A Provenance-Aware Core Calculus for Triton-Distributed Synchronization

Baber Rehman ✉
Huawei Technologies, Hong Kong

Abstract

Triton-Distributed exposes signal-based synchronization operations intended to order communication and computation in distributed accelerator kernels. Reasoning about the intended wait–token–load discipline is difficult because the Python API, compiler IR, and backend memory model provide different parts of the contract. We present a mechanized core calculus that isolates one such discipline and makes its obligations explicit. The Lean 4 model includes typed values and environments, notification state, successful waits, token consumption, and guarded loads. Its transition semantics associates pending and consumed capabilities with signal, rank, pointer, scope, and ordering metadata. We prove preservation of capability origins across traces and a universal provenance theorem: from an initial state with no pending or consumed capabilities, every permitted load has an earlier matching wait and consume in the abstract action trace. We also mechanize a three-step consumer wait–consume–load trace from initialized state and negative examples that exclude loads without the required abstract capabilities. A source-pattern validator checks three representative upstream tutorial shapes against a pinned Triton-distributed checkout. These results establish properties of the encoded calculus and checked source patterns only. They do not establish a source-to-core simulation, compiler correctness, backend memory-model refinement, or hardware safety, and they identify the additional translation, compiler, and target evidence needed to support stronger claims.

2012 ACM Subject Classification Software and its engineering → Formal software verification; Software and its engineering → Concurrency

Keywords and phrases Lean, distributed accelerators, synchronization, provenance, operational semantics

Digital Object Identifier 10.4230/LIPIcs...

Supplementary Material `https://github.com/baberrehman/triton-distributed-formalization`

1 Introduction

Distributed accelerators make communication part of the computation: while one rank produces data, another may need to wait for a signal, establish a dependency on the data, and only then load it. Getting this order wrong can leave a kernel with values that are stale or otherwise not justified by the intended communication protocol. Yet the relevant contract is distributed across layers. An API can describe the shape of `notify`, `wait`, and `consume_token`; an intermediate representation can encode dependencies; compiler transformations can preserve or alter those dependencies; and a backend must map the operations to target-specific ordering and visibility. A source-level sequence that looks right is not, by itself, evidence that all these layers agree.

This creates a verification gap with a deceptively simple symptom: an ordinary load can be well typed and still lack a valid synchronization history. Scalar and pointer types tell us what a value represents, but not whether the consumer has waited for the matching signal, whether the resulting token was consumed for this data value, or whether the load is covered by that consumption. Conversely, proving that a particular example follows the expected sequence does not establish that every load admitted by a semantics has the required history. A useful first step is therefore to make the protocol obligation precise at the level where it can be stated and checked: the transition trace.


© Baber Rehman;
licensed under Creative Commons License CC-BY 4.0


Three schematic kernel fragments illustrate missing synchronization evidence. Here `tl` denotes Triton and `dl` denotes Triton-Distributed. These are protocol-risk examples, not experimentally confirmed runtime failures: compilation and observable behavior depend on the backend and optimization passes. Rejection refers to their corresponding abstract action patterns from empty pending and consumed capability lists; no verified Python-to-core translation is claimed.

Fabricated token without a wait.

```
fake_token = tl.full((), 1, tl.int32)
ordered_ptr = dl.consume_token(data_ptr, fake_token)
x = tl.load(ordered_ptr)
```

An unrelated integer establishes no producer completion. The inspected Python `consume_token` wrapper checks an integer element type without establishing wait-result provenance. In the calculus, `Step.consume` requires a matching pending capability created by a prior wait; an environment token alone is insufficient. The mechanized fabricated-token example demonstrates this rejection.

Wait without token consumption.

```
token = dl.wait(signal_ptr, 1)
# Missing consume_token dependency.
x = tl.load(data_ptr)
```

The ordinary load interface does not require evidence that its pointer is covered by token consumption. The fragment therefore lacks the explicit dependency prescribed by the TD programming contract, although this alone does not demonstrate incorrect execution on a particular backend. In the calculus, wait creates pending evidence, while load requires consumed evidence; a wait witness alone cannot authorize the load. The example `load_without_consume_is_not_derivable` exhibits that distinction.

Consumption for one location, followed by another load.

```
# signal_a protects data_a only.
token_a = dl.wait(signal_a, 1)
ordered_a = dl.consume_token(data_a, token_a)
x = tl.load(ordered_a)
# data_b has an independent producer.
y = tl.load(data_b)
```

Waiting for A provides no readiness evidence for independent data B, and ordinary `tl.load` does not check such a protocol association. In the calculus, a consumed record for A cannot authorize a load from an unrelated pointer variable or rank/address: `Step.load` requires matching fields. For reachable states, the universal provenance theorem reconstructs the required wait and consume history for each permitted load.

These examples motivate a history discipline, rather than establishing implementation bugs or producer-to-consumer data correctness. Its rejection results concern abstract capability histories; they do not establish hardware visibility or correctness of the values read.

We study that obligation in a small Lean 4 core calculus motivated by Triton-Distributed's documented producer–consumer idiom. The calculus separates expression typing and evaluation from machine transitions, and represents synchronization evidence explicitly. A successful wait creates a pending capability carrying signal, rank, expected value, scope, and ordering metadata; consuming a token for a data value turns that evidence into a consumed record for the value variable, token variable, signal address, and rank. A load is admitted

when that record matches its pointer variable, rank, and address. Crucially, notification visibility is an explicit premise of successful waiting. The model does not silently derive a hardware fact that it has not formalized.

The central result is a universal provenance theorem, not merely a checked execution: for every trace admitted by the transition relation from an initial machine with no pending or consumed capabilities, any permitted load has an earlier wait followed by a consume using the same token variable. The consumed record must match the loaded pointer variable, rank, and address; the consume must match a pending record's token variable, rank, address, scope, and ordering. However, the consumed record omits the expected value, scope, and ordering, so these are not matched at the final load. The theorem rules out a class of internally inconsistent abstract executions and makes the required evidence inspectable in Lean. This gives a precise local invariant that can guide stronger verification: a future source-to-core translation could show that source dependencies create the modeled capabilities, and a backend refinement could show that generated operations preserve their ordering and visibility.

That boundary is essential to the contribution. We do not prove that arbitrary Triton-Distributed programs translate into this calculus, that compiler passes preserve its protocol, that any hardware backend satisfies the visibility premise, or that a kernel is safe end to end. We instead formalize one synchronization discipline, prove what follows from it, audit selected tutorial-shaped source patterns, and identify the missing links needed to connect the abstract result to executable kernels.

Contributions.

- We define a typed Lean core with notification, wait, consume, and load transitions, and state the notification-visibility condition as an explicit semantic premise.
- We prove that pending and consumed capabilities retain trace origins, and derive a universal wait–consume provenance decomposition for permitted loads from empty capability state.
- We provide executable positive and negative protocol examples, together with a standard-library Python AST validator for selected upstream tutorial call shapes.
- We document the source correspondence and trusted boundary, including the missing translation, dependency-preservation, and backend memory-model evidence.

These contributions should be read as a source-audited formal prototype, not as verification of Triton-Distributed or a claim of priority; the related-work comparison is focused rather than systematic.

Roadmap and notation. Section 2 records the source contract and abstraction boundary. Section 3 defines the Lean vocabulary, expression evaluator, and transition rules; subsections follow the order of the declarations in `Core.lean` and `Transition.lean`. Section 4 states the mechanized invariants and gives proof decompositions. Section 5 presents concrete traces and source-pattern validation. Section 6 compares the calculus with the implementation, while Section 7 lists assumptions and limitations. We write $x \in L$ for list membership, $L_1 ++ L_2$ for list concatenation, and $m \xrightarrow{a} m'$ for an enabled transition labelled by action a. Lean identifiers are set in typewriter font when the exact declaration is relevant.

2 Background and Source Contract

Distributed kernels execute across ranks that own distinct portions of a computation but may communicate through remotely addressable, symmetric allocations. A rank can form

an address for a peer's corresponding allocation, issue communication or computation, and coordinate progress with signals. This setting makes synchronization more than a control-flow concern: the consumer must establish that the producer's relevant data is ready and that the dependency remains attached to the later data access. The paper focuses on one documented notify–wait–consume–load pattern rather than attempting to model this broader execution environment.

The source interface. The audited Triton-Distributed checkout provides rank and world-size queries, symmetric addressing, signal notification, waiting, token consumption, and ordinary data operations. We use the Python API, distributed operation definitions, language and semantics documentation, and the notify–wait tutorial at revision `1e6d688` (2026-08-24). The revision is also recorded in the data-availability statement. These sources establish the operation shapes and motivate the selected pattern; they are not treated as a formal semantics for all source programs.

