DESIGN OVERVIEW

Clarity on Wasm

The chain ABI, the execution model, the language, and the decisions behind them

Brice Dobry

```
;; source: Clarity
(define-data-var counter uint u0)

(define-public (set-counter (n uint))
  (begin
    (var-set counter n)
    (ok true)))
```

↓ compiled off chain, deterministically

```
;; module: Wasm
(import "stacks-abi-v1" "kv_set")
(func (export "set-counter"))
abi.interface · abi.types
```

WHY WASM

Run contracts on a standard, not on our own VM

A proven engine: wasmi

Parity's Wasm interpreter, in Rust. It runs smart contracts on Stellar (Soroban), Polkadot (ink!), and soon XRPL

A standard

A W3C spec with formal semantics and a shared test suite. The chain ABI builds on it, not on stacks-core code

Metering built in

The engine counts fuel per operator. We set prices; we don't instrument code

Fast, and no JIT

A deterministic interpreter with no native code generation to attack. Room for significant performance gains over today's interpreter

More than one language

Rust or anything else can target the same ABI. Clarity is the first consumer, not the only one

Language off the chain

A new Clarity version is a compiler release. Today it's a hard fork

What it costs: a compiler in the trust path, and wasmi's fuel accounting becomes consensus, so a wasmi upgrade can be a consensus change.

01-why-clarity.md §1; 03-architecture.md §2

WHY CLARITY ON A WASM CHAIN

The chain sets the floor. Clarity raises it.

EVERY MODULE · WASM AND THE HOST

What a contract can’t do

- Touch anything outside its own memory and the host interface
- Run past its fuel budget
- Re-enter itself, or write inside a read-only call
- Keep changes from a call that returned `err`
- Move assets past post-conditions and allowances

CLARITY MODULES · ON TOP

What you can know before it runs

- It terminates: no recursion, no unbounded loops
- Every value is bounded by its type
- Its worst-case cost, as a function of its inputs
- Every path its assets can move: analysis is complete
- Proofs of invariants like supply conservation, access control, “no path locks funds”

Why that matters on a chain: contracts hold real money, can’t be patched once deployed, and are called by strangers. Knowing what code will do before you deploy it, call it, or sign for it is worth more there than anywhere else.

01-why-clarity.md §1; 03-architecture.md §5

THE SHAPE

Three parts, and only the first knows Clarity

OFF CHAIN · CLARITY-SPECIFIC

Clarity toolchain

parse → type-check → expand macros and instantiate generics → lower → link the prelude → Wasm module

Deterministic, with a provenance section

→

SPEC · EVERY LANGUAGE

Chain ABI

Runtime layer: module format, host functions, frames, fuel

Interpretation layer: declared types, canonical encoding

→

ON CHAIN · LANGUAGE-AGNOSTIC

Chain runtime

Validate at deploy, store module bytes, execute with fuel, enforce the runtime guarantees, dispatch calls

> The chain never sees Clarity source. A Clarity contract has no power on chain that a Rust module lacks.

01 · CHAIN ABI

The ABI is a narrow, language-agnostic contract

What the ABI defines

- **Module format:** a narrow Wasm profile, exports, custom sections
- **65 host functions** in 9 groups: contract boundary, context, block data, storage, assets, frames, calls, events, crypto
- **Frames:** rollback, read-only, reentrancy, allowances
- **Fuel:** charges, limits, and outcomes

What crosses the boundary

`principals` · `u128` · `raw bytes`

Nothing with more structure. Arguments, results, storage, and event data are bytes the host passes through or stores.

> **No Clarity types, no traits, no Clarity code on chain.** Today the interface between a contract and the chain is the Clarity VM itself. Here it's a versioned spec (`stacks-abi-v1`), frozen once released, so a deployed module's semantics never change under it.

07-chain-abi.md §2, §6, §11

A contract is one Wasm module

```
(module
  (import "stacks-abi-v1" "kv_get" ...)  ;; host only
  (memory (export "memory") ...)
  (func (export "init"))          ;; deploy-time
  (func (export "transfer"))      ;; every export: [] → []
  (func (export "get-balance"))

  abi.interface    ;; the chain reads this
  abi.types        ;; tools read this
  abi.provenance   ;; compiler, hashes
)
```

`abi.interface` · the chain reads it

Function names and kinds, token and event names. Required. Sorted, so it has one hash. No types, no traits.

`abi.types` · tools read it

Typed signatures, traits, storage layout, event data types. Compilers type-check against it; wallets clear-sign with it.

`abi.provenance` · verifiers read it

Compiler and version, a source hash, and the hashes of each callee’s sections the build read. Enough to rebuild and compare.

