elrond

The Internet-Scale Blockchain

A highly scalable, fast and secure blockchain platform for distributed apps, enterprise use cases and the new internet economy.

Pushing Rust to the limit in a Blockchain Environment

🦀

Who I am

- Engineer at Elrond for almost 4 years, rustacean for 3
- Passionate about languages, frameworks and models
- Favorite emoji: 🪓 (the axe)
- Also, 🦀❤️

This Presentation

1. Crash course in Elrond Architecture
2. How to build a Smart Contract framework in 300+ easy steps (abridged)
3. How to push Rust to the breaking point (almost)

Crash Course in Elrond Architecture

What is Elrond?

- A super fast & cheap **Layer 1 blockchain**
- Sharding, fast **smart contracts**, great **dApps** (Maiar, Maiar DEX, etc.)
- Innovative eGold **tokenomics**
- A growing **ecosystem** of developers & users

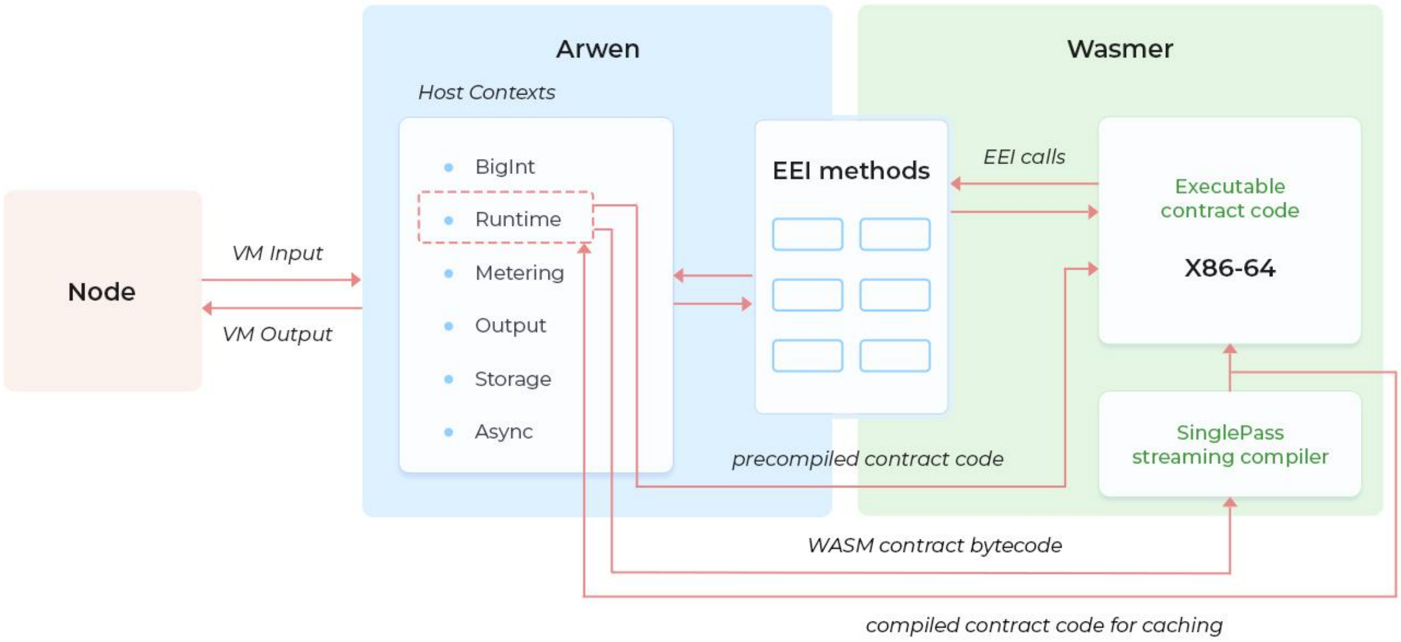
What about Smart Contracts?