Producer–consumer pattern. In the tutorial-shaped idiom, a producer writes data and notifies a signal associated with a consumer. The consumer waits for the expected signal state, receives a token, associates that token with a data value using `consume_token(value, token)`, and subsequently loads through the relevant pointer. Informally, the signal communicates progress, while token consumption makes the synchronization dependency relevant to the data use. The exact details of signal setup, allocation, rank selection, and reuse belong to the surrounding program and are outside the small protocol modeled here.

What the token means here. At the source interface, the wait result is represented as an integer token, and `consume_token` is described as introducing a dependency associated with its value operand. An integer type alone cannot express where a token came from or which data access it justifies. The core therefore distinguishes the token's static scalar type from its abstract metadata and capability history: a successful wait creates pending evidence, a matching consume transforms that evidence into an ordering witness for a value, and a guarded load consults that witness. This is a formal design choice for the calculus, not a claim that the source API's integer token intrinsically carries all of this metadata.

From source operations to abstract actions. The source audit records a deliberately partial correspondence. Rank and symmetric-addressing operations motivate typed rank and pointer values; notify and wait motivate signal and wait actions; consume motivates a transition that requires a matching pending token; and load motivates a transition guarded by an ordering witness. The model abstracts away pointer lowering, address calculation, general tensor values, program control flow, concurrency scheduling, and the complete semantics of the source language. The accompanying AST validator checks selected tutorial call shapes; it is a source-pattern check, not a translation into Lean and not a proof that the source executes according to the core semantics.

Protocol and readiness mapping. The pinned tutorial and Ascend test validly use separate signal and data buffers, a pattern not directly represented by the model's address-equality premise at consume. A source-to-core abstraction mapping must explicitly relate signal, rank, and epoch to the protected data region, including pointer offsets. Wait readiness also differs: the model requires $signal \geq expected$ and the positive-signal predicate `NotificationVisible`, whereas the pinned NVIDIA and AMD lowerings poll for equality. The model permits positive

overshoots but excludes a zero signal; the tutorial initializes signals to zero and validly waits for `queue_repeat * 2`, including zero on the first producer iteration. These differences establish no general acceptance or refinement ordering between model and implementation; a simulation needs an explicit protocol relation and backend-specific readiness semantics.

Ordering and visibility boundary. The intended reasoning depends on two distinct obligations: synchronization must establish the relevant data visibility, and the dependency introduced by token consumption must remain effective until the data load. The calculus uses `NotificationVisible` as an abstract wait premise, but its current definition checks only a positive signal counter, an in-range signal owner, and that scope is `gpu` or `system`. It neither inspects data memory nor relates producer writes to consumer reads; it has no ordering argument. A matching consumed record guards the load. Scope and ordering metadata are recorded and matched at consume, but these checks establish neither memory visibility nor compiler preservation of the dependency. Establishing those facts requires target- and compiler-specific evidence, as detailed in the limitations and memory-model discussion.

The resulting contract is intentionally conditional: if the abstract transition system admits a load, then its state must contain the corresponding protocol evidence. The theorem proved later reconstructs the wait and consume origins of that evidence from the trace. Connecting this conditional result to Triton-Distributed executions would additionally require a source/IR translation, a simulation argument, compiler dependency-preservation evidence, and a backend refinement for notification visibility. The current artifact supplies none of those end-to-end links; it makes their exact role explicit.

3 Core Calculus

This section gives the model in enough detail to reconstruct its definitions and to understand exactly what the proved theorems quantify over. The development is split into two Lean modules. `Core.lean` defines expressions, values, states, typing, and big-step evaluation. `Transition.lean` extends this vocabulary with explicit state transitions, capabilities, traces, and provenance theorems. `Examples.lean` constructs concrete traces and counterexamples over the transition semantics.

The model has three design goals. First, symmetric addressing and loads should preserve pointer element types. Second, successful synchronization should leave an explicit capability that can be matched against a later data access. Third, that capability should have a trace origin, so an unrelated integer token cannot justify a load. Keeping the value and state domains small keeps the artifact focused on the protocol rather than on parser, substitution, or thread-scheduling machinery.

The model separates expression evaluation from protocol transitions. The expression relation describes the result of one operation in a fixed state; the transition relation evolves signals and capability state across actions. This separation permits ordinary type preservation to be stated independently from token provenance: a token can have the scalar token type without yet having a valid wait origin.

3.1 Machine state

A machine pairs a state with a typed environment, a list of ordering witnesses, a list of pending tokens, and a list of consumed orderings. The state itself is introduced below; the two capability lists are populated only by the transitions of Section 3.7, where their fields are also given.

3.1.1 Primitive domains

Besides the data types of the next subsection, the model fixes two finite domains. Scopes are `gpu` and `system`; orderings are `relaxed`, `acquire`, `release`, and `acqRel`. Both are Lean inductive types, as is `Result`, the outcome of evaluation. Ranks, variables, and addresses are all aliases for `Nat`.

The explicit scope and ordering fields are retained even though the current provenance theorem does not model a weak-memory execution. A wait records these fields in its pending capability, and consume requires them to match. They are parameters of the future backend-refinement boundary: a target proof must explain what each pair means on the implementation platform.

3.1.2 Types, expressions and values

Types. Types are the `Ty` structure, a data type paired with a size:

$$\tau ::= \langle d, n \rangle, \qquad d ::= \mathsf{i32} \mid \mathsf{i64} \mid \mathsf{f16} \mid \mathsf{f32}.$$

Two sizes carry a special meaning throughout: $\langle d, 1 \rangle$ is a scalar of dtype d and $\langle d, 0 \rangle$ is the unit type, which is why a notification or a store returns $\mathsf{i32}_0$ rather than a distinguished void type. We write $\mathsf{i32}_n$ for $\langle \mathsf{i32}, n \rangle$ and likewise for $\mathsf{i64}_n$.

Expressions. The expression grammar is the following rendering of the `Expr` inductive type:

$$\begin{aligned} e \quad ::= \quad & x \mid \mathsf{rank}(a) \mid \mathsf{numRanks}(a) \mid \mathsf{symmAt}(e, r) \mid \mathsf{notify}(e, e, e, b, s) \mid \mathsf{wait}(e, e, e, s, o) \\ & \mid \mathsf{consumeToken}(e, e) \mid \mathsf{load}(e) \mid \mathsf{store}(e, e). \end{aligned}$$

Here a is an integer axis parameter, r is a target rank, b is the Boolean choice between set and add notification, and s, o are scope and ordering. The grammar retains the source operation boundaries. In particular, `consumeToken` is not reduced to an ordinary identity expression, even though it returns its value operand; its protocol effect is represented in the transition layer.

Values. Runtime values include tiles, scalar integers, pointers, tokens, and unit:

$$v ::= \mathsf{tile}(\tau, \vec{z}) \mid \mathsf{i32}(z) \mid \mathsf{i64}(z) \mid \mathsf{ptr}(\tau, r, a, \mathsf{sym}) \mid \mathsf{token}(a, n, s, o) \mid \mathsf{unit}.$$

The pointer stores an element type, owner rank, address, and a symmetric bit. The token stores signal address, observed value, scope, and ordering. Its static type is still scalar $\mathsf{i32}$; metadata is used by the abstract protocol.

Environments and machine state. The environment ρ maps variables to optional values. The state is $\sigma = (W, R, M, T, S, O)$, containing world size, current rank, memory, cell types, signals, and signal ownership. Memory maps a rank and address to an optional value; cell types map the same pair to an optional type. Signals map rank and address to natural counters. The separate memory and cell-type maps support a store well-typedness predicate without introducing a full typed heap semantics.

$$\frac{\Gamma.vars(x) = \mathsf{some}\ \tau}{\Gamma \vdash x : \tau}\ \textsf{t-var} \qquad \frac{}{\Gamma \vdash \mathsf{rank}(a) : \mathsf{i32}_1}\ \textsf{t-rank} \qquad \frac{}{\Gamma \vdash \mathsf{numRanks}(a) : \mathsf{i32}_1}\ \textsf{t-nrank}$$

$$\frac{\Gamma \vdash e : \tau \quad r < \Gamma.world}{\Gamma \vdash \mathsf{symmAt}(e, r) : \tau}\ \textsf{t-symm} \qquad \frac{\Gamma \vdash p : \mathsf{i64}_1 \quad \Gamma \vdash r : \mathsf{i32}_1 \quad \Gamma \vdash v : \mathsf{i64}_1}{\Gamma \vdash \mathsf{notify}(p, r, v, b, s) : \mathsf{i32}_0}\ \textsf{t-notify} \qquad \frac{\Gamma \vdash p : \mathsf{i64}_1 \quad \Gamma \vdash n : \mathsf{i32}_1 \quad \Gamma \vdash v : \mathsf{i64}_1}{\Gamma \vdash \mathsf{wait}(p, n, v, s, o) : \mathsf{i32}_1}\ \textsf{t-wait}$$

$$\frac{\Gamma \vdash v : \tau \quad \Gamma \vdash t : \mathsf{i32}_1}{\Gamma \vdash \mathsf{consumeToken}(v, t) : \tau}\ \textsf{t-consume} \qquad \frac{\Gamma \vdash p : \tau}{\Gamma \vdash \mathsf{load}(p) : \tau}\ \textsf{t-load} \qquad \frac{\Gamma \vdash p : \tau \quad \Gamma \vdash v : \tau}{\Gamma \vdash \mathsf{store}(p, v) : \mathsf{i32}_0}\ \textsf{t-store}$$

Figure 1 The complete static typing relation $\Gamma \vdash e : \tau$ for the core calculus. The rules are the constructors of the inductive predicate `HasType` in `Core.lean`. Written as $\mathsf{i32}_n$ and $\mathsf{i64}_n$, the types are those of the `Ty` structure, which records a dtype and a size only.

