Ora Type System Specification v0.1

A Region-Aware, Refinement-Typed Smart Contract Language for the EVM

Nuno Machado
seamac@gmail.com

January 1, 2026

Abstract

Ora is a strongly typed, region-aware, refinement-based smart contract language targeting the Ethereum Virtual Machine (EVM). Its core type system combines region types, refinement types, mutability with invariant-preserving assignments, an effect system for storage writes, and a transaction-scoped locking discipline to prevent multi-write reentrancy patterns.

This document specifies the core calculus and type system of Ora, focusing on the foundations that must be preserved across all lowerings to intermediate representations (IR) and ultimately to EVM bytecode.

1 Introduction

Smart contracts deployed to the EVM are notoriously difficult to reason about. Existing languages such as Solidity and Vyper provide only coarse-grained type systems, weak guarantees about reentrancy, and largely ad-hoc treatment of memory regions and persistent state. Correctness arguments are typically pushed into informal reasoning, off-chain tooling, or manual audits.

Ora is designed to address these shortcomings by:

- Making *regions* first-class in the type system, so that the language explicitly distinguishes between persistent `storage`, transient transaction-scoped state, `memory`, and `calldata`.
- Using *refinement types* to express and enforce invariants on values and storage, with proofs supported by SMT solvers.
- Supporting *mutable refinement types* through a hybrid static–dynamic model: invariants are discharged statically when possible; when proof is not possible or not economical, the compiler inserts explicit runtime checks.
- Providing an explicit *effect system* and *locking model* that expose and constrain stateful operations, especially writes to `storage`.

The goal is to provide a core language in which safety properties, such as the absence of certain reentrancy patterns, the preservation of global invariants, and the disciplined use of regions, can be stated and mechanically enforced. High-level syntactic sugar and library constructs are intended to desugar to this core while preserving its guarantees.

This specification presents the core calculus and the Ora type system. It is not a full language manual: many surface-language conveniences are omitted when they do not affect the semantic guarantees. Instead, the focus is on the core typing judgments, operational semantics, and refinement obligations that must hold for all well-typed Ora programs.

2 Design Goals

The Ora type system is designed around the following goals:

1. **Safety by construction.** Storage invariants should be guaranteed primarily at compile time through the type system, refinement types, and comptime evaluation. Reentrancy and multi-write races are intended to be statically ruled out for all well-typed programs within the safe core of the language (i.e., code that does not bypass the type system via raw EVM escapes).

2. **Comptime-first verification (Zig-inspired).** Ora follows a Zig-style `comptime` philosophy: as much reasoning as possible happens at compile time using the same language and syntax as runtime code. Invariants, pre- and postconditions, and many calculations are intended to be validated or executed at compile time, with runtime checks treated as a fallback when proof is not possible or not economical. The type system, refinements, effect analysis, and comptime execution are integrated so that rich static reasoning does not require a separate meta-language.

3. **Decidability and SMT-friendliness.** Refinements are restricted to decidable quantifier-free logical fragments suitable for automated SMT solving. Type checking and refinement checking must remain decidable and fast enough for practical compilation, even when combined with comptime execution and SMT queries. The design explicitly avoids language features that would force unbounded search or higher-order reasoning in the refinement logic.

4. **Zero-runtime overhead whenever possible.** When refinement obligations and invariants can be proven using the type system, SMT, and comptime evaluation, they incur no runtime overhead. Runtime assertions are generated only when static proof is not possible or not economical, and are treated as explicit, visible costs in the program surface language or its desugaring.

5. **Region separation and explicit side effects.** Region types ensure that movement and aliasing between `storage`, `memory`, `transient`, and `calldata` are explicit and statically checked. An effect system makes all `storage` writes (and other observable side effects) visible and analyzable, enabling comptime inspection of stateful behavior and compatibility with global invariants.

6. **Direct alignment with EVM bytecode semantics.** Regions and the storage model are designed to map directly to EVM execution primitives (`SLOAD`, `SSTORE`, `MLOAD`, `MSTORE`, `TLOAD`, `TSTORE`, `CALLDATALOAD`, etc.). Core Ora terms are lowered to an intermediate representation (SIR) and then to EVM bytecode in a way that preserves the guarantees of the type, region, and effect systems (type soundness and preservation of verified invariants). Formalizing SIR and the full lowering pipeline is left for future work.[**Open question:** A future revision should introduce SIR and state a type-preserving translation from Ora into SIR.]