The **fastest Blockchain** is useless without the **fastest VM** …

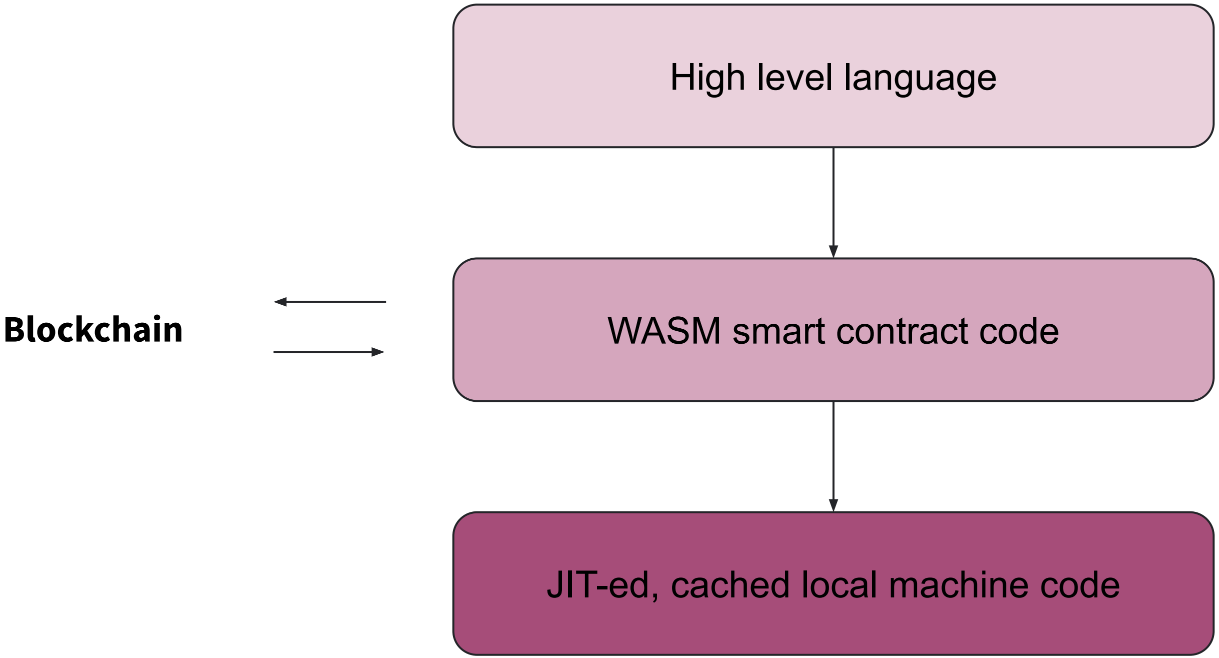
… which is useless without the **fastest smart contract code.**

Portable code, near-native execution:
WebAssembly + Wasmer

VM-Wasmer integration



How it all works?



So the objectives:

- Contract size is crucial (JIT & blockchain storage are expensive)
- Speed (obviously)
- Devs shouldn’t worry about a lot of things (and cannot be trusted)

Only one modern language cuts it …

How to build a Smart Contract framework in 300+ easy steps (abridged)

Make it fast!

No time for:

- ➢ big number arithmetic,
- ➢ crypto function implementation,
- ➢ memory allocation 🤕

SC	🧠
VM	💪

```
(func $add_big_int (type 13) ;; the smarts 🧠
  call $checkNoPayment
  i32.const 2
  call $check_num_arguments
  i32.const 0
  call $get_argument_big_int ;; conveniently, by the VM 💪
  i32.const 1
  call $get_argument_big_int ;; conveniently, by the VM 💪
  call $add_two_big_ints     ;; the main muscle 💪
  call $return_big_int       ;; conveniently, by the VM 💪
)
```

Make it pretty!

```rust
#[endpoint]
fn add_big_int(&self, a: BigInt, b: BigInt) -> BigInt {
    a + b
}
```