3.2 Static typing

The context Γ contains a world size and a variable-to-type map. The judgment $\Gamma \vdash e : \tau$, implemented by `HasType`, has one constructor for each expression form; its nine rules are collected in Figure 1 and discussed one at a time below.

Because `Ty` records only a dtype and a size, the judgment tracks no provenance: a token is well typed by its type alone whether or not it came from a successful wait. The runtime `Eval` rules supply the constructor obligation and the capability invariants supply the provenance fact, so the main result is a protocol refinement of an ordinary type system rather than a typing discipline that encodes lineage.

3.2.1 Typing rules

Each rule in Figure 1 is named after the `HasType` constructor it renders, with a `t-` prefix; types use the $\mathsf{i32}_n$ notation of Section 3.1.2, so that $\mathsf{i32}_1$ is a scalar and $\mathsf{i32}_0$ is unit.

Rule t-var is the only rule that consults the context: it succeeds when $\Gamma.vars(x)$ is defined at τ, and, since the judgment admits no subtyping or weakening step, a variable has exactly one type. Rules t-rank and t-nrank are axioms: both queries have scalar type $\mathsf{i32}_1$ for any axis argument a, which the evaluator ignores and which is present only to match the source signature. Rule t-symm reads the operand type and a target rank, preserves the element type τ, and requires the target to lie below the context world size; because the conclusion has the same type as the premise, symmetric addressing never changes a type, and an out-of-range target is rejected statically rather than at run time. Rule t-notify requires pointer, rank, and signal-value operands of types $\mathsf{i64}_1$, $\mathsf{i32}_1$, and $\mathsf{i64}_1$, and has unit type $\mathsf{i32}_0$; the set/add flag b and the scope s carry no typing obligation, so both notification forms are typed by the same rule. Rule t-wait imposes the same three operand types but returns $\mathsf{i32}_1$, the scalar token type; its scope and ordering arguments are likewise unconstrained. Rule t-consume is polymorphic in the value type τ: it requires only that the token operand has type $\mathsf{i32}_1$, returns the value operand's type, and constrains neither the value's constructor nor the token's metadata. Rules t-load and t-store propagate the operand type to the result, and t-store additionally requires the address and the stored value to agree, returning unit. Every rule therefore either preserves the type of a data operand or produces a scalar of known dtype and size, and no rule inspects a value's runtime constructor.

Two consequences of this formulation are used later. First, the permissiveness about pointer-versus-scalar shape is what lets the runtime `Eval` constructors carry the pointer

obligation, so that a load whose address has the right element type but no prior wait is statically well typed yet blocked dynamically. Second, because a token with fabricated metadata satisfies `HasType`, no typing derivation on its own rules out the negative traces of Section 5.2; only the capability invariants do. These rules are a direct mathematical rendering of the constructors of `HasType`; they are not an independently implemented type checker.

3.2.2 Runtime typing judgments

Static typing is paired with three runtime predicates, one for each of value, environment, and store. The first, `ValueHasType W v` τ, is defined by cases on the value: a tile has the declared type and a payload whose length equals the type size; integer values have exactly their scalar types; a pointer has the declared element type and an owner below W; a token has scalar i32 type regardless of metadata; and unit has the zero-sized i32 type. A token therefore cannot express its protocol metadata through ordinary type identity.

The second, `EnvHasType` Γ ρ, is pointwise: whenever Γ assigns a type to a variable and the environment assigns a value, that value satisfies `ValueHasType`,

$$\frac{\Gamma.vars(x) = \mathsf{some}\ \tau \quad \rho(x) = \mathsf{some}\ v}{\mathsf{EnvHasType}(\Gamma, \rho)\ \mathsf{requires}\ \mathsf{ValueHasType}(\Gamma.world, v, \tau)}.$$

The third, `StoreWellTyped` σ, quantifies over every in-range declared cell with a present memory value and requires that value to have the declared type. Together the predicates state the runtime portion of type safety without requiring every variable to be initialized, and they make load preservation direct: the load rule supplies the cell lookup, while `StoreWellTyped` supplies value typing.

3.3 Expression evaluation

The relation $\sigma, \rho \vdash e \Downarrow r$, implemented by `Eval`, returns either $\mathsf{value}(v)$ or blocked. Variable lookup returns the environment value. A rank query returns the current rank when it is in range, and a world-size query returns the state world. Symmetric addressing takes a symmetric pointer and a valid target and returns a pointer with the target rank while preserving element type and address.

Notification evaluation checks for a symmetric i64 signal pointer and an in-range rank, returning unit; its signal update is represented by the transition semantics. A ready wait checks a pointer, positive count, expected value, and `NotificationVisible`; it returns a token carrying signal address, expected value, scope, and ordering. A blocked wait checks a positive count and a signal below the expected value and returns `blocked`. Rank-query axis arguments are ignored. Expression-level consume looks up an arbitrary value and a token-shaped value and returns the former; it does not validate the token's address, observed value, scope, or ordering against any other operand or capability. The expression rule does not require that the token originated from a wait; that stronger condition belongs to `Step.consume`. Load checks a symmetric pointer, valid rank, matching cell type, and present memory value.

3.4 Expression-level metatheory

The evaluator is a partial function: a terminating evaluation determines its result. The remaining expression-level results relate static typing to the runtime values that evaluation

produces, and they all share the same shape, which is preservation. Writing Γ for the typing context of the surrounding program and σ for the state in which the expression is evaluated, each such result assumes that Γ agrees with the state on the world size and that the environment is well typed, and concludes that the value produced at a typed subexpression satisfies the runtime typing judgment at the type assigned to that subexpression. The four cases that matter are variable lookup, rank query, wait, and load, together with symmetric addressing, which is the only expression that manufactures a pointer.

All five results are stated formally as Lemmas 2–7 in Section 4.1, and their proofs are discussed in Section 4.2. We separate statement from proof this way because the interesting content of these results is not the induction but the hypothesis each one demands.

Three scope remarks belong here. First, the results are local: they quantify over one expression in one fixed state and environment, so they say nothing about how a distributed execution schedules the ranks that evaluate that expression. Second, they concern shape rather than provenance. A value can satisfy the runtime typing judgment and carry no synchronization history at all, which is why the capability invariant needs a separate theorem over the transition trace.

3.5 Type-safety theorems

The artifact's type-safety results are preservation-style theorems for the encoded language and machine. They should be distinguished from the provenance theorem: type safety establishes that values have the declared shapes, whereas provenance establishes that a load has a synchronization history. The expression-level results were summarized in Section 3.4; here we define the machine predicate they preserve, state the step and trace theorems, and record the boundary of the typing discipline.

Together the expression results say that a well-typed lookup returns a value of its context type, that rank and wait queries return values of the scalar and token types respectively, that symmetric addressing yields a pointer of the input element type whose target lies in range, and that a successful load returns a value carrying the store's declared cell type. Two limitations are worth stating up front. The wait result being typed at the static token type does not claim the token has a valid pending origin, and the load result being typed does not constrain which cell supplied it, because the evaluator reads the cell type from the state rather than carrying it in the result type.

▶ **Definition 1** (Well-typed machine). *A machine m is well typed under context Γ, written* $\mathsf{WellTypedMachine}(\Gamma, m)$*, if*

$$\Gamma.world = m.state.world \quad \wedge \quad \mathsf{EnvHasType}(\Gamma, m.env) \quad \wedge \quad \mathsf{StoreWellTyped}(m.state).$$

This packages the three runtime invariants of the predicate. The mechanized type-safety results are preservation theorems over it.