3 Informal Overview

[**Open question:** Decide later whether this section stays in the full spec only or also in a condensed overview version.]

Before presenting the formal syntax and rules, we briefly outline the key concepts and how they fit together.

Types and regions. Every runtime value has a *value type* (e.g., `u256`, `bool`) and a *region* (e.g., `@memory`, `@storage`). A located type is written

$$\tau@\rho,$$

where τ is a value type and $\rho \in \{\mathsf{memory}, \mathsf{transient}, \mathsf{storage}, \mathsf{calldata}\}$ is its region.

Refinements. A refinement type $\{x : \tau \mid \varphi\}$ describes values of type τ that satisfy predicate φ, where φ is a decidable logical predicate. For example, $\{b : \texttt{u256} \mid b > 0\}$ is the type of strictly positive integers.

Mutability. Variables are bound as `let` (immutable) or `var` (mutable). Immutable bindings behave like traditional `let` bindings; mutable bindings can be reassigned with assignments of the form $x := e$.

Mutable refinements. A mutable variable can have a refinement type, e.g.:

```
var bal: { b: u256 | b <= maxSupply } @memory;
```

On every assignment to `bal`, the predicate $b \leq$ `maxSupply` must be preserved. When this cannot be proved statically, the compiler inserts a runtime `assert` to guard the assignment. In v0.1, refinement predicates for mutable bindings are restricted to mention only the bound variable and comptime constants (such as contract-level constants); they cannot depend on other mutable program state. This keeps invariant checking local to each binding.

Effects. Typing judgments carry an effect annotation ϵ describing which storage slots may be written. In the core system:

$$\epsilon ::= \mathsf{Pure} \mid \mathsf{Writes}(S),$$

where S is a finite set of storage slots. Pure means "no storage writes".

Locks (transaction-scoped write discipline). A lockset Λ tracks which storage-backed variables are currently *write-locked* in the ongoing transaction. In the surface language, annotations like `@lock(x)` and `@unlock(x)` mark regions where a variable `x` must not be written again. These annotations desugar to core operations that add or remove the corresponding storage slot(s) from Λ.

Formally, once a slot s associated with `x` is in Λ, the safe core forbids any further writes to s until it is explicitly removed from Λ (via `@unlock(x)`) or the transaction finishes. This is not a concurrency primitive: it does not coordinate between threads, but instead enforces a single-write-after-lock discipline for selected variables within a transaction, which is used to rule out multi-write reentrancy patterns on those variables.

Typing environments and judgments. The main typing judgment has the form

$$\Sigma; \Gamma; \Lambda \vdash e : \sigma \,!\, \epsilon$$

where:

- Σ is the storage layout (declaring slots and their located types),
- Γ is the typing context (variables, mutability, and logical assumptions),
- Λ is the current lockset,
- e is the expression being typed,
- σ is its located type, and
- ϵ is its effect.

SMT and comptime integration. When type checking encounters a refinement obligation (for example, before a `store` or a refined assignment), the compiler:

- extracts the logical assumptions from Γ,
- optionally simplifies expressions via `comptime` evaluation, and
- asks an SMT solver to prove the predicate.

If the obligation cannot be proved and the context permits, the compiler emits an explicit runtime `assert` that enforces the refinement at execution time.

4 Syntax

We define the abstract syntax of regions, types, predicates, expressions, and environments. This is the core calculus that the surface Ora language desugars into.

4.1 Regions

$$\rho ::= \mathsf{memory} \mid \mathsf{transient} \mid \mathsf{storage} \mid \mathsf{calldata}$$

- memory standard EVM memory, local to the current call.
- $\mathsf{transient}$ transaction-scoped, cleared at the end of the transaction (backed by `TSTORE`/`TLOAD`).
- $\mathsf{storage}$ persistent contract storage (backed by `SSTORE`/`SLOAD`).
- $\mathsf{calldata}$ immutable call input (backed by `CALLDATALOAD`).

4.2 Base Types

$$\beta ::= \texttt{u256} \mid \texttt{bool} \mid \texttt{address} \mid \texttt{bytes32} \mid \cdots$$

β ranges over primitive value types. The ellipsis stands for other fixed-size EVM friendly base types (e.g., unsigned integers of other widths).

