem 4.6 (Value $\iff$ Empty Present Moment)**.** $\mathsf{Value}(p) \iff \neg(\mathsf{presentMoment}(p) \neq \emptyset)$

This connects the semantic notion of value (irreducibility) directly to the paper's temporal structure: an agent whose present moment is empty has no available interactions—it is stuck, i.e., a value. The proof follows immediately from `presentMoment_nonempty_iff_reduces`.

4.4 The Past: Temporal Duality

The past is the temporal dual of the future (Section 4.4.3 of [2]). Where $\mathsf{futureStates}$ asks "where can I go?", past states ask "where could I have come from?":

Definition 4.7 (Past States)**.**

$$\begin{aligned}\mathsf{pastStates}(p, n) &= \{q \mid q \rightsquigarrow^n p\} \\ \mathsf{predecessors}(p) &= \{q \mid q \rightsquigarrow^* p\}\end{aligned}$$

The central theorem is past–future duality:

Theorem 4.8 (Past–Future Duality)**.** $q \in \mathsf{pastStates}(p, n) \iff p \in \mathsf{futureStates}(q, n)$

As an immediate corollary, the present moment bridges past and future: q is an immediate predecessor of p iff p is in the present moment of q (`immediatePast_iff_presentMoment`). Together with `predecessors_eq_union_past` (the dual of `star_eq_union_future`), this shows the temporal structure is fully symmetric.

A fundamental asymmetry remains: values have no future (`value_no_future`: $\mathsf{Value}(p) \Rightarrow \mathsf{presentMoment}(p) = \emptyset$) but always have a past—every pattern p has $*(@(p))$ as an immediate predecessor via DROP (`drop_in_immediatePast`).

4.5 Race Conditions

Section 4.3.1 of [2] discusses *races*: situations where multiple inputs compete for the same output on a channel. We formalize this:

Definition 4.9 (Race Condition)**.** $\mathsf{hasRace}(P, x)$ holds when P is a parallel composition containing an output $x\,!\,(Q)$ and at least two distinct inputs $\mathsf{for}(y_1 \leftarrow x)\{P_1\}$, $\mathsf{for}(y_2 \leftarrow x)\{P_2\}$.

Our formalization proves:

- `hasRace_implies_canInteract`: a race on x implies interaction.
- `hasRace_implies_canStep`: a racing process can step (not a value).
- `value_no_race`: values have no race conditions.
- `race_nondeterminism`: under `List.Nodup` (processes are resources), a race on x implies ≥ 2 distinct (non-equal) reducts, witnessing genuine non-determinism. The proof constructs two COMM reductions pairing the output with each competing input, then shows the results differ because each retains the input the other consumed.

4.6 Episodic Memory

Section 4.4.4 of [2] decomposes an agent into *recipes* (input guards) and *facts* (output guards). We formalize this via `AgentMemory` and prove soundness and completeness:

- **Soundness**: `extractMemory_recipes_sound` and `extractMemory_facts_sound`—every extracted recipe/fact corresponds to a real `PInput`/`POutput` in the original bag.
- **Completeness**: `extractMemory_recipes_complete` and `extractMemory_facts_complete`—every top-level `PInput`/`POutput` in the bag appears in the extracted memory.
- **Self-interaction**: `memory_self_interaction`—if an agent has both a recipe and a fact on the same channel, COMM can fire.

4.7 Logged COMM and Reversibility

Section 4.4.3 of [2] states that "the past only comes into being if there is some record or trace" and connects this to reversibility. We formalize a *logged COMM* that records exactly what was consumed:

Definition 4.10 (COMM Record)**.** A `CommRecord` stores the channel, consumed input guard (variable and body), consumed output payload, and remaining parallel siblings.

From a COMM record, we prove:

- **Forward**: `CommRecord.forward`—the record witnesses a genuine reduction `preState` $\rightsquigarrow$ `postState`.
- **Backward**: `CommRecord.reconstruct`—from the record, the pre-state is exactly recoverable.

- **Linearity**: `CommRecord.recipe_consumed` and `CommRecord.fact_consumed`—the consumed recipe and fact are absent from the post-state (under non-duplication hypotheses), formalizing the "linear and destructive" interaction from Section 4.4.4.

This demonstrates the core reversibility idea: each COMM step is undoable when the record is available, without requiring the full continuation-saturated form transformation.

5 Formalization Architecture

The core formalization spans the following Lean files:

File	Content
`MeTTaIL/Syntax.lean`	Pattern AST, language definitions
`MeTTaIL/Substitution.lean`	Substitution, free variables
`RhoCalculus/StructuralCongruence.lean`	$\equiv_\alpha$, $\equiv$, $\equiv_N$
`RhoCalculus/Types.lean`	Typing judgments, barbs, ProcEquiv
`RhoCalculus/Reduction.lean`	$\rightsquigarrow$, Value, normal forms
`RhoCalculus/Soundness.lean`	Substitutability, syntactic progress
`RhoCalculus/Context.lean`	Eval contexts, labeled transitions
`RhoCalculus/MultiStep.lean`	$\rightsquigarrow^*$, $\rightsquigarrow^n$
`RhoCalculus/SpiceRule.lean`	Future states, spice evaluation
`RhoCalculus/CommRule.lean`	Spice COMM, $n = 0$ recovery
`RhoCalculus/PresentMoment.lean`	Agent's present moment, surf, int

The formalization builds on Lean 4.27.0 with Mathlib v4.27.0. All files compile with zero warnings, zero sorries, and zero axioms.