Theorem 8 (`step_preserves_well_typed`) establishes that each transition step preserves well-typedness. The proof is by cases on the step constructor. Notifications change only the signal function, and the point-update definitions preserve the world and all memory cells. Wait, consume, and load leave the state, environment, and cell typing unchanged. The theorem therefore exposes a useful limitation: the current transition system has no data-changing store step, so it proves preservation rather than a full subject-reduction theorem for mutable programs.

Theorem 9 (`trace_preserves_well_typed`) lifts the step result to arbitrary traces by induction on the trace derivation. The separate theorem `trace_preserves_world` proves

$$\frac{r < W}{m \xrightarrow{\mathsf{notifySet}(r,a,v,s)} m[\sigma.S(r,a) := v]}\ \textsf{step-notify-set} \qquad \frac{r < W}{m \xrightarrow{\mathsf{notifyAdd}(r,a,v,s)} m[\sigma.S(r,a) := \sigma.S(r,a) + v]}\ \textsf{step-notify-add}$$

$$\frac{n \le S(r,a) \qquad \mathsf{NotificationVisible}(\sigma, r, a, s)}{m \xrightarrow{\mathsf{wait}(r,a,n,t,s,o)} m[Wit := w :: Wit,\ Pen := p :: Pen]}\ \textsf{step-wait}$$

$$\frac{\rho(x) = \mathsf{ptr}(\tau, r, a, \mathsf{true}) \quad c(x,t,a,r) \in Con}{m \xrightarrow{\mathsf{load}(x)} m}\ \textsf{step-load}$$

$$\frac{\begin{array}{c}\rho(x) = \mathsf{ptr}(\tau, r, a, \mathsf{true}) \qquad a' = a \\ \rho(t) = \mathsf{token}(a', n, s, o) \\ \exists p \in Pen.\ p.\mathsf{tokenVar} = t\ \wedge\ p.\mathsf{signalRank} = r \\ \wedge\ p.\mathsf{signalAddr} = a\ \wedge\ p.\mathsf{scope} = s\ \wedge\ p.\mathsf{ordering} = o\end{array}}{m \xrightarrow{\mathsf{consume}(x,t)} m[Pen := \mathsf{filter}_t(Pen),\ Con := c(x,t,a,r) :: Con]}\ \textsf{step-consume}$$

Figure 2 The five transition constructors of the relation $m \xrightarrow{a} m'$. Each rule is one constructor of the inductive predicate `Step` in `Transition.lean`; witnesses are $w = (r, a, n, s, o)$, pending records $p = (t, r, a, n, s, o)$, and consumed records $c(x, t, a, r)$. The rules never modify the environment ρ.

that the state world remains equal to the context world along the same trace. These are the paper's explicit type-safety theorems. They establish preservation of the formal runtime invariants under every admitted transition, not safety of arbitrary source programs or generated accelerator code. The transition-level behaviour that they preserve is specified by the five rules of Section 3.7.

3.6 Trace semantics

A trace is a list-indexed inductive relation generated by single-step transitions. The semantics distinguishes an action sequence from a mere collection of state facts, allowing provenance to be stated over the complete trace. The calculus models no state-changing data-store transition. It represents notification state and abstract data-pointer metadata in a simplified address namespace.

3.7 Operational rule schemas

The transition relation is generated by the five constructors of `Step`, rendered in rule form in Figure 2. As with the typing rules, the figure is a description of the Lean constructors rather than a second semantics; each rule is named after the constructor it renders, with a step- prefix. Write $m = (\sigma, \rho, Wit, Pen, Con)$ for a machine, where Wit, Pen, and Con are the witness, pending, and consumed lists, W for the world size, and $S(r, a)$ for the signal counter at rank r and address a. Every rule leaves the environment ρ untouched; only the two notification rules modify the state, and only the wait, consume, and load rules touch the capability lists.

Rules step-notify-set and step-notify-add are the only state-changing transitions: the first assigns the counter at (r, a) to v, the second increments it by v, and both require only that the target rank lie below the world size. Neither consults the environment or the capability lists, so the constructors check no pointer binding and no operation tag: the action already fixes the operation together with the abstract rank, address, and value, and pointer evaluation and source-level operation selection lie outside this transition layer. Keeping set and add

as distinct constructors is deliberate, because replacing one by the other changes the signal history even when the final counter happens to coincide.

Rule step-wait requires the expected value n to be no greater than the current counter and the explicit predicate $\mathsf{NotificationVisible}(\sigma, r, a, s)$ to hold; it then prepends the witness $w = (r, a, n, s, o)$ and the pending record $p = (t, r, a, n, s, o)$. The two records serve different purposes, which is why only the second carries the token variable: the witness denotes the synchronization event, whereas the pending record is the resource that a later consume must spend. Unlike expression-level wait evaluation, this transition imposes no positive-count premise, and since the rule neither creates nor binds a runtime token nor inspects the binding of t, no rule connects the token value returned by an expression-level wait to an environment update. A later consume therefore relies on a token-shaped value that already exists in the environment, which every transition rule preserves.

Rule step-consume is the most constrained: it requires the value variable x to hold a symmetric pointer, the token variable t to hold a token, the two addresses to agree, and some pending record to match the token variable, signal rank, signal address, scope, and ordering. It then removes the pending records for t, written filter_t, and prepends the consumed record $c(x, t, a, r)$. The match does not compare the token's observed value with the pending record's expected value, and the consumed record retains only the value variable, token variable, signal address, and rank, so scope and ordering are checked at consume time but neither retained nor re-matched by the later load. This asymmetry is a conscious abstraction and is discussed in the limitations section.

Rule step-load is a permission self-loop: it fires only for a symmetric pointer that already has a matching consumed record, and it leaves the machine unchanged because the value delivered to the program comes from the expression evaluator rather than from the transition relation. The self-loop still matters: it lets the theorems quantify over a final action and recover that action's history from the capability the premise demands.

3.8 Capability records and ownership

There are three related notions of identity in the model. A signal location is identified by rank and address. A token is identified by its environment variable and runtime metadata. A consumed ordering is identified by the value variable, token variable, signal address, and rank. These identities should not be conflated. For example, two token variables may contain equal runtime tokens, while only one may have a pending capability in the machine.

The ownership map is used by `NotificationVisible`: the owner must be below the world size. Together with a positive signal counter and the scope check, this supplies a basic abstract premise. It does not encode a distributed ownership protocol, require the current rank to equal the owner, or establish visibility of data memory.

The world size occurs in several premises rather than in one global invariant. Pointer values require their owner to be in range; symmetric addressing requires its target to be in range; notification transitions require the signal rank to be in range; and visibility requires the signal owner to be in range. This repetition is useful in the Lean proofs because each constructor exposes the exact arithmetic fact it needs. It also identifies a future refactoring opportunity: a dependent location type could package these bounds, but would make the current examples and theorem statements less direct.

3.9 Invariant decomposition

The main safety argument is the conjunction of four smaller facts, each of which is a separate declaration in the development. We state them here as schemas, using $m \xrightarrow{a} m'$ for a single transition labelled a and $m \xrightarrow{A} m'$ for a trace, and we name the corresponding Lean declaration with each schema.

The two shape invariants are the following.

(I1) World stability. $m_1 \xrightarrow{A} m_2 \implies m_2.\mathit{state}.\mathit{world} = m_1.\mathit{state}.\mathit{world}$

(I2) Environment stability. $m \xrightarrow{a} m' \implies m'.\mathit{env} = m.\mathit{env}$

(I1) is `trace_preserves_world`, which is stated relative to a context and proved by induction on the trace; the step case is immediate because no transition constructor touches the world field. (I2) is not a separate declaration: it follows definitionally, because no constructor of `Step` rebinds the environment. Together they justify treating the world size and the environment as static for a whole execution, which is what allows the expression lemmas of Section 4.1, whose premises mention $\Gamma.\mathit{world}$, to be applied inside a trace.

The two protocol invariants are the following.

(I3) Capability origins. $\mathit{pending}(m_1) = [\,] \wedge \mathit{consumed}(m_1) = [\,] \wedge m_1 \xrightarrow{A} m_2$ $\implies$ every record of m_2 has a wait, resp. wait–consume, origin in A

(I4) Load admissibility. $m \xrightarrow{\mathsf{load}(x)} m' \implies \exists c \in \mathit{consumed}(m).\ c.\mathit{valueVar} = x$

(I3) is Theorem 11 in the general form that carries an arbitrary history prefix; the displayed version is its specialization to the empty prefix, which is the form used for the central result. (I4) is `load_requires_consume_token`, a corollary of `load_requires_ordering_witness`, which additionally recovers the pointer type, rank, and address and shows that the consumed record agrees with the pointer on rank and address.