What the chain enforces, what tools agree on

Value	Runtime layer · the chain	Interpretation layer · compilers, wallets, explorers
Interface	Function names and kinds, token and event names	Typed signatures, traits
Arguments	Bytes, passed through unchanged	Declared types, canonical encoding
Results	Bytes plus an ok/err flag that closes the frame	Types of each `response` arm
Storage	Bytes under byte keys	Declared layout per variable and map
Events	Bytes, tagged with contract and event name	Each event's declared data type
NFT IDs	Compared byte for byte	Declared identifier type

The interpretation layer's encoding is SIP-005, byte for byte. What changes is decoding: canonical form only, always against a type, no sanitizing.

A module that ignores the interpretation layer still deploys and runs. Wallets just can't clear-sign it, and explorers show hex.

A call, end to end

1 Wallet encodes

Args encoded from the callee's `abi.types`, then clear-signed

2 Tx carries bytes

One opaque byte string, plus `fuel-limit` and `fuel-rate`

3 Host opens a frame

Public frame, fresh instance, runs the export

4 Module validates

`args_read`, then decode against its own declared types

5 Module sets result

`result_set(ok, bytes)` ends the export, so a result can't be set twice

6 Host closes the frame

ok commits, err discards; post-conditions checked over the asset map

```
call(target, fn, args, read_only, out, ok) → len    ;; caller validates the result
```

The chain checks no types

What we gain

- No language’s type checker in the host or the mempool
- One ABI for every language, and no Clarity on chain
- A Clarity contract can trust its own types, whoever calls it
- The chain has no traits; they’re a compile-time concept

What it costs

- Every entry point decodes and validates in Wasm, paid in fuel
- Malformed args get included, fail in the module, and pay
- A trait call is checked only for “exists” and, if asked, “read-only”
- Legacy ↔ Wasm calls: the one place the chain must interpret values

> **Today:** the VM type-checks every argument and result against the signature, for every contract.

> **Now:** each module checks what it receives. The callee checks its arguments; a Clarity caller checks the result.

07-chain-abi.md §5.4, §12 item 7

Who guarantees what

Guarantee	Enforced by	Holds for
Execution stops within the fuel budget	Chain	Every module
No reentrancy; contract-call depth ≤ 64	Chain	Every module
No writes inside a read-only call	Chain	Every module
A public call that returns `err` rolls back	Chain	Every module
Post-conditions, `as-contract?` allowances	Chain (host code)	Every module
Termination; worst-case cost computable	Clarity language	Clarity modules
No undefined behavior; responses handled	Clarity compiler	Clarity modules
Received values match declared types	Clarity compiler	Clarity modules

03-architecture.md §5; 07-chain-abi.md §7

Cost has one dimension: fuel

LEGACY: FIVE DIMENSIONS

~~runtime~~

~~read_count~~

~~read_length~~

~~write_count~~

~~write_length~~

NOW: ONE

fuel

A transaction states two fields

`fuel-limit`: the most fuel it may consume
`fuel-rate`: uSTX per million fuel

`max fee = fuel-limit × fuel-rate / 10⁶`

- You pay for fuel used, rounded up to a whole uSTX
- Exceeding the limit is `budget-exceeded`, and you pay the whole limit
- Miners order by rate and pack against one block fuel limit
- Read-only queries are free; each node sets its own fuel limit for them
- For Clarity, the static analyzer can supply `fuel-limit` without simulating

What gets charged, and when

Charge	What it covers
Intrinsic	Tx bytes and signature check; for a deploy, module bytes, validation, compilation. Charged before anything runs
Execution	Metered by the Wasm engine per operator; memory growth per page
Host functions	A base charge plus per byte in and out
Storage	Per access and per byte. Writes are priced well above compute: that’s the only brake on state growth
Instantiation	Per byte of the callee module on every call (replaces `LoadContract`)

> **Every charge is a worst-case price.** A node can be faster in the common case, but the charge can’t depend on that.

> **The engine’s fuel accounting is consensus.** An engine upgrade that changes fuel is a consensus change.

Closing the gaps where nodes could disagree

No floats

Rejected at deploy, so there are no NaN bit-pattern questions

Engine settings pinned

wasmi's fuel costs and limits are fixed by the ABI, not left to each node's config

Stack in our own units

Computed per function at deploy, so a deep call fails the same way on every node and engine version

Memory caps

16 MiB per instance and 64 MiB per tx. Otherwise 64 × 16 MiB = 1 GiB

Fresh instance per call

No globals or linear memory survive between calls

Every failure is a receipt

No "unmineable" outcome, and a failed deploy never reports `(ok true)`

Outcomes: ok · err · aborted · post-condition failure · failed before execution are all **included, and the fee is paid**. Only an invalid tx (one that can't be charged) or one that doesn't fit the block is left out.