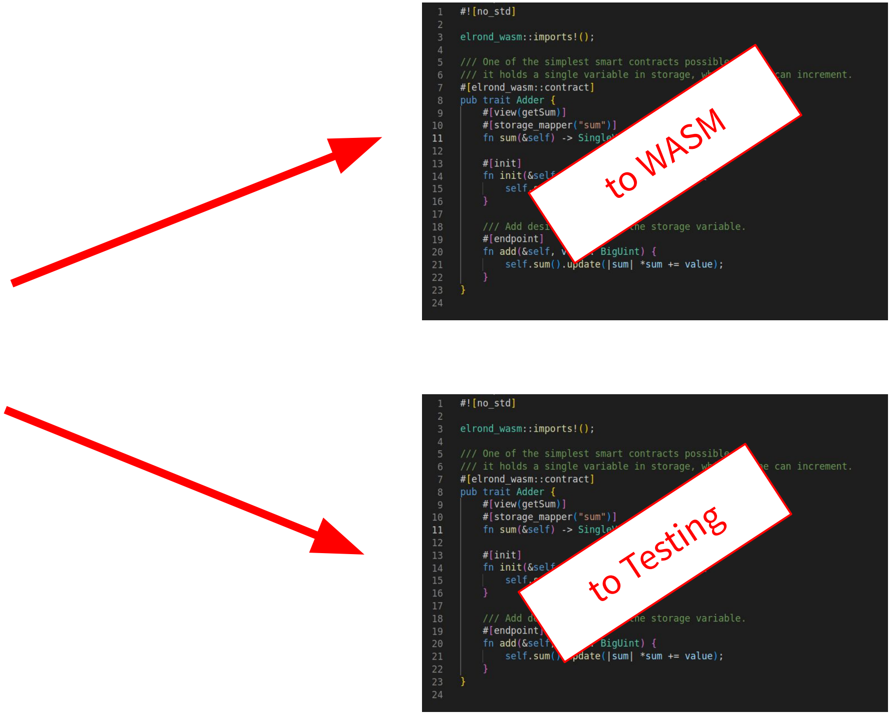
```wasm
(func $add_big_int (type 13) ;; the smarts 🧠
  call $checkNoPayment
  i32.const 2
  call $check_num_arguments
  i32.const 0
  call $get_argument_big_int ;; conveniently, by the VM 💪
  i32.const 1
  call $get_argument_big_int ;; conveniently, by the VM 💪
  call $add_two_big_ints     ;; the main muscle 💪
  call $return_big_int       ;; conveniently, by the VM 💪
)
```

Make it pretty!

```rust
#![no_std]

elrond_wasm::imports!();

/// One of the simplest smart contracts possible,
/// it holds a single variable in storage, which anyone can increment.
#[elrond_wasm::contract]
pub trait Adder {
    #[view(getSum)]
    #[storage_mapper("sum")]
    fn sum(&self) -> SingleValueMapper<BigUint>;

    #[init]
    fn init(&self, initial_value: BigUint) {
        self.sum().set(initial_value);
    }

    /// Add desired amount to the storage variable.
    #[endpoint]
    fn add(&self, value: BigUint) {
        self.sum().update(|sum| *sum += value);
    }
}

```

Make it testable!

```rust
#![no_std]

elrond_wasm::imports!();

/// One of the simplest smart contracts possible,
/// it holds a single variable in storage, which anyone can increment.
#[elrond_wasm::contract]
pub trait Adder {
    #[view(getSum)]
    #[storage_mapper("sum")]
    fn sum(&self) -> SingleValueMapper<BigUint>;

    #[init]
    fn init(&self, initial_value: BigUint) {
        self.sum().set(initial_value);
    }

    /// Add desired amount to the storage variable.
    #[endpoint]
    fn add(&self, value: BigUint) {
        self.sum().update(|sum| *sum += value);
    }
}
```



Make it interoperable!

```rust
fn call_add(&self) {
    self.adder_proxy()
        .contract(self.adder_address())
        .add(BigUint::from(5))
        .async_call()
        .call_and_exit()
}
```

MAGIC!

```rust
#![no_std]

elrond_wasm::imports!();

/// One of the simplest smart contracts possible,
/// it holds a single variable in storage, which anyone can increment.
#[elrond_wasm::contract]
pub trait Adder {
    #[view(getSum)]
    #[storage_mapper("sum")]
    fn sum(&self) -> SingleValueMapper<BigUint>;

    #[init]
    fn init(&self, initial_value: BigUint) {
        self.sum().set(initial_value);
    }

    /// Add desired amount to the storage variable.
    #[endpoint]
    fn add(&self, value: BigUint) {
        self.sum().update(|sum| *sum += value);
    }
}
```

Make it interoperable!