The first two facts support ordinary typing and are independent of the wait protocol. The third fact is generated by wait and consume. The fourth fact is the final interface between the protocol and data access. Separating them prevents a common proof error: using a well-typed pointer as if it already carried synchronization permission. In this calculus, pointer typing proves shape and rank bounds; consumed capability proves history.

The decomposition also explains why the initial-state hypothesis matters. If the initial machine were allowed to contain arbitrary pending or consumed records, the origin theorem would need an explicit origin history for those records. The universal load theorem instead assumes both lists are empty, so all capabilities used by a load must have been generated by the trace itself.

3.10 Blocked evaluation versus absent transitions

The expression relation includes `blocked` as a result because a wait may be evaluated before its signal reaches the expected value. A blocked result is not a transition: it does not add a witness, does not add a pending token, and does not advance a machine trace. This distinction is reflected in the Lean development by separate constructors for `waitBlocked` and the successful wait transition.

The distinction is particularly important for progress claims. From a blocked evaluation one can conclude that the observed signal is below the expected value, but one cannot conclude that the signal will eventually increase. From a successful evaluation one can extract

visibility and a token, but one cannot conclude that a consumer will execute consume. The artifact therefore proves a conditional local progress characterization rather than a liveness theorem.

3.11 Relationship between expression and transition premises

Expression evaluation and transition execution share the state and environment vocabulary but are not definitionally identical. A successful expression wait returns a token value. Independently, transition wait records a successful wait in the machine's capability lists without updating the environment; no rule binds the expression result to that environment. The two consume rules differ in the asymmetry described at the end of Section 3.3: the expression rule checks the shape of its operands only, whereas the transition rule matches the pointer against the pending record.

This two-level design permits negative examples that are useful for auditing. An environment can contain a correctly shaped token and therefore satisfy the expression consume rule, while no transition consume is possible because the pending list is empty. Conversely, a pending capability alone is insufficient for consume if the environment token or pointer metadata does not match. The formalization thus distinguishes shape errors, readiness errors, and provenance errors rather than collapsing them into one failure result.

4 Mechanized Results

The results are organized into three layers. Core-level lemmas reason about values produced by one expression. Step-level lemmas reason about the shape of one enabled action. Trace-level lemmas lift the step facts to arbitrary finite executions. This organization mirrors the Lean files and avoids hiding a transition premise inside an informal operational argument.

4.1 Principal theorems and lemmas

The mechanized results are stated as theorems and lemmas over the transition system. We give the key contracts formally, using the notation of Section 3 for the judgment forms; each statement below corresponds to one closed `theorem` in the Lean development, and the corresponding declaration name is given in the accompanying discussion so that a reader can locate it in the artifact.

▶ **Lemma 2** (Evaluation determinism). *For all states σ, environments ρ, expressions e, and results r_1 and r_2,*

$$\sigma, \rho \vdash e \Downarrow r_1 \quad \wedge \quad \sigma, \rho \vdash e \Downarrow r_2 \quad \Longrightarrow \quad r_1 = r_2.$$

Evaluation is deterministic: one expression in a fixed state and environment produces exactly one result. This is mechanized as `eval_deterministic` and proved by case analysis on the two derivations.

▶ **Lemma 3** (Variable preservation). *If $\Gamma \vdash \mathsf{var}(x) : \tau$, $\mathsf{EnvHasType}(\Gamma, \rho)$, $\Gamma.\mathit{world} = \sigma.\mathit{world}$, and $\sigma, \rho \vdash \mathsf{var}(x) \Downarrow \mathsf{value}(v)$, then*

$$\mathsf{ValueHasType}(\sigma.\mathit{world}, v, \tau).$$

A typed variable lookup returns a value of the context type. This is the fundamental lemma for environment reasoning, mechanized as `eval_var_preservation`.

▶ **Lemma 4** (Load preservation). *For all states σ, environments ρ, pointers p, and values v, if* StoreWellTyped(σ) *and* $\sigma, \rho \vdash \mathsf{load}(p) \Downarrow \mathsf{value}(v)$*, then*

$$\exists\, \tau.\ \mathsf{ValueHasType}(\sigma.world, v, \tau).$$

A successful load is covered by store typing: the returned value has the declared cell type. This is `eval_load_preservation`, which uses the store well-typedness premise to establish the value's type. The conclusion is existential because the load rule obtains the cell type from the state rather than carrying it in the result type; the `Rank` and `Addr` witnesses that the declaration also introduces are unused in the statement.

▶ **Lemma 5** (Rank preservation). *For all states σ, environments ρ, and integers axis, if*

$$\sigma, \rho \vdash \mathsf{rank}(axis) \Downarrow \mathsf{value}(\mathsf{i32}(\sigma.currentRank)),$$

then

$$\mathsf{ValueHasType}\big(\sigma.world,\ \mathsf{i32}(\sigma.currentRank),\ \langle \mathsf{i32}, 1\rangle\big).$$

A rank query returns an i32 scalar with size 1. The declaration `eval_rank_preservation` proves this by reflexivity: the left-hand side is fixed by the constructor alone. The axis argument is discarded, so the lemma says nothing about which axis was queried.

▶ **Lemma 6** (Wait preservation). *For all states σ, environments ρ, variables t, counts, expected values, scopes, and orderings, if*

$$\sigma, \rho \vdash \mathsf{wait}(t, \ldots) \Downarrow \mathsf{value}(\mathsf{token}(a, e, s, o)),$$

then

$$\mathsf{ValueHasType}\big(\sigma.world,\ \mathsf{token}(a, e, s, o),\ \langle \mathsf{i32}, 1\rangle\big).$$

A successful wait returns a value of the static token type. This is `eval_wait_preservation`, again by reflexivity. It is deliberately a typing statement only: it does not assert that the token has a pending origin, that its metadata matches any capability, or that a later consume will succeed.

▶ **Lemma 7** (Symmetric addressing safety). *For all states σ, environments ρ, variables x, types τ, ranks r, addresses a, and targets tgt, if*

$$\sigma, \rho \vdash \mathsf{symmAt}(x, tgt) \Downarrow \mathsf{value}(\mathsf{ptr}(\tau, tgt, a, \mathsf{sym})),$$

then

$$\mathsf{ValueHasType}\big(\sigma.world, \mathsf{ptr}(\tau, tgt, a, \mathsf{sym}), \tau\big).$$

Symmetric addressing returns a pointer whose element type is the input type and whose rank field lies inside the world, since ValueHasType on a pointer abbreviates exactly those two conjuncts. The proof of `eval_symmAt_safe` inverts the `Eval.symmAt` derivation and combines pointer typing with the target bound that the rule demands as a premise.

▶ **Theorem 8** (Step preservation). *If* WellTypedMachine(Γ, m) *and* Step(m, a, m')*, then*

$$\mathsf{WellTypedMachine}(\Gamma, m').$$

Each transition step preserves the runtime typing invariants: world size, environment typing, and store well-typedness. The proof (`step_preserves_well_typed`) is by cases on the step constructor. Notifications update only the signal function while preserving all cell types; wait, consume, and load leave the state and environment unchanged.

▶ **Theorem 9** (Trace preservation). *If* $\mathsf{WellTypedMachine}(\Gamma, m_1)$ *and* $\mathsf{Trace}(m_1, A, m_2)$*, then*

$$\mathsf{WellTypedMachine}(\Gamma, m_2).$$

Type safety lifts to arbitrary traces by induction on the trace derivation, applying Theorem 8 at each transition. The declaration is named `trace_preserves_well_typed`.

▶ **Theorem 10** (Wait progress). *Let* m *be a machine with signal state and let* r*,* a*,* e*,* s *be a rank, address, expected value, and scope such that* $\mathsf{NotificationVisible}(m.state, r, a, s)$*. Then for every token variable* t *and ordering* o*,*

$$\big(\exists m'.\ \mathsf{Step}(m, \mathsf{wait}(r, a, e, t, s, o), m')\big) \ \vee \ m.state.signals(r, a) < e.$$

Under the visibility premise, a wait is either enabled or the observed signal lies below the expected value; there is no third outcome. This is `wait_progress`, and it is a local characterization only. It does not give deadlock freedom, fairness, or collective progress, because the calculus does not guarantee that any rank eventually raises a signal.