5.1 Design Decisions

Type-valued reduction. The `Reduces` inductive is `Type`-valued rather than `Prop`-valued. This captures reduction derivations as computational witnesses, enabling proof complexity analysis and extraction. The cost is that some lemmas require `Nonempty` wrapping, but this is a minor syntactic overhead.

MeTTaIL as base syntax. Rather than defining a bespoke process calculus AST, we use the MeTTaIL `Pattern` type—the same AST used in the Rust implementation of RChain. This grounds the formalization in the actual implementation language and enables future verification of the Rust code against the Lean specification.

Finiteness parameterization. The spice calculus substitution `spiceCommSubst` takes an explicit finiteness proof as a parameter rather than relying on a global axiom. This is because finiteness of future states is *false* in general (alpha-equivalence produces infinitely many syntactic variants); it becomes true only after quotienting by structural congruence. The parameterized design keeps the formalization sound while enabling use wherever finiteness can be established.

6 Related Work

Formal mechanization of process calculi has a substantial history. Gay and Hole [3] formalized the π-calculus in Isabelle/HOL. Hirschkoff [4] verified the π-calculus metatheory in Coq. More recently, Bengtson et al. [5] developed the Psi-calculi framework in Isabelle, parametric in the data and channel structure.

Our work differs in three respects:

1. We formalize the ϱ-calculus specifically, including the quote/dereference mechanism that is absent from π-calculus formalizations.
2. We formalize the spice calculus, a temporal extension not present in any prior mechanized work, including the $n = 0$ recovery theorem.
3. We formalize the agent's *present moment*—the set of immediately available interactions—and prove its connection to semantic values.

7 Conclusion

We have presented a fully mechanized formalization of the ϱ-calculus and spice calculus in Lean 4, establishing:

1. **Structural congruence and reduction**: 20 structural congruence constructors covering α-equivalence, parallel composition laws, and congruence; reduction via COMM, DROP, PAR, and EQUIV (Section 2).
2. **Semantic values**: Value = NormalForm with canonical forms and the equivalence to empty present moment (Theorems 3.2, 4.6).
3. The **spice calculus recovery** theorem: horizon $n = 0$ exactly recovers the standard ϱ-calculus (Theorem 4.4).
4. The **present moment** formalization: surface channels, internal channels, and the connection between irreducibility and empty interaction sets (Section 4.3).
5. **Past–future duality**: the temporal structure is symmetric, with the present moment bridging the two directions (Section 4.4).
6. **Race conditions and episodic memory**: races imply non-stuck processes and, under non-duplication, at least two distinct reducts; agent memory decomposition is sound and complete (Sections 4.5, 4.6).

All results are fully machine-checked with zero sorries and zero axioms.

Future work. Several directions remain:

- Quotienting by structural congruence to enable finiteness proofs for $n > 0$ spice evaluation.
- Formalizing the full continuation-saturated form (CSF) as a source-to-source transformation, beyond the logged-step POC.
- Connecting to typed communication via MeTTaIL/GSLT: the present formalization uses MeTTaIL `Patterns` as base syntax and includes mechanized substitution infrastructure, but does not yet formalize channels carrying terms in a graph-structured λ-theory as described in Section 5 of [2]. A companion Lean library already provides the categorical scaffolding needed for this step: a `SubobjectFibration` with frame-valued fibers and a `ChangeOfBase` construction that equips each morphism f with the adjoint triple $\exists_f \dashv f^* \dashv \forall_f$. Instantiating this infrastructure over the ϱ-calculus reduction graph would yield the modal operators ($\Diamond$, $\Box$) and their Galois connection from the OSLF algorithm of [1]. This bridge is not yet established in the present development.

Acknowledgments. This work was inspired by and formalized from the mathematical ideas of L. Gregory Meredith.

References

[1] L.G. Meredith and M. Radestock, "A Reflective Higher-Order Calculus," *Electronic Notes in Theoretical Computer Science*, vol. 141, no. 5, pp. 49–67, 2005.

[2] L.G. Meredith, "How the Agents Got Their Present Moment," Preprint, 2026.

[3] S. Gay and M. Hole, "Subtyping for Session Types in the Pi Calculus," *Acta Informatica*, vol. 42, pp. 191–225, 2005.

[4] D. Hirschkoff, "A Full Formalisation of Pi-Calculus Theory in the Calculus of Constructions," *Proceedings of TPHOLs*, Springer LNCS, 1997.

[5] J. Bengtson, M. Johansson, J. Parrow, and B. Victor, "Psi-Calculi: A Framework for Mobile Processes with Nominal Data and Logic," *Proceedings of LICS*, IEEE, 2009.