4.3 Value Types

$$\tau ::= \beta \mid \{x : \tau \mid \varphi\} \mid \tau \to \tau \mid \cdots$$

- $\{x : \tau \mid \varphi\}$ is the set of values of type τ that satisfy predicate φ.
- Function types $\tau_1 \to \tau_2$ describe value-level functions in the core calculus. In v0.1, effects are tracked by the typing judgment rather than in the arrow type itself. An extension with explicit effectful arrows $\tau_1 \xrightarrow{\epsilon} \tau_2$ is left as open design space.

The dots indicate future extensions (e.g., tuples and structs) that are omitted from the minimal v0.1 core.

4.4 Located Types

A *located type* packages a value type with a region:

$$\sigma ::= \tau @ \rho$$

Examples:

- $\texttt{u256 @memory}$ a 256-bit unsigned integer in memory.
- $\{b : \texttt{u256} \mid b \leq \texttt{maxSupply}\}\texttt{@storage}$ a refined integer value located in storage.

4.5 Mutability

$$\mu ::= \mathsf{imm} \mid \mathsf{mut}$$

- imm immutable binding (`let`).
- mut mutable binding (`var`).

Mutability is a property of a binding in the typing context, not of the type itself.

4.6 Effects

The core effect algebra is:

$$\epsilon ::= \mathsf{Pure} \mid \mathsf{Writes}(S)$$

where:

- Pure no `storage` writes.
- $\mathsf{Writes}(S)$ may write to a finite set of storage slots S.

We leave the exact representation of sets of slots abstract; in later sections, S is derived from the storage layout Σ.

4.7 Predicates

Refinement predicates φ are drawn from a decidable, quantifier-free logic over base types. We write the grammar at a high level:

$$\begin{aligned} \varphi &::= \mathsf{true} \mid \mathsf{false} \mid a \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \\ a &::= e_p \odot e_p \mid P(e_{p1}, \ldots, e_{pn}) \\ \odot &::= = \mid \neq \mid < \mid \leq \mid > \mid \geq \end{aligned}$$

- P ranges over a fixed set of uninterpreted predicate symbols, which can be used to abstract complex properties while remaining SMT-friendly.
- The expressions e_p are *pure* expressions, defined below.

4.7.1 Pure Expressions for Predicates

We write e_p for pure expressions that may appear inside predicates:

$$e_p ::= n \mid x \mid \mathsf{true} \mid \mathsf{false} \mid e_p + e_p \mid e_p - e_p \mid e_p * e_p \mid \cdots$$

These expressions are side-effect free: they do not contain `load`, `store`, function application, or any other effectful constructs. Boolean connectives such as $\wedge$, $\vee$, and $\neg$ appear only at the level of predicates φ, not inside e_p.

The exact logical fragment (e.g., QF-LIA with uninterpreted functions) is fixed at the implementation level and chosen to keep refinement checking decidable.

> ***Refinement logic / SMT reading.*** *For me: decide a precise SMT-LIB logic for Ora's refinements and write it down in the SMT Integration section.*
>
> *References:*
>
> - *Heule, CMU 15-311 SMT slides: overview of SMT, models, and theories.*
> - *SMT-LIB logics catalogue: official list of logics and their features.*
> - *Barrett, Tinelli et al. on SMT-LIB and SMT (optional background for formalizing Ora's logical fragment).*
> - *Z3 (or CVC5) user guides for concrete encodings of QF_LIA + UF.*
>
> *Action items:*
>
> - *Pick a default SMT-LIB logic (probably some QF_LIA + UF).*
> - *Explicitly list which operators from e_p map to which SMT symbols.*
> - *Clarify how contract-level constants (e.g. `maxSupply`) appear in the SMT environment.*

4.8 Expressions

The core expression syntax is:

$$e ::= n \mid \mathsf{true} \mid \mathsf{false} \mid x \mid \mathsf{fun}\ (x : \sigma) \to e \mid e(e) \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid \mathsf{var}\ x = e\ \mathsf{in}\ e \mid x := e \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \mathsf{load}\ s \mid \mathsf{store}\ s$$

- n ranges over integer literals.
- x ranges over term variables.
- s ranges over storage slot identifiers declared in the storage layout (see §4.9.1).
- `assert` φ raises a runtime error if φ does not hold at execution time.
- `if` e_1 `then` e_2 `else` e_3 evaluates e_1 first; if it is `true`, the result is e_2, otherwise e_3.
- `lock` s marks the storage slot s as write-locked for the remainder of the transaction or until a matching `unlock` s is executed.
- `unlock` s removes the write-lock on storage slot s (if present).