A pending token's origin is a matching wait action; a consumed ordering's origin is a wait followed by a matching consume. Writing $\mathit{pending}(m)$ and $\mathit{consumed}(m)$ for the two capability lists of a machine, the two auxiliary predicates are

$$\begin{aligned}\mathsf{PendingHasWait}(h, p) &\triangleq \exists h_1\, h_2.\ h = h_1 {+\!+} [\mathsf{wait}(p)] {+\!+} h_2,\\ \mathsf{ConsumedHasWC}(h, c) &\triangleq \exists h_1\, h_2\, h_3\, e\, s\, o.\ h = h_1 {+\!+} [\mathsf{wait}(c, e, s, o)]\\ &\qquad {+\!+}\, h_2 {+\!+} [\mathsf{consume}(c)] {+\!+} h_3,\end{aligned}$$

and they lift to capability lists pointwise, giving

$$\begin{aligned}\mathsf{PendingTokensHaveOrigins}(h, \ell) &\triangleq \forall p \in \ell.\ \mathsf{PendingHasWait}(h, p),\\ \mathsf{ConsumedOrderingsHaveOrigins}(h, \ell) &\triangleq \forall c \in \ell.\ \mathsf{ConsumedHasWC}(h, c).\end{aligned}$$

The abbreviations $\mathsf{wait}(p)$ and $\mathsf{wait}(c, e, s, o)$ expand to the full action carrying the record's rank, address, expected value, token variable, scope, and ordering, and $\mathsf{consume}(c)$ expands to the action naming c's value variable and token variable; the record's remaining fields are not written into the action.

▶ **Theorem 11** (Capability origins). *For all history prefixes* h*, action lists* A*, and machines* m_1 *and* m_2*, if*

$$\begin{aligned}&\mathsf{PendingTokensHaveOrigins}(h, \mathit{pending}(m_1))\\ \wedge\ &\mathsf{ConsumedOrderingsHaveOrigins}(h, \mathit{consumed}(m_1))\\ \wedge\ &\mathsf{Trace}(m_1, A, m_2),\end{aligned}$$

then

$$\begin{aligned}&\mathsf{PendingTokensHaveOrigins}(h {+\!+} A, \mathit{pending}(m_2))\\ \wedge\ &\mathsf{ConsumedOrderingsHaveOrigins}(h {+\!+} A, \mathit{consumed}(m_2)).\end{aligned}$$

Capabilities retain their trace origins across transitions: a pending token keeps the wait that created it, and a consumed ordering keeps the wait and consume pair that produced it. This is `trace_preserves_capability_origins`, proved by induction on the trace derivation. Notification and load steps leave both capability lists unchanged, so the induction hypothesis applies after extending the history. A wait introduces one pending record whose origin is the current action. A consume removes the matching pending entries by filtering and introduces one consumed record; the transition premise supplies the pending token whose wait origin extends to a wait–consume decomposition. In every case an existing origin is extended by appending the current action, which is why the history in the conclusion is the concatenation $h{+\!+}A$ rather than h.

▶ **Theorem 12** (Universal load provenance). *For all machines m_0 and m_f, action lists A, and variables x, if*

$$\mathit{pending}(m_0) = [\,] \;\wedge\; \mathit{consumed}(m_0) = [\,] \;\wedge\; \mathsf{Trace}(m_0, A, m_f) \;\wedge\; \mathsf{Step}(m_f, \mathsf{load}(x), m_f),$$

then there exist lists A_{before}, A_{middle}, A_{after} and metadata τ, r, a, t, e, s, o such that

$$A = A_{before} {+\!+} [\mathsf{wait}(r, a, e, t, s, o)] {+\!+} A_{middle} {+\!+} [\mathsf{consume}(x, t)] {+\!+} A_{after}$$
$$m_f.\mathit{env}(x) = \mathsf{some}(\mathsf{ptr}(\tau, r, a, \mathsf{sym})).$$

This is the central result. From empty initial capability state, every permitted load has an earlier matching wait and consume in the action trace: the wait and consume share the token variable t, the consume's value variable is the loaded pointer variable x, and the consumed record matches its rank r and address a. The proof, mechanized as `load_has_prior_wait_and_consume`, applies Theorem 11 to the empty initial lists, extracts the consumed record that the load premise demands, and decomposes the action list around it. Observe that the decomposed trace need not be contiguous: the middle and suffix lists may contain further actions, so the obligation is the existence and order of the capability edge, not adjacency of the three operations.

The statement is deliberately narrow, and three omissions matter for reading it as a safety property. First, consume requires the token's address, scope, and ordering to match the pending record, but it does not compare the token's observed value with that record's expected value, and the consumed record retains neither the expected value nor the scope or ordering. Second, consumed records are never invalidated, and load is a permission self-loop, so one record can authorize repeated matching loads, including loads issued after later notifications or waits; there is no epoch freshness check. Third, the theorem concerns abstract capabilities only: it does not show that the wait made memory visible, that the compiler retained a data dependency, or that program execution terminates. Those are the obligations of the correspondence discussion in Section 6.

4.2 Proof structure and induction measures

The expression lemmas use inversion on the relevant `Eval` constructor. For example, inversion on a successful symmetric-addressing derivation exposes the pointer lookup and target-bound premise, so the conclusion follows by the definition of `ValueHasType`. The store lemma is similarly a direct application of the quantified premise in `StoreWellTyped`.

The transition lemmas use the concrete list operations in the machine. A wait step prepends one pending record and one witness; a consume step filters the pending list and prepends one consumed record. The key proof obligation is therefore not list membership

alone but preservation of the metadata relation between the environment and the capability record. The implementation proves this by destructing the matching pending record supplied by the consume rule.

The trace theorems use induction on the derivation rather than induction on an untrusted action list. The empty case establishes the initial capability origin assumption. In the step case, the induction hypothesis handles all pre-existing capabilities, while the action-specific lemma handles the one new capability introduced by wait or consume. The load theorem then applies the origin invariant to the consumed record required by the final load step and decomposes the action list into prefix, wait, middle, consume, and suffix.

This proof architecture is intentionally auditable. It makes the source of each premise visible in Lean and prevents a theorem about arbitrary lists from being mistaken for a theorem about enabled executions.

5 Examples and Source Validation

The Lean examples construct a producer state with a signal value and a consumer trace of `wait -> consume -> load`, starting with empty capability lists. Negative results show that a load is not derivable from a machine lacking a consumed capability, and that an environment token alone cannot authorize consume when no pending wait capability exists. These examples demonstrate the transition rules, while the universal theorem supplies the general trace property.

5.1 Concrete positive trace

We fix a two-rank instance. Writing $S(r, a)$ for the signal at rank r and address a, the machine m_0 is defined by

$$\begin{aligned} &\mathit{world} = 2, \qquad \mathit{currentRank} = 0, \qquad S(1,7) = 1, \\ &\mathit{env} = \left\{ \begin{array}{l} 0 \mapsto \mathsf{ptr}(\langle \mathsf{i64}, 1\rangle, 1, 7, \mathsf{sym}), \\ 1 \mapsto \mathsf{ptr}(\langle \mathsf{i32}, 1\rangle, 1, 7, \mathsf{sym}), \\ 2 \mapsto \mathsf{token}(7, 1, \mathsf{system}, \mathsf{acquire}) \end{array} \right\}, \\ &\mathit{witnesses} = \mathit{pending} = \mathit{consumed} = [\,]. \end{aligned}$$

The signal owner is the rank itself, so NotificationVisible holds at $(1, 7)$. The machine contains no producer notification and no data-store transition, and the token value at variable 2 preexists the wait: the wait does not create or bind it. The environment is fixed for the whole example, and m_0 is the machine with all three capability lists empty; the negative cases of Section 5.2 reuse m_0 and the witness list produced below.

▶ **Lemma 13** (Consumer protocol instance). *There exist machines m_1 and m_2 such that*

$$\begin{aligned} &\mathsf{Step}(m_0, \mathsf{wait}(1,7,1,2,\mathsf{system},\mathsf{acquire}), m_1) \\ &\wedge\ \mathsf{Step}(m_1, \mathsf{consume}(1,2), m_2) \ \wedge\ \mathsf{Step}(m_2, \mathsf{load}(1), m_2). \end{aligned}$$

The three steps are exactly the protocol shape that the provenance theorem abstracts over. The wait is enabled because $1 \le S(1,7)$ and visibility holds; it prepends the witness $(\mathsf{sysRank}\ 1, \mathsf{sysAddr}\ 7, \mathsf{expected}\ 1)$ and the pending record $(t = 2, r = 1, a = 7, e = 1, s = \mathsf{system}, o = \mathsf{acquire})$. The consume step matches variable 1's pointer against the pending record on rank, address, scope, and ordering, so it removes that record by filtering and

prepends the consumed record ($valueVar = 1$, $tokenVar = 2$, $a = 7$, $r = 1$). The load then succeeds because the consumed record agrees with the pointer on rank and data address. Observe that the load is a self-loop: m_2 is left unchanged, since the value the expression produces comes from the evaluator, not from the transition relation. The three-step trace is mechanized as `end_to_end_wait_consume_load_trace`, and the notification half as `producer_notify_set_example`, which is the only statement that exercises a signal-raising transition.