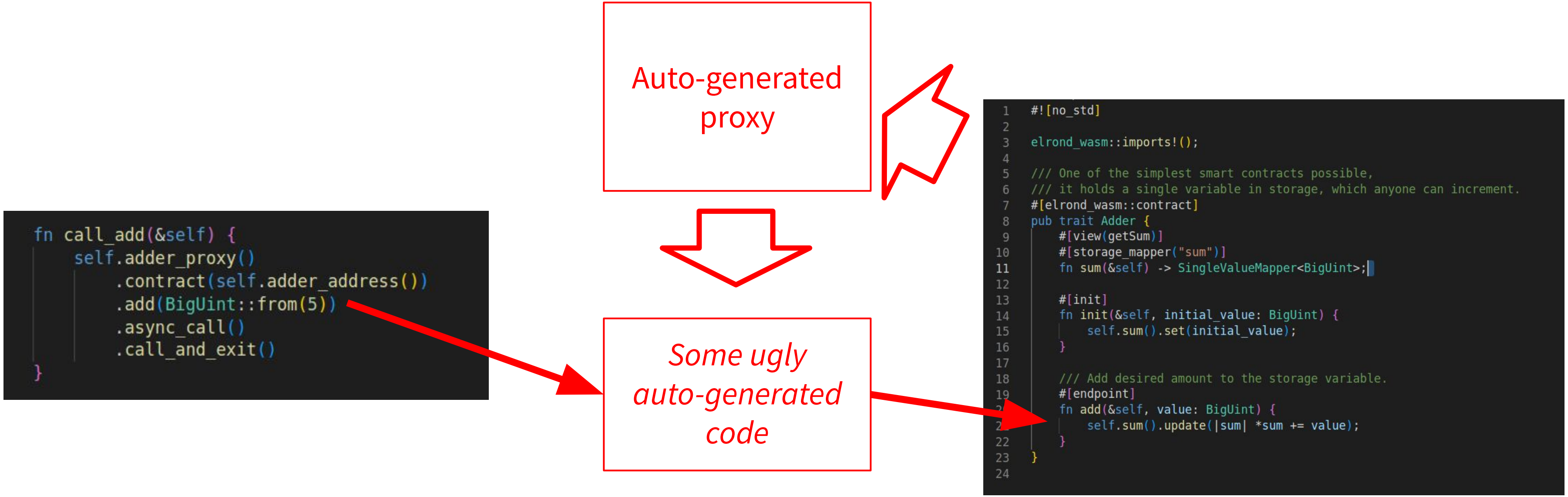
```rust
#[elrond_wasm::proxy]
pub trait AdderProxy {
    #[init]
    fn init(&self, initial_value: BigUint);

    #[endpoint]
    fn add(&self, value: BigUint);
}
```

```rust
fn call_add(&self) {
    self.adder_proxy()
        .contract(self.adder_address())
        .add(BigUint::from(5))
        .async_call()
        .call_and_exit()
}
```

Some ugly auto-generated code

```rust
#![no_std]

elrond_wasm::imports!();

/// One of the simplest smart contracts possible,
/// it holds a single variable in storage, which anyone can increment.
#[elrond_wasm::contract]
pub trait Adder {
    #[view(getSum)]
    #[storage_mapper("sum")]
    fn sum(&self) -> SingleValueMapper<BigUint>;

    #[init]
    fn init(&self, initial_value: BigUint) {
        self.sum().set(initial_value);
    }

    /// Add desired amount to the storage variable.
    #[endpoint]
    fn add(&self, value: BigUint) {
        self.sum().update(|sum| *sum += value);
    }
}
```

Make it interoperable!