A max fee you can prove

`verified source`
loop and size bounds

`the module`
fuel per Wasm operator

`host-charge table`
fuel schedule version N

→

Static cost analyzer

→

Worst-case fuel for each public function

A function of argument sizes, not just types: a 1-element list isn't priced like 1,000

→ the wallet proposes it as `fuel-limit`; miners pack against it

> **Bounds stop at foreign code.** A trait call, or a call into a non-Clarity module, is bounded only by the remaining budget, and the analyzer reports that. **Open question:** a fuel cap passed into `call`.

A small core, and a library written in Clarity

~12 expression forms, plus ~12 `define-*` forms

~70 primitives

~35 library definitions, in plain Clarity

```
;; spec/std/prelude.clar
(define-macro (if (condition bool) (then T) (else T))
  (match condition
    (true then)
    (false else)))

(define-macro (unwrap! (value (optional T)) (thrown E))
  (match value
    ((some inner) inner)
    (none (return thrown))))
```

Core only if: host access · not expressible · a cryptographic standard · part of a type's base operators. **Cost is not a test.**

spec/README.md; spec/std/prelude.clar

What we added

Generic functions

`(xs (list N T))`

Type and size variables in private functions and macros. Each call site gets its own instance, so every bound stays static

Overloads

`(unwrap! opt-or-resp e)`

One name, several definitions, chosen by arity and type. Exactly one must match the call

Macros

`(define-macro (if …))`

Typed, hygienic, call by name. `if` and `unwrap!` are library macros, not special forms

Rest parameters

`(+ a b c)` · `(+ a xs...)`

`+` takes many arguments because it has a rest parameter, and any user function can have one

Typed events

`(define-event transfer {…})`

`emit` replaces `print`. Indexers select by name and decode by the declared type

Type aliases

`(define-type order {id: …})`

Name a large tuple once instead of repeating it. Also replaces `use-trait` for trait types

Reference kinds

`(m (map principal uint))`

Pass a map, var, token, event, or function to a helper. Which storage it touches stays statically known

match, return, never

`(match x (some v) … none …)`

A small core for control flow. `try!`, `asserts!`, and the early-exit family are library code

spec/02-core-language.md §2.3, §2.4.1, §2.6

Toward a type system we can prove

Work in progress

- Formalizing the spec's typing rules in Rocq, and identifying gaps and ambiguities
- Machine-checked proofs about the rules, not about one implementation
- A verified reference checker, extracted to OCaml, that the compiler is tested against

Already changed the spec

- **#20** typing rules for `never` were incomplete
- **#36** equivalent traits made `lub` ambiguous
- **#37** overload resolution wasn't stated for function references
- No `NoType`: every expression has a complete type

> **Only possible because the spec is the definition.** Today the type checker is the definition, bugs and all, so there's nothing to prove it against.

spec/02-core-language.md; issues #20, #36, #37

The compiler is now in the trust path

Legacy had no compiler between author and chain. Now there is one, and a bug there makes the deployed code differ from the source everyone audited.

1 Spec and conformance vectors

Vectors are data: input, expected outcome, rules exercised. Four kinds: language, ABI, encoding, golden build

2 A direct evaluator as oracle

Applies the spec’s rules one at a time. Differential testing and fuzzing against the compiler. It never runs on chain

3 Reproducible builds

Same inputs, byte-identical module on every platform. Provenance names the compiler, source hash, and callee section hashes

4 Analyses attach to a verified triple

(source, compiler version, module): a bound or proof counts only when the module rebuilds byte for byte

HOW WE DECIDE

Eight principles every proposal must pass

P1

Decidability is the invariant

Every program terminates; every value and loop is bounded by its type; no recursion, no re-entry

P2

Minimal core

Only what the language can’t express itself*. Everything else is standard library

P3

The library has no special powers

Ordinary Clarity, checked and metered like any contract. Speed is never a reason to add a native

P4

Cost is not a language concern

No cost table. The chain meters the module’s fuel, and bounds still follow from P1

P5

The deployed Wasm is law

The source is its specification; a module that differs from the source is a compiler bug

P6

One way to do each thing

No aliases, no overlapping forms, uniform naming, declared arity

P7

No undefined behavior, no silent failure

Overflow and bad conversions abort; errors are values or aborts; no nulls, no implicit conversions

P8

Spec first, then implementation

Every rule cites conformance vectors, and none is frozen without them

Questions?

stx-labs.github.io/clarity-vision