5.2 Negative traces and blocked waits

The universal theorem is only as strong as the negative cases it excludes, so the development proves three non-derivability results. Two auxiliary machines are m_0 of Section 5.1 and m_w, which agrees with m_0 on state and environment but carries as its witness list the single wait witness that Lemma 13 creates. Both have empty pending and consumed lists, so the two differ only in whether the wait has been recorded.

▶ **Lemma 14** (Load requires a consumed capability). *For m_w as just defined,*

$$\neg\, \exists m'.\ \mathsf{Step}\big(m_w, \mathsf{load}(1), m'\big).$$

Lemma 14 is the direct contrapositive of invariant (I4) of Section 3.9: m_w has an empty consumed list, so the existential that `Step.load` demands cannot be inhabited. The declaration is `load_without_consume_is_not_derivable`. The pointer at variable 1 is well typed, symmetric, and in range, and the wait is already recorded, which is precisely the point: neither pointer typing nor a witness authorizes the load.

▶ **Lemma 15** (Load requires a wait). *For m_0 of Section 5.1,*

$$\neg\, \exists m'.\ \mathsf{Step}\big(m_0, \mathsf{load}(1), m'\big).$$

The difference from Lemma 14 is instructive. Both machines have empty consumed lists, so both fail for the same local reason, but m_0 additionally lacks the witness that a successful wait would have created. The declaration `load_without_wait_is_not_derivable` therefore separates the capability premise from the witness premise, confirming that the guard on `Step.load` is the consumed record rather than the witness list.

▶ **Lemma 16** (Fabricated token). *For m_0 as above,*

$$\neg\, \exists m'.\ \mathsf{Step}\big(m_0, \mathsf{consume}(1,2), m'\big).$$

Here the environment still contains a token-shaped value at variable 2, and expression-level consume would happily return variable 1's value, because that rule checks shape only. The transition-level premise fails: no pending record with token variable 2 exists, so the filtering antecedent is uninhabited. This is the sharpest illustration of why the scalar token type is weaker than provenance. The declaration behind it is named after that reading, as the proof that a fabricated environment token cannot authorize a consume.

A fourth case concerns readiness rather than capability. If the signal is below the expected value, the evaluator derives waitBlocked and `Step.wait` has no corresponding transition, which is the right-hand disjunct of Theorem 10. The model thus separates three failure modes that a single error code would conflate: *shape* errors, caught statically; *readiness* failures, which are legitimate blocking states; and *provenance* failures, which the two capability premises exclude. The development also fixes a total classification function *classify* on machines returning safe, blocked, or invalid, and checks that m_w classifies as safe and that m_0 with an all-zero signal classifies as blocked; these are `consumer_classification_safe` and `consumer_classification_initially_blocked`.

5.3 What the examples do not test

The examples are proof-producing unit cases, not distributed executions. They do not test network latency, multiple concurrent producers, signal reuse across epochs, allocation failures, compiler scheduling, or device memory behavior. They also do not establish that the source tutorial necessarily lowers to the same action sequence. Their role is to make the abstract premises and the counterfactual failures executable at the Lean level.

A separate Python validator parses selected upstream tutorial files using the standard-library AST module. It checks three source patterns: a notify/wait producer–consumer example, an intra-node all-gather, and an overlapping all-gather/GEMM example. Mutation checks reject selected in-memory source alterations, including removal of token consumption or signal-bearing communication. This validator checks names and relative source-call shapes. It is not an execution test, a translation validator, or evidence about generated IR or hardware behavior.

5.4 Artifact and reproducibility

The formal artifact is a Lake project pinned to Lean 4.19.0. Building the project with `lake build` checks the core, transition semantics, and examples together. The source audit is run with Python 3 and an explicit local source root; pass the path to the audited checkout explicitly with `python3 validation/validate_source_patterns.py -source-root PATH/TO/Triton-distributed-ascend -mutate`. The validator reports 12 positive source-pattern checks and, with `-mutate`, three additional negative mutation checks. The validator uses only the Python standard library. The revision string is reported for provenance but is not checked against Git; reproduction requires checking out revision `1e6d688` first.

The Lean source contains no `sorry`, `admit`, or user-declared `axiom`. This statement concerns the checked artifact, not explicit theorem premises such as `NotificationVisible`; these assumptions are deliberately exposed in theorem statements. The paper's PDF is built using the official `lipics-v2021` class and the `plainurl` bibliography style.

6 Relationship to the Implementation

6.1 Correspondence obligations

The relationship between the source API and the calculus can be stated as a sequence of obligations. The first is syntactic: a source call must have the argument shape recognized by the audit. The second is typing: translated expressions must inhabit the corresponding `HasType` judgments. The third is enabling: the source execution must supply the signal and visibility premises required by a transition. The fourth is provenance: the token consumed by a data value must be the token generated by the matching wait. The fifth is refinement: source and target executions must agree on visibility and dependency preservation.

Only the middle part of this chain is present here. Lean defines what counts as a typed and enabled core execution, while the Python validator checks a small set of source shapes. There is no function from Python ASTs, Triton IR, or MLIR values to `Expr` values. In particular, a validator success does not construct an `Env`, `Ctx`, or `Machine` and cannot be used as a premise of a Lean theorem without an additional translation proof.

An eventual translation would need to specify how source objects become abstract addresses and variables. It would also need to state whether a source token is linear, affine, or freely copyable. The current model records pending capability consumption but does not encode uniqueness of a runtime token in the environment. A source-level proof could

enforce linear use or prove that duplicated token values cannot create an additional consumed ordering. This is a design choice rather than an implementation detail.

6.2 Compiler transformations

The abstract consume action represents a dependency edge between communication completion and a later data use. A compiler may preserve that edge by keeping a data dependency, attaching an ordering attribute, or lowering to a backend primitive with an equivalent effect. The calculus does not distinguish these strategies. It records the edge as a capability and proves that a load is guarded by it.

This abstraction does not prove optimizer preservation. Common-subexpression elimination could duplicate or merge token-producing expressions; dead-code elimination could remove a consume whose result appears unused; scheduling could move a load before communication; and lowering could erase scope or ordering metadata. A compiler correctness theorem would need an observation relation that treats the capability edge as semantically relevant. It would then show that every target load corresponding to a source load has a source wait and consume, or that a target primitive provides an equivalent guarantee.

6.3 Signal reuse and epochs

Signals are natural-number counters, and waits compare an expected value with the current counter. This supports monotone protocols such as waiting for one and later two, but it does not establish that a value belongs to the intended communication epoch. If a signal address is reused, an old value may satisfy a later wait unless allocation and epoch discipline are supplied externally.

The witness list records the expected value and can distinguish waits for different counter values. As noted after Theorem 12, consume does not compare the token's observed value with the pending record's expected value, and the consumed record retains neither that value nor the scope or ordering. Consumed records are also never invalidated, so a single record can authorize repeated loads after later notifications or waits. Consequently the load theorem proves a wait–consume history for the recorded association, but does not establish that a load observes the newest producer epoch. Strengthening it would require an epoch discipline, matching or invalidation rules, and the additional record fields those rules would use.

6.4 Alternative modeling choices

One alternative would encode protocol state in dependent types. A token type could carry its signal address and expected value, and consume could require a proof that the token was produced by wait. This would make invalid consume expressions unrepresentable, but would move the central result from trace reasoning into type formation. The present model keeps tokens scalar-typed and proves provenance as an invariant over enabled transitions, matching the fact that compiler IR values often carry less static information than a protocol requires.

A second alternative would model complete configurations with control-flow locations and multiple ranks. That would support interleaving and scheduling theorems, but would add machinery not needed for the local wait–consume–load property. The current machine represents one abstract protocol history and leaves interleaving to the supplied trace. A third alternative would expand `NotificationVisible` into a weak-memory model. Keeping it as an interface lets a backend-specific happens-before or message-passing relation replace the predicate while preserving the provenance proof's structure.

The model is informed by source operation signatures but abstracts several API dimensions. Rank and world-size queries are represented at a high level; pointer/count evaluation for waits is abstracted; the wait is represented by a scalar rank/address/expected-value tuple; and arbitrary SSA token dataflow is not modeled. The source operation `consume_token` accepts a value and token independently, while the calculus adds explicit matching premises. Ordinary loads are guarded by a capability in the calculus, whereas the Python-level load API does not generally reject a load lacking such a consume dependency.