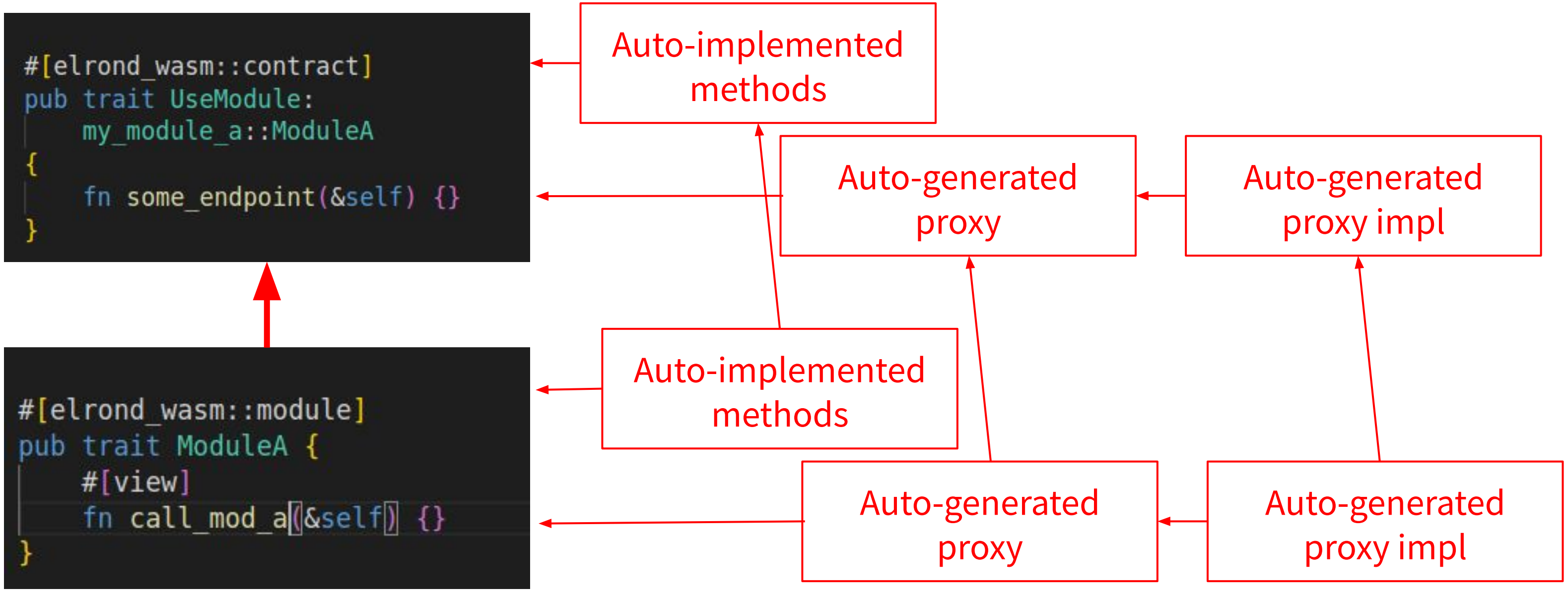
Make it composable!

```
#[elrond_wasm::contract]
pub trait UseModule:
    my_module_a::ModuleA
{
    fn some_endpoint(&self) {}
}
```

```
#[elrond_wasm::module]
pub trait ModuleA {
    #[view]
    fn call_mod_a(&self) {}
}
```

Seems easy, until you realize ….

Make it composable!

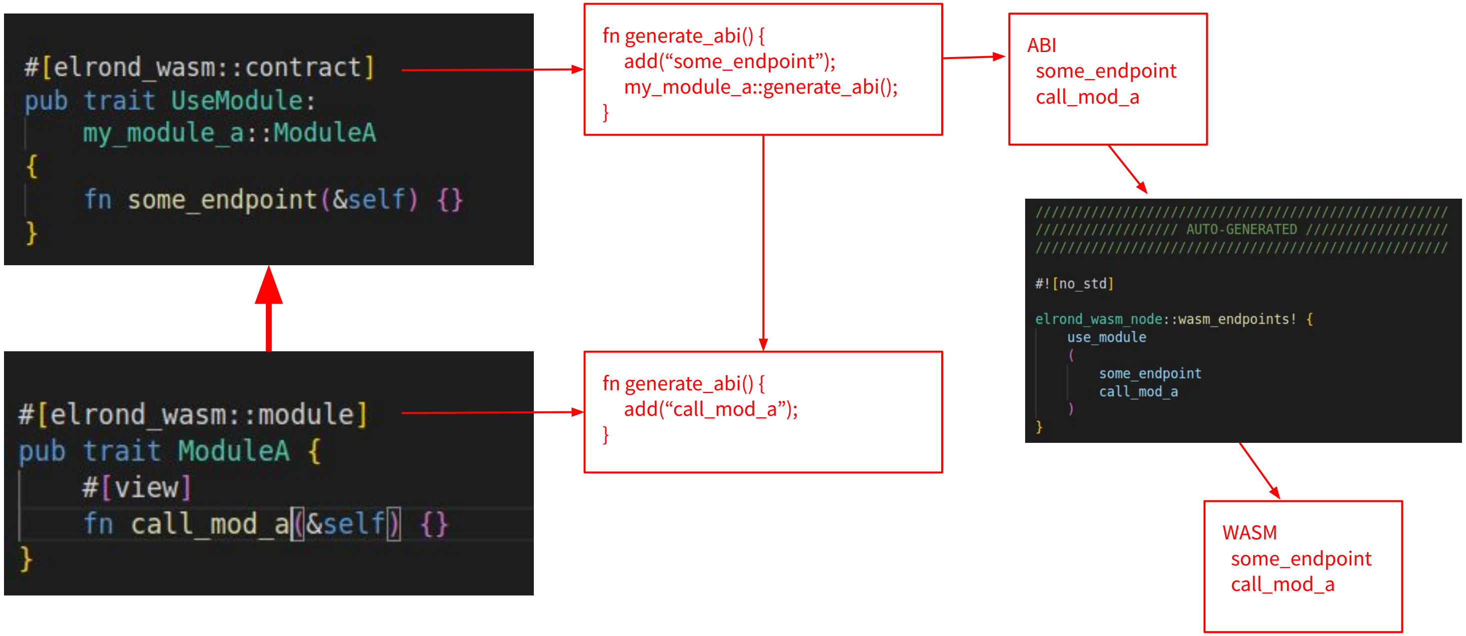


Make it composable!

How to avoid **stray endpoints**:

1. Prepare oneself spiritually for an onslaught of meta-programming 🧘‍♂️
2. ~~Generate the ABI~~ (too soon!)
3. Via macros, generate code that generates an ABI (for each module)
4. Make a *meta* crate that will handle meta-things
5. Call the ABI generator (for the entire contract)
6. Generate a *wasm* crate based on the ABI
7. Build the *wasm* crate to produce a .wasm file
8. Move the .wasm file somewhere nice
9. Sigh in relief 🧘‍♂️

Make it composable!



Make it composable!

```rust
#[elrond_wasm::contract]
pub trait UseModule:
    my_module_a::ModuleA
{
    fn some_endpoint(&self) {}
}
```

```
fn generate_abi() {
    add("some_endpoint");

}
```

ABI
some_endpoint

```rust
////////////////////////////////////////////////////
////////////////// AUTO-GENERATED //////////////////
////////////////////////////////////////////////////

#![no_std]

elrond_wasm_node::wasm_endpoints! {
    use_module
    (
        some_endpoint
    )
}
```

WASM
some_endpoint

```rust
/// Just chilling in the project.
#[elrond_wasm::module]
pub trait DontIncludeMe {
    #[view]
    fn definitely_nope(&self) {}

    #[endpoint]
    fn probably_hack(&self) {}
}
```

Make it escape!

Why not use all the magic off-chain too?

```rust
async fn vm_query_sum() -> usize {
    interactor.vm_query(adder.sum()).await
}

async fn tx_add() {
    interactor
        .sc_call(
            adder
                .add(5u32) // magic conversion u32 -> BigUint
                .into_blockchain_call()
                .from(my_wallet_address)
                .gas_limit("5,000,000")
                .into(),
        )
        .await;
}
```

So to sum up …

… the 300+ easy steps to building a framework can be grouped into:

Make it fast!

Make it pretty!

Make it testable!

Make it interoperable!

Make it composable!

Make it escape!

(not necessarily in that order)

How to push Rust to the breaking point (almost)

Exhibit A: Fat Result<T, E>

How a sane person writes a deserializer trait:

```rust
pub trait Decode {
    fn decode<I>(input: I) -> Result<Self, DecodeError>
    where
        I: TopDecodeInput,
    {
        // ...
    }
}
```

… turns out Result handling inflates bytecode size quite a bit

Exhibit A: Fat Result<T, E>

How we do:

```
pub trait Decode {
    fn decode_or_handle_err<I, H>(input: I, h: H) -> Result<Self, H::HandledErr>
    where
        I: TopDecodeInput,
        H: DecodeErrorHandler,
    {
        // ...
    }
}
```

- In contracts we use a “panicking” error handler, with error type ! (never)
- Result<Self, !> **is compiled as** Self

Exhibit B: Vararg Madness

- Requirements:
  - Auto-generate an argument loader for each endpoint
  - Allow variadic args
  - If there are no varargs, output code like: *get_arg(0); get_arg(1); …*
  - If there are varargs, output code like: *while more_args() { get_arg(i); i+= 1 }*
  - The compiler should decide by arg type alone (macros have no idea)
  - … so looking like heavy generics ahead

Exhibit B: Vararg Madness

- Solution:
  - Compile-time functional-style “fold”, or whatever this is:

```
// from this ...
#[endpoint]
fn add(&self, value1: u32, value2: u32, varargs: MyVarArgsImpl<u32>) {
    // ...
}

// ... we generate:
fn call_add(&self) {
    let (value1, (value2, (varargs, ()))) = load_args::<(u32, (u32, (MyVarArgsImpl<u32>, ())))>(
        ("value1", ("value2", ("varargs", ()))),
    ));
    self.add(value1, value2, varargs);
}
```

  - Using the magic of generics and monomorphization, all *if*s in load_args can peek into the future and are resolved at compile time!
  - Yes, you can make static lists by nesting tuples forever!

Exhibit C: Owning stuff that isn’t there

Managed types™:

the VM owns the data, types are glorified handles, but we still need to play the Rust ownership game!

What the dev sees	What’s in the bytecode	Who owns the data
let x: BigInt;	i32	x
let y: &BigInt = &x;	&i32	x
let z = y.clone();	i32 (+ a clone instruction)	z
let v: ManagedVec<BigInt> = …	i32	v
let item: ??? = v.get(i);	*cannot be &i32 … nothing to point to*	v
let item_ref: &BigInt = item.deref();	*&i32 ???*	v

Exhibit C: Owning stuff that isn't there

Managed types™:

the VM owns the data, types are glorified handles, but we still need to play the Rust ownership game!

What the dev sees	What's in the bytecode	Who owns the data
let x: BigInt;	i32	x
let y: &BigInt = &x;	&i32	x
let z = y.clone();	i32 (+ a clone instruction)	z
let v: ManagedVec<BigInt> = …	i32	v
let item: ManagedRef<'_, BigInt> = v.get(i);	i32	v
let item_ref: &BigInt /* 😕? */ = item.deref();	&i32, and we transmute from there!	v

More information at https://docs.elrond.com/

Reach out:

- andrei.marinica@elrond.com
- https://t.me/ElrondDevelopers
- https://t.me/ElrondNetwork

Follow on:

- https://twitter.com/andreimmarinica
- https://twitter.com/ElrondNetwork

Thank you for watching!

Hope you enjoyed the ride!