High-level constructs (e.g., loops, pattern matching, user-defined control operators) are considered syntactic sugar and are not part of the minimal core calculus in v0.1.

4.9 Environments

We use three main environments: storage layout, typing context, and lockset.

4.9.1 Storage Layout

$$\Sigma ::= \cdot \mid \Sigma, s : \sigma$$

- Σ declares each storage slot s and its located type σ.
- We assume Σ maps each s to at most one σ.
- The set of storage slot identifiers s is disjoint from the set of term variables x. Surface-language constructs such as mappings (e.g., `balanceOf[user]`) are compiled to concrete slots in Σ.

4.9.2 Typing Context

$$\Gamma ::= \cdot \mid \Gamma, x : (\sigma, \mu) \mid \Gamma, \varphi$$

- Bindings of the form $x : (\sigma, \mu)$ track both the located type and mutability of x.
- Bindings of the form φ record logical assumptions available to refinement checking (e.g., from prior `asserts` or control-flow conditions).

4.9.3 Lockset

$$\Lambda ::= \varnothing \mid \Lambda \cup \{s\}$$

- The lockset Λ is a finite set of storage slots that are currently *write-locked* in the ongoing transaction.
- In the operational semantics (and, optionally, in typing), each `lock` s extends Λ with s, and each `unlock` s removes s from Λ (if present).
- The safe core enforces that no `store` to a slot s is allowed while $s \in \Lambda$. Equivalently: once a programmer calls `@lock(x)` (locking the slot(s) for `x`), the code cannot perform further writes to `x` until a corresponding `@unlock(x)` or the end of the transaction.

This syntax forms the backbone of the core calculus. The next sections define typing judgments, effects, and operational semantics over these syntactic categories.

5 Typing Judgments and Effects

5.1 Typing Judgments

Typing judgment. *Introduce the main typing judgment $\Sigma; \Gamma; \Lambda \vdash e : \sigma \,!\, \epsilon$, where ϵ is an over-approximation of the storage slots that e may write. Then define well-formedness conditions for types, environments, and storage layouts.*

5.2 Basic Typing Rules

Basic rules. *Add rules for literals, variables, `let`/`var`, functions, application, and conditionals. Make sure effects are threaded properly (e.g., application should combine the effect of the function value and the argument) and that all rules are consistent with the syntax in Section 4.*

5.3 Typing with Effects

Effect discipline. *Specify how effects combine (e.g., sequencing, application), and how* Pure *vs* $\mathsf{Writes}(S)$ *is handled. For example, a sequential composition $e_1; e_2$ will typically have effect $\mathsf{Writes}(S_1 \cup S_2)$ if e_i has effect $\mathsf{Writes}(S_i)$, and* Pure *acts as the identity element for composition. Clarify whether function types remain $\tau_1 \to \tau_2$ or eventually become $\tau_1 \xrightarrow{\epsilon} \tau_2$.*

5.4 Assignments and Mutable Refinements

Mutable refinements. *Add explicit rules for assignments $x := e$ when x has a refinement type. For a mutable binding $x : (\{v : \tau \mid \varphi\}@\rho, \mathsf{mut})$ and assignment $x := e$, the typing rule should generate a refinement obligation that the new value satisfies φ (with v substituted by the result of e), discharged either statically by the SMT solver or dynamically via an inserted* `assert`*, as described in Section 7.*

6 Operational Semantics

6.1 Evaluation Configuration

Configurations. *Define the runtime configuration (global storage state, memory, transient state, lockset Λ, current expression). Decide how much of Σ is needed at runtime vs compile time only.*

6.2 Small-Step Semantics

Reduction rules. *Add small-step rules for the core constructs:* `if`*,* `let`*/*`var`*, function application,* `load`*,* `store`*,* `lock`*,* `unlock`*, and* `assert`*.*

The rules for `lock` *and* `unlock` *should update the lockset Λ. The rule for* `store` *s e should require that $s \notin \Lambda$; if $s \in \Lambda$, executing* `store` *s e yields a runtime error (or is rejected statically in the safe core).*

6.3 Runtime Errors and Lock Violations

Error states. *Specify the error states (failed* `assert`*, writes to locked slots, type-stuck configurations, etc.) and how they relate to the progress/preservation theorems later.*

7 Refinements and SMT Integration

7.1 Logical Environment Extraction

Logical Γ. *Define how to extract the "logical context" from Γ: which bindings become logical variables, which assumptions φ are fed directly to the SMT solver, and how e_p expressions are interpreted. Clarify how contract-level constants and comptime-known values are represented in the logical environment.*

7.2 Choice of SMT-LIB Logic

[Note: Use Heule's SMT slides and the SMT-LIB logics catalogue to choose a concrete SMT-LIB logic (e.g. some QF_LIA + UF variant).]

SMT-LIB target. *State the chosen SMT-LIB logic (e.g. QF_LIA + UF) and briefly justify the choice (decidability, support in Z3/CVC5, expressiveness for Ora's needs). List any operators from e_p that are* not *supported and must be disallowed or desugared.*

7.3 Encoding Refinement Obligations

Obligation → SMT. *Explain how each refinement obligation (e.g. before a* `store` *or a refined assignment) is translated to an SMT query:*

- *Extract assumptions from Γ.*
- *Map e_p to SMT terms.*
- *Ask the solver to prove validity or satisfiability of the relevant formula.*

Clarify which side of the judgment (implication direction) you use and whether you check unsatisfiability of the negation or validity of the implication directly.

7.4 Static vs Dynamic Enforcement

Static/dynamic split. *Describe the hybrid model: if the SMT solver can prove the obligation, no runtime cost; otherwise, generate a runtime* `assert` *and treat it as part of the program's explicit behavior. Connect this back to the design goals in Section 2 (zero-overhead when possible).*

7.5 Soundness Assumptions

Meta-level assumptions. *List meta-assumptions:*

- *The SMT solver is sound for the chosen logic.*
- *The encoding from Ora's fragment to SMT-LIB is faithful.*
- *Comptime evaluation used in simplification is semantics-preserving.*

These will be referenced later in any formal soundness statement.

8 Locking Discipline and Reentrancy

Ora provides an explicit, transaction-scoped write-locking mechanism for storage locations. At the surface level, programmers use annotations such as `@lock(x)` and `@unlock(x)` to mark regions where a storage-backed variable `x` must not be written again. These annotations desugar to core expressions `lock` s and `unlock` s, where s is the storage slot (or set of slots) associated with `x` in the storage layout Σ.

Only slots that are explicitly locked participate in this discipline; other slots can be written freely and do not enjoy the same guarantees.

Each EVM transaction is associated with a transaction-scoped lockset Λ_{tx}. All Ora call frames (including reentrant calls) within the same transaction share the same Λ_{tx}:

- Executing `lock` s adds s to Λ_{tx}; attempting to `lock` s again while $s \in \Lambda_{\text{tx}}$ is an error.
- Executing `unlock` s removes s from Λ_{tx} (if present).
- Executing `store` s e is only permitted when $s \notin \Lambda_{\text{tx}}$; otherwise it is an error.

The intended usage pattern is:

1. Perform any necessary writes to `x`.
2. Call `@lock(x)` to freeze further writes to `x` for the rest of the transaction (or until `@unlock(x)`).
3. Optionally perform external calls; any reentrant Ora code will see s in Λ_{tx} and will be unable to perform further `store` operations to `x`.

This discipline enforces a single-write-after-lock policy for selected variables within a transaction, which is sufficient to rule out a large class of multi-write reentrancy patterns on those variables, provided that locks are placed after the final intended update.

[Open question: Decide later whether certain slots are declared as "must-be-locked-before-write" in Σ, and enforce this in the typing rules.]

Locking structure. *Future revisions can:*

- *Add illustrative examples showing how* `@lock(x)` / `@unlock(x)` *prevent multi-write reentrancy patterns on balances or other critical variables.*
- *Clarify at the typing level which stores are required to be protected by locks (e.g., for slots marked as "lock-protected" in the storage layout) and which slots are intentionally left unlocked.*

9 Lowering to SIR and EVM

Preservation of guarantees. *Describe, at a high level, how typed Ora programs are lowered to SIR and then to EVM bytecode while preserving the type, region, and effect guarantees. A full formalization of SIR and the lowering pipeline is future work and is not needed in v0.1.*