Accordingly, the calculus is a selected abstraction for proof tractability, with no established general acceptance or refinement ordering relative to the implementation. As discussed in Section 2, valid separate-buffer and initial-zero-wait protocols need an explicit abstraction mapping, and the model's overshoot readiness differs from the pinned equality polls.

7 Assumptions and Limitations

The results rely on explicit premises and omitted mechanisms:

- `NotificationVisible` is assumed for successful waits; no CANN, CUDA, NVSHMEM, or ROCSHMEM rule is proved.
- Compiler preservation of the consume dependency through optimization and lowering is not established.
- Symmetric allocation agreement, signal initialization and epoch discipline, and valid collective participation are not modeled as global invariants.
- The model has no source/MLIR-to-core translation or simulation, no backend refinement, and no hardware execution semantics.
- The core does not prove deadlock freedom, fairness, buffer lifetime, general data-race freedom, or end-to-end kernel correctness.

A stronger result would require a typed source or IR subset, a translation into the core, a simulation theorem, and evidence or a formal refinement connecting visibility and dependency preservation to each supported backend. Targeted litmus tests and signal-reuse tests would complement, but not replace, those proofs.

8 Related Work

Distributed accelerator programming. Triton-Distributed provides a programming model for distributed GPU kernels, including communication and synchronization operations intended to overlap communication with computation [12]. TileLink studies tile-centric primitives and optimization techniques for generating such overlapping kernels [13]. These works address the programming model and performance of distributed accelerator computations; our goal is different. We isolate one wait–consume–load discipline and formalize the trace evidence required by an abstract machine before it permits a load. The calculus does not model the complete Triton-Distributed language, reproduce its compiler, or establish the performance or correctness of its kernels.

Choreographic programming and session types. Choreographic programming describes interactions from a global perspective: a protocol specifies the roles and communication events of a distributed computation, and endpoint programs are obtained by projecting that description. Multiparty session types provide related techniques for specifying structured conversations and checking that participants respect their communication protocols [5]. Such

approaches typically reason about the consistency of multiple participants and, in some settings, communication safety or progress.

The present calculus is adjacent to this line of work, but it is not a choreography or a multiparty session-type system. It has no global protocol syntax, endpoint projection, or collection of independently executing rank-indexed processes. Its trace is an abstract sequence of enabled actions, not a theorem that projected endpoints implement a global specification. The connection is instead at the protocol level: ranks and signal metadata identify participants and communication resources, while pending and consumed capabilities record local obligations induced by synchronization. A choreography-aware extension could define a global producer–consumer protocol, project it to rank-indexed actions, and prove that the projected traces are admitted by this calculus. The current provenance theorem could then serve as a local obligation in a larger projection or refinement proof.

Session-typed and separation-logic reasoning. Session-typed reasoning in Actris combines protocol specifications with separation logic, enabling modular reasoning about communication and shared resources [4]. Iris provides a general foundation for higher-order concurrent separation logic [6], while Aneris and Trillium develop mechanized reasoning for distributed systems and intensional refinement [8, 10]. These frameworks have substantially richer accounts of concurrency, resources, and composition than the transition system studied here. Our pending and consumed records are protocol resources in a narrower sense: they record the origin and use of a synchronization capability, but are not assertions in a separation logic and do not support modular reasoning about arbitrary concurrent components. The artifact therefore complements rather than implements these logical frameworks. General work on logical type soundness provides a broader account of the relationship between typing and semantic safety [11]; our preservation results concern only the specific runtime predicates of the encoded calculus.

Weak memory and synchronization semantics. A synchronization protocol is meaningful only relative to an account of ordering and visibility. Work on release–acquire reasoning and weak-memory fences studies how synchronization operations constrain observable executions [7, 1]. In our model, `NotificationVisible` is an explicit premise of a successful wait. The calculus proves that a permitted load has the required abstract wait–consume history; it does not derive visibility from a weak-memory semantics, nor does it establish that a particular CANN, CUDA, NVSHMEM, or ROCSHMEM operation implements that premise. This separation makes the backend obligation visible, but leaves its justification to a future target-specific semantics or refinement proof.

GPU verification, hardware models, and compiler correctness. GPUVerify checks classes of data races and barrier-divergence errors in GPU kernels [3]. Kami demonstrates mechanized hardware specification and modular verification [2], while CompCert proves semantic preservation for a defined compiler pipeline [9]. These works address different layers and broader properties than our abstract protocol theorem. In particular, we do not prove race freedom, hardware refinement, or preservation of synchronization dependencies through compiler optimization and lowering. The capability trace provides a candidate intermediate-level property for such a proof: a compiler account could relate source or IR dependencies to the abstract consume capability and then show that the target implementation preserves the corresponding ordering.

Positioning. The contribution is thus narrower than choreographic programming, session typing, weak-memory verification, GPU-kernel verification, or compiler verification. It is a mechanized local protocol model with a provenance invariant: from empty initial capability state, every load admitted by the abstract transition relation has an earlier matching wait and consume. Connecting this result to global protocol specifications or executable Triton-Distributed kernels requires additional endpoint projection, source-to-core translation, compiler-preservation, and backend-refinement results.

9 Conclusion

We presented a Lean core calculus for a wait–consume–load synchronization discipline motivated by Triton-Distributed. The mechanized provenance invariant and universal load theorem show that the transition system itself requires a matching earlier wait and consume before it permits a load. A source-pattern audit checks selected examples and exposes, rather than closes, the gap between source operations and the abstract calculus. The immediate technical value is a precise statement of the obligations a future source-to-core and backend-refinement account must discharge. Establishing such a connection, and validating target-specific memory-ordering assumptions, remains future work.

Data Availability. The accompanying source tree contains the Lean source, validation script, source audit, and build instructions. The artifact is built with Lean 4.19.0 and Lake; source-pattern validation uses Python 3 and the standard library. The audited upstream revision is recorded as `1e6d688`; the audit reads checkout working files and does not verify Git history.

References

1. Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Fences in weak memory models. In *Computer Aided Verification*, volume 6174 of *Lecture Notes in Computer Science*, pages 258–272, 2010. doi:10.1007/978-3-642-14295-6_24.
2. Adam Chlipala. Kami: A platform for high-level parametric hardware specification and its modular verification. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–14, 2017. doi:10.1145/3062341.3062350.
3. Alastair F. Donaldson, Darren Liew, Cedric Fumex, et al. Gpuverify: A verification framework for gpu kernels. In *Proceedings of the 2012 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 113–132, 2012. doi:10.1145/2384616.2384654.
4. Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. Actris: Session-type based reasoning in separation logic. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 672–687, 2020. doi:10.1145/3385412.3385970.
5. Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *Journal of the ACM*, 63(1):9:1–9:67, 2016. doi:10.1145/2827695.
6. Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Bart Jacobs, Jean Pichon-Pharabod, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, 2018. doi:10.1017/S0956796818000151.
7. Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. Reasoning under release-acquire consistency. *Electronic Proceedings in Theoretical Computer Science*, 263:1–15, 2017. doi:10.4204/EPTCS.263.3.

8 Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, Simon Oddershede Gregersen, and Lars Birkedal. Aneris: A mechanised logic for modular reasoning about distributed systems. In *Programming Languages and Systems*, volume 12075 of *Lecture Notes in Computer Science*, pages 336–365, 2020. doi:10.1007/978-3-030-44914-8_13.

9 Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009. doi:10.1145/1538788.1538814.

10 Amin Timany, Simon Oddershede Gregersen, Leo Stefanesco, Jonas Kastberg Hinrichsen, Leon Gondelman, Abel Nieto, and Lars Birkedal. Trillium: Higher-order concurrent and distributed separation logic for intensional refinement. *Proceedings of the ACM on Programming Languages*, 8(POPL):241–272, 2024. doi:10.1145/3632851.

11 Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. A logical approach to type soundness. *Journal of the ACM*, 71(6):Article 40, 2024. doi:10.1145/3676954.

12 Size Zheng, Wenlei Bao, Qi Hou, Xuegui Zheng, Jin Fang, Chenhui Huang, Tianqi Li, Haojie Duanmu, Renze Chen, et al. Triton-distributed: Programming overlapping kernels on distributed ai systems with the triton compiler, 2025. URL: https://arxiv.org/abs/2504.19442, arXiv:2504.19442.

13 Size Zheng, Jin Fang, Xuegui Zheng, Qi Hou, Wenlei Bao, Ningxin Zheng, Ziheng Jiang, Dongyang Wang, Jianxi Ye, et al. Tilelink: Generating efficient compute-communication overlapping kernels using tile-centric primitives, 2025. URL: https://arxiv.org/abs/2503.20313, arXiv:2503.20313.