Turning Smart Contracts into Indexers with Cross-Compilation in Rust

Conf42 Rustlang 2023

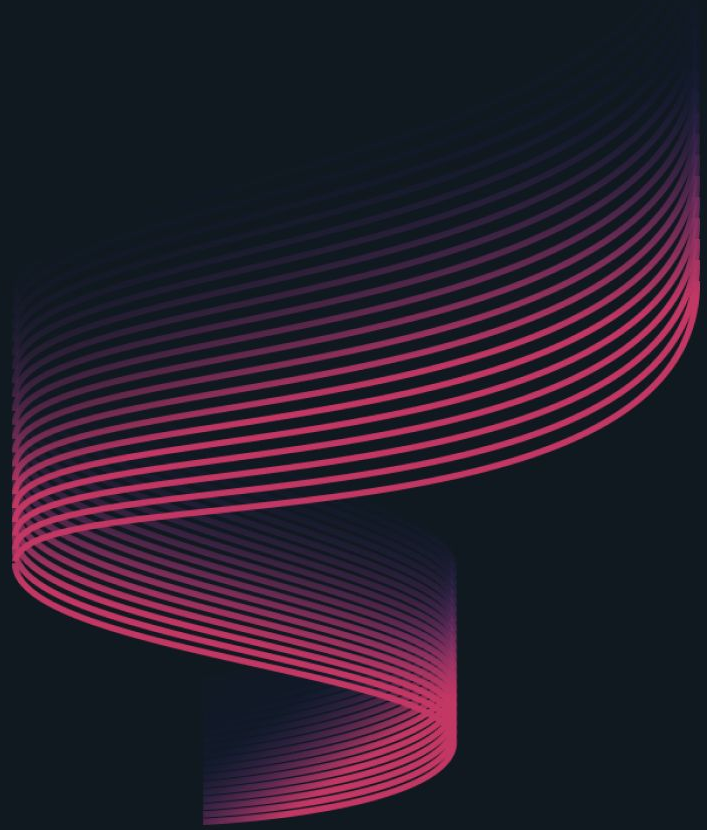
Michael Birch

Goals

- Show functional programming (FP) patterns in Rust
- Resulting code is:
  - Easier to test, maintain
  - Easier to re-use
- Case example:
  - shared codebase for a smart contract and indexer

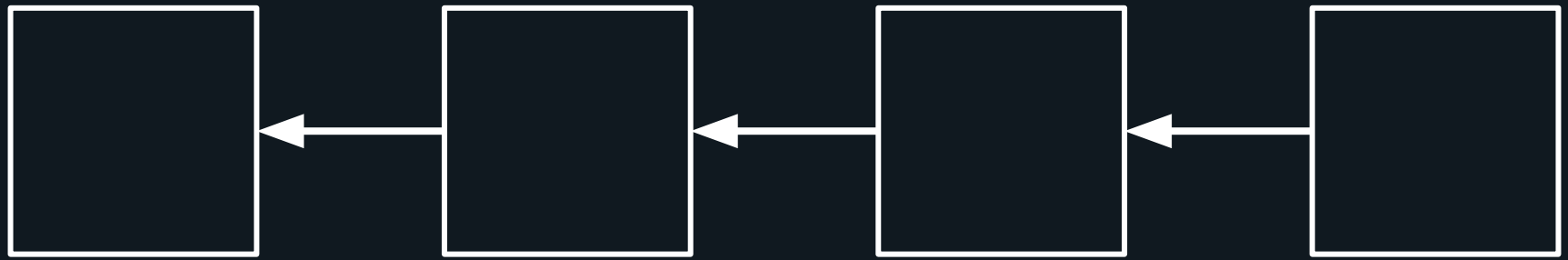
Outline

- Blockchain basics
  - Jargon quick start
  - Smart contracts
  - Indexers
- Key Rust concepts
  - Compilation targets
  - Type generics
- Case example: Aurora Engine
- Conclusion



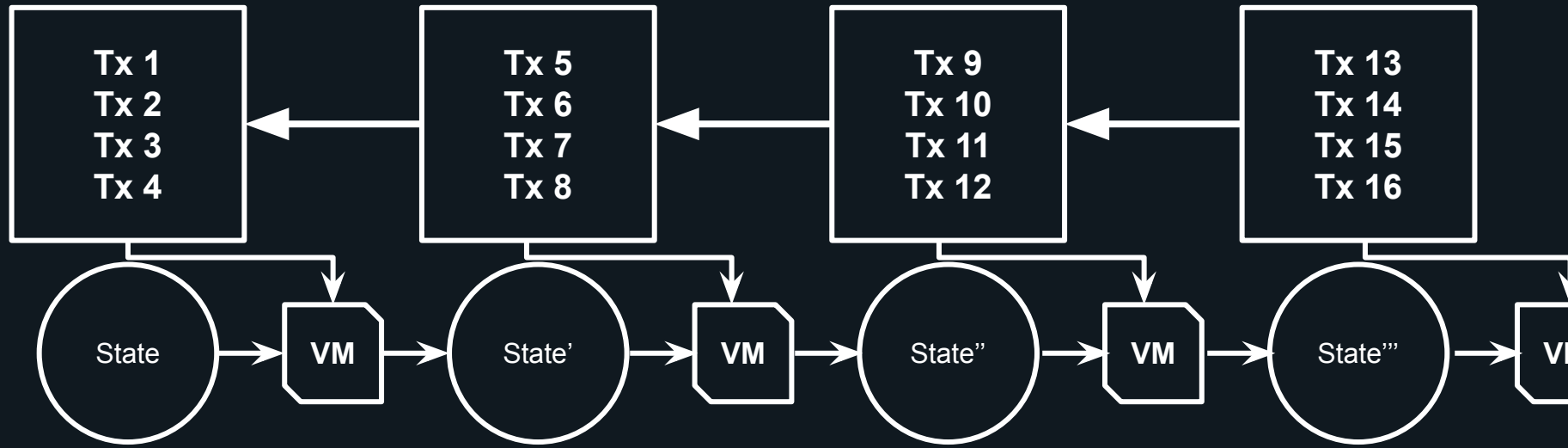
Blockchain basics: Jargon quick start

- **Blockchain**: append-only data structure with tamper-proof history
- Individual data chunks called **blocks**



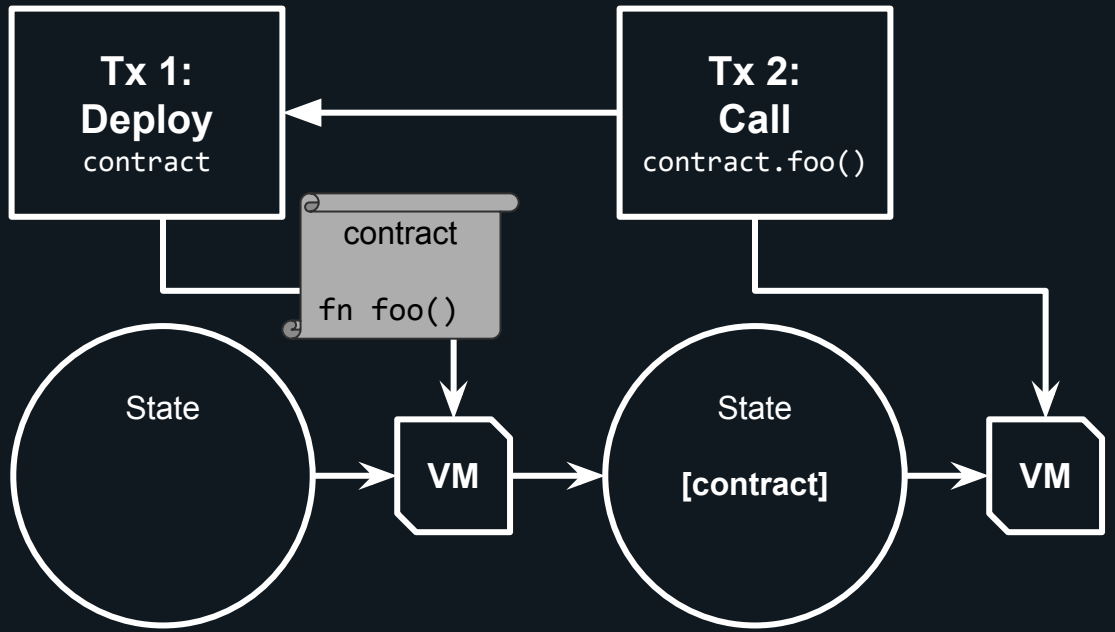
Blockchain basics: Jargon quick start

- **Transaction**: data element within a block
- Transactions are interpreted inside a **VM** to cause state transitions



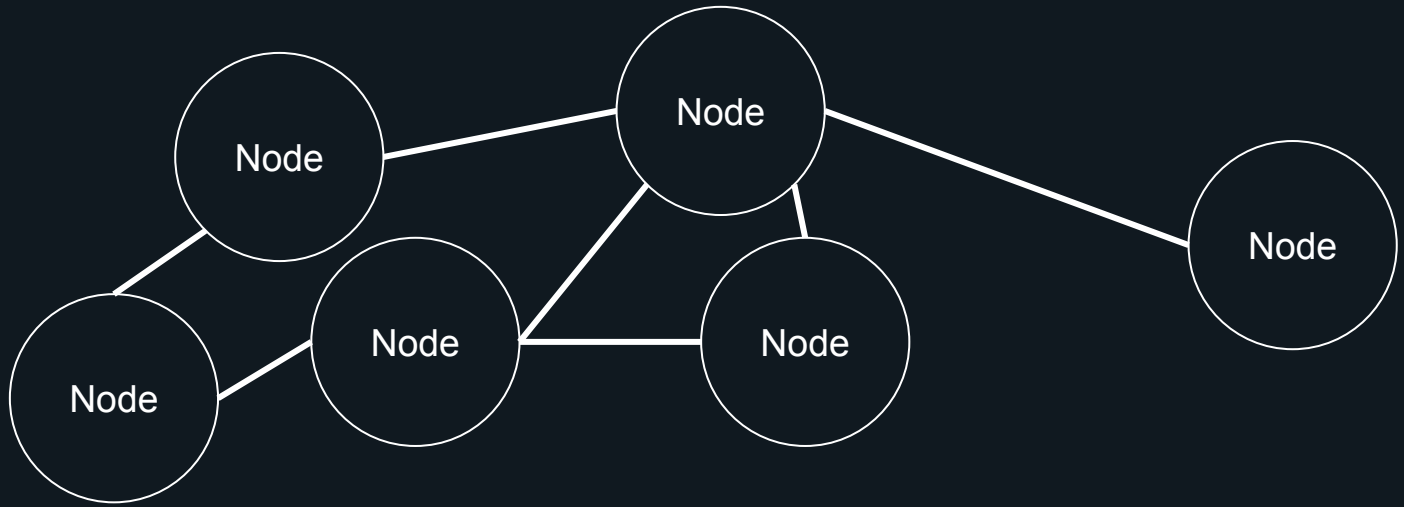
Blockchain basics: Smart contracts

- **Smart contract**: program for the VM of a blockchain platform
- Transactions may invoke a method of a smart contract



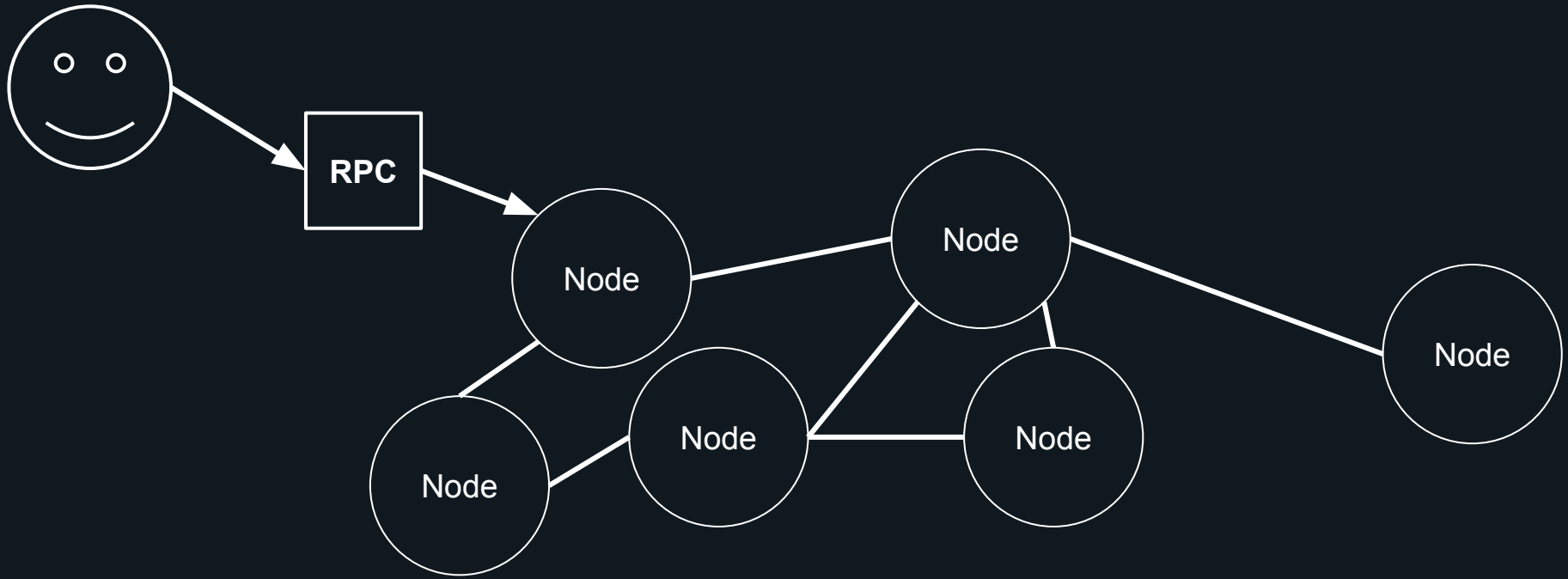
Blockchain basics (continued)

- **Blockchain platform**: a distributed blockchain continuously built by decentralized participants (**nodes**) from user-submitted transactions
- Nodes eventually agree on the blockchain via a **consensus algorithm**



Blockchain basics (continued)

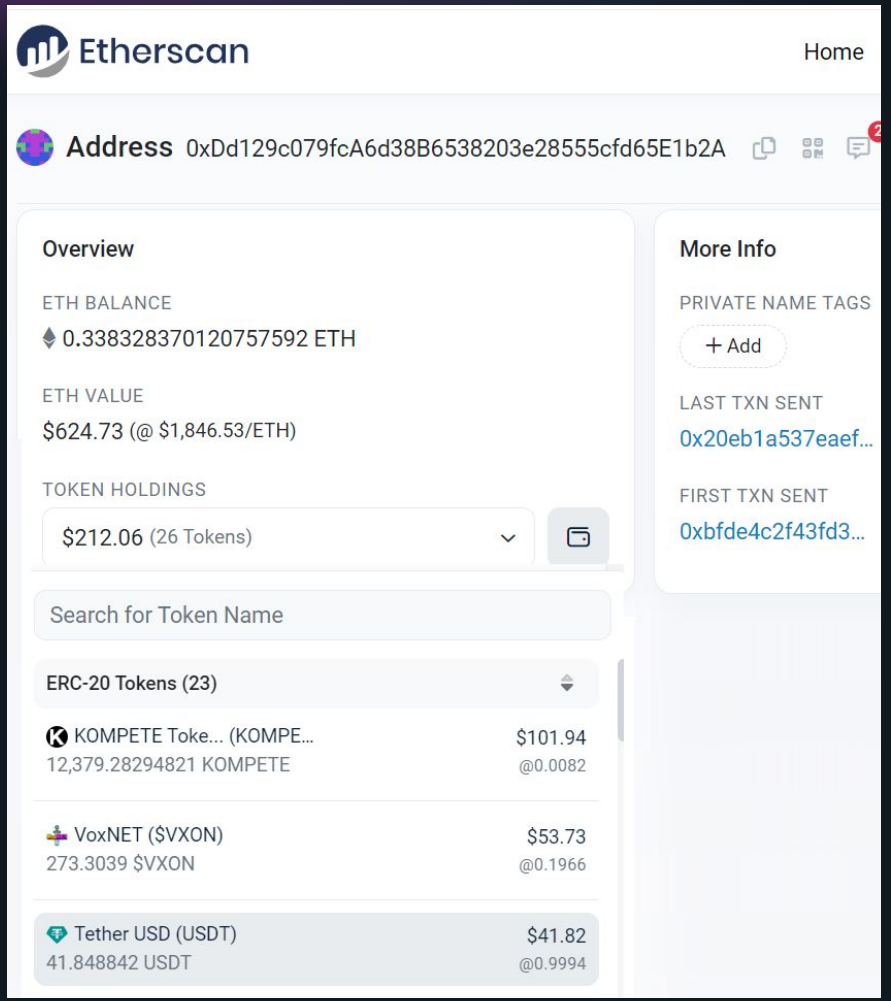
- Users interact with a blockchain platform via an **RPC**
  - Either by running their own node or using a service provider



Blockchain basics: Indexers

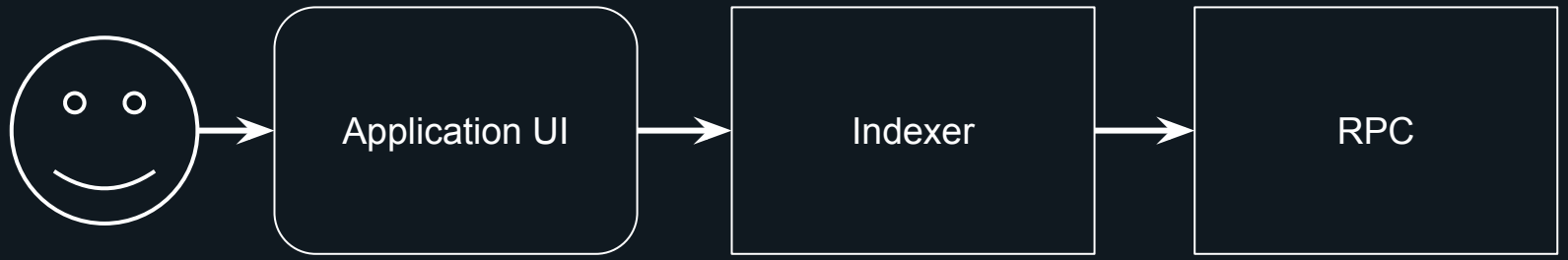
- **Indexer**: off-chain program creating a specialized view of the state
  - Addresses problem of some queries being too slow via RPC

- Example: block explorers index tokens for users



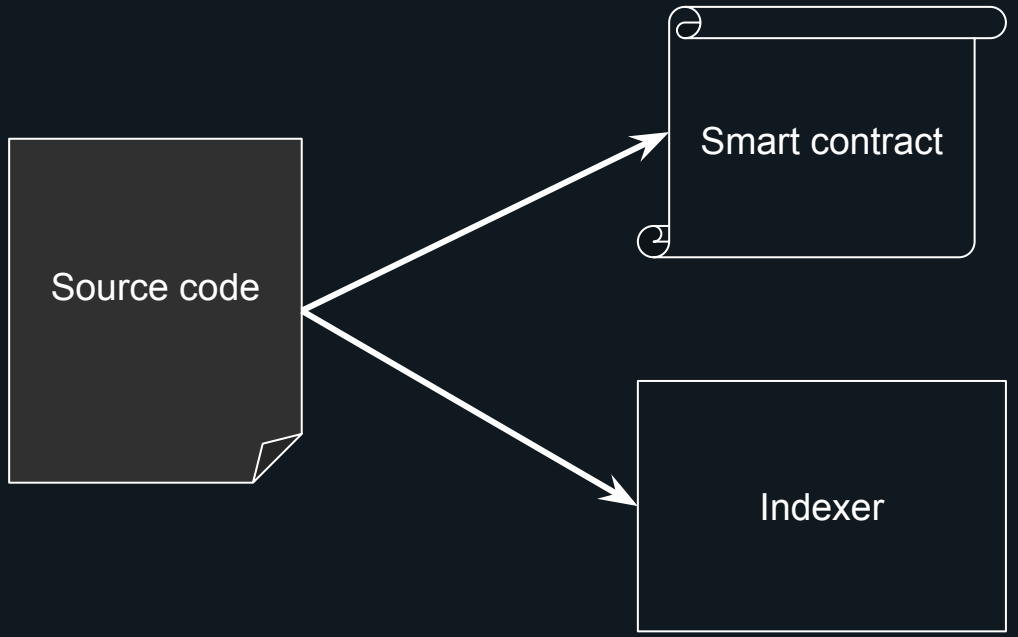
Blockchain basics: Indexers

- Indexers help create low-latency (web2-like) experiences for users



Idea: Turning smart contracts into indexers

- Same code both a smart contract and its own indexer
  - Lower maintenance
  - Uses beyond state query



Key Rust concepts: Compilation targets

- Rust allows compiling different kinds of output
  - https://rust-lang.github.io/rustup/cross-compilation.html
  - https://doc.rust-lang.org/nightly/rustc/platform-support.html

```
$ rustup target add wasm32-unknown-unknown
```

```
$ cargo build --release --target wasm32-unknown-unknown
```

Key Rust concepts: Compilation targets

- Conditional compilation can handle platform-specific logic
  - https://doc.rust-lang.org/reference/conditional-compilation.html
- Drawbacks: verbose, tedious with IDEs

```
fn foo() {
    #[cfg(target_arch = "wasm32")]
    foo_for_wasm();

    #[cfg(not(target_arch = "wasm32"))]
    foo_for_generic_arch();
}
```

Key Rust concepts: Type generics

- Write code generic over an interface using type generics and trait bounds

```
trait IO {
    fn read(&self, key: &[u8]) -> Vec<u8>;
    fn write(&mut self, key: &[u8], value: &[u8]);
}
fn get_balance<I: IO>(io: &I, user: User) -> u128 {
    u128::from_be_bytes(&io.read(&user.id()))
}
```

Key Rust concepts: Type generics

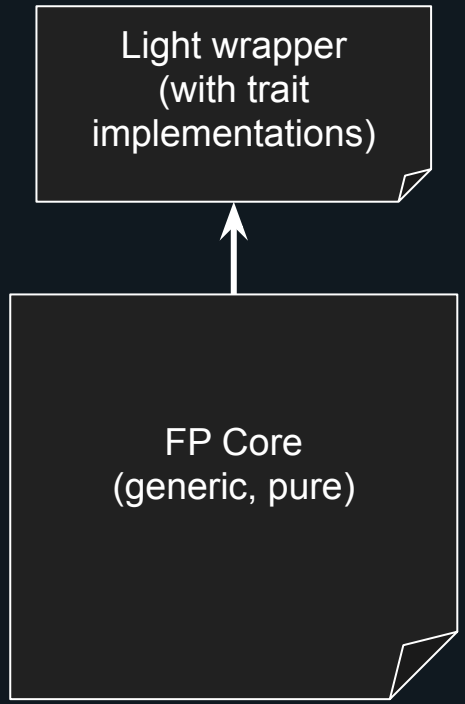
- Include an implementation for the trait in both targets
- Reuse the generic code in both smart contract and indexer

```
// indexer/src/main.rs

struct IndexerIO { ... }

impl IO for IndexerIO { ... }

fn main() {
    let io = IndexerIO::new();
    let balance = get_balance(&io, user);
    ...
}
```

```
// contract/src/lib.rs

struct WasmIO { ... }

impl IO for WasmIO { ... }

fn method_entry_point() {
    let io = WasmIO::new();
    let balance = get_balance(&io, user);
    ...
}
```

Aside: Patterns from functional programming (FP)

- Pure code does not depend on the environment
  - I.e. no target-specific effects
- Factoring out target-specific effects as generics makes code easier to test and maintain



Aside: Patterns from functional programming (FP)

- Advantages to this style of programming
  - Easier to test
    - Effects like IO can be done in-memory with test-only implementations of the traits
  - Easier to reason about
    - Effects are explicit in the type signature, no need to check to implementation details
  - Easier to re-use
    - Abstract code can be applied to more situations (like both smart contracts and indexers)

Case example: Aurora Engine

- Aurora is an Ethereum scaling solution built on the Near blockchain platform
  - https://aurora.dev/
  - https://near.org/
- Core product is an EVM deployed as a smart contract on Near
- Need an RPC compatible with Ethereum spec to integrate with Ethereum tooling (e.g. Metamask)
- Possible implementations:
  - Convert Ethereum RPC calls to Near RPC calls (slow)
  - Use the same Aurora Engine code as an indexer

Case example: Aurora Engine

- https://github.com/aurora-is-near/aurora-engine

```rust
pub fn get_balance<I: IO>(io: &I, address: &Address) -> Wei {
    let raw: U256 = io.&I
        .read_u256(key: &address_to_key(prefix: KeyPrefix::Balance, address))
        .unwrap_or_else(op: |_| U256::zero());
    Wei::new(amount: raw)
}
```

```rust
pub fn set_balance<I: IO>(io: &mut I, address: &Address, balance: &Wei) {
    io.write_storage(
        key: &address_to_key(prefix: KeyPrefix::Balance, address),
        value: &balance.to_bytes(),
    );
}
```

```rust
pub fn add_balance<I: IO>(
    io: &mut I,
    address: &Address,
    amount: Wei,
) -> Result<(), BalanceOverflow> {
    let current_balance: Wei = get_balance(io, address);
    let new_balance: Wei = current_balance.checked_add(amount).ok_or(err: BalanceOverflow)?;
    set_balance(io, address, &new_balance);
    Ok(())
}
```

Case example: Aurora Engine

```rust
#[derive(Copy, Clone, Default)]
pub struct Runtime;
```

```rust
impl crate::io::IO for Runtime {
    type StorageValue = RegisterIndex;

    fn read_storage(&self, key: &[u8]) -> Option<Self::StorageValue> {
        unsafe {
            if exports::storage_read(
                key.len() as u64,
                key.as_ptr() as u64,
                Self::READ_STORAGE_REGISTER_ID.0,
            ) == 1
            {
                Some(Self::READ_STORAGE_REGISTER_ID)
            } else {
                None
            }
        }
    }

}
```

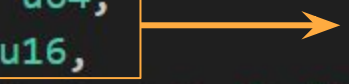
Case example: Aurora Engine

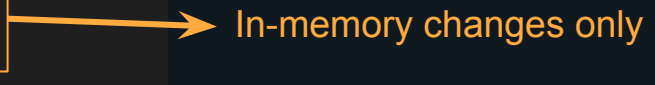
```rust
#[derive(Copy, Clone, Default)]
pub struct Runtime;
```

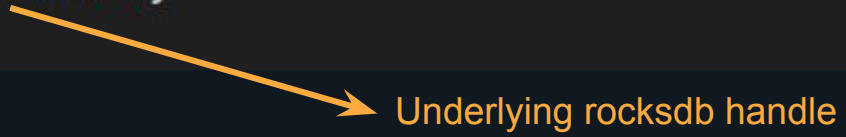
```rust
impl crate::io::IO for Runtime {
    type StorageValue = RegisterIndex;
    fn write_storage(&mut self, key: &[u8], value: &[u8]) -> Option<Self::StorageValue> {
        unsafe {
            if exports::storage_write(
                key.len() as u64,
                key.as_ptr() as u64,
                value.len() as u64,
                value.as_ptr() as u64,
                Self::WRITE_REGISTER_ID.0,
            ) == 1
            {
                Some(Self::WRITE_REGISTER_ID)
            } else {
                None
            }
        }
    }
}
```

Case example: Aurora Engine

```rust
#[derive(Copy, Clone)]
4 implementations
pub struct EngineStateAccess<'db, 'input, 'output> {
    input: &'input [u8],
    bound_block_height: u64,
    bound_tx_position: u16,
    transaction_diff: &'output RefCell<Diff>,
    output: &'output Cell<Vec<u8>>,
    db: &'db DB,
}
```







Case example: Aurora Engine

```rust
impl<'db, 'input: 'db, 'output: 'db> IO for EngineStateAccess<'db, 'input, 'output> {
    type StorageValue = EngineStorageValue<'db>;

    fn read_storage(&self, key: &[u8]) -> Option<Self::StorageValue> {
        if let Some(diff) = self.transaction_diff.borrow().get(key) {
            return diff
                .value()
                .map(|bytes| EngineStorageValue::Vec(bytes.to_vec()));
        }

        let opt = self.construct_engine_read(key);
        let mut iter = self.db.iterator_opt(mode: rocksdb::IteratorMode::End, readopts: opt);
        let value = iter.next().and_then(|maybe_elem| {
            maybe_elem
                .ok()
                .map(|(_, value)| DiffValue::try_from_bytes(&value).unwrap())
        })?;
        value.take_value().map(EngineStorageValue::Vec)
    }
}
```

Case example: Aurora Engine

```rust
impl<'db, 'input: 'db, 'output: 'db> IO for EngineStateAccess<'db, 'input, 'output> {
    type StorageValue = EngineStorageValue<'db>;

    fn write_storage(&mut self, key: &[u8], value: &[u8]) -> Option<Self::StorageValue> {
        let original_value = self.read_storage(key);

        self.transaction_diff
            .borrow_mut()
            .modify(key: key.to_vec(), value: value.to_vec());

        original_value
    }

}
```

Case example: Aurora Engine

- Pattern applies to all effects, not just storage!
  - Environment variables
  - Calls to other on-chain contracts

```rust
pub trait Env {
    /// Account ID that signed the transaction.
    fn signer_account_id(&self) -> AccountId;
    /// Account ID of the currently executing contract.
    fn current_account_id(&self) -> AccountId;
    /// Account ID which called the current contract.
    fn predecessor_account_id(&self) -> AccountId;
    /// Height of the current block.
    fn block_height(&self) -> u64;
    /// Timestamp (in ns) of the current block.
    fn block_timestamp(&self) -> Timestamp;
    /// Amount of NEAR attached to current call
    fn attached_deposit(&self) -> u128;
    /// Random seed generated for the current block
    fn random_seed(&self) -> H256;
    /// Prepaid NEAR Gas
    fn prepaid_gas(&self) -> NearGas;
}
```

```rust
pub trait PromiseHandler {
    fn promise_results_count(&self) -> u64;

    fn promise_result(&self, index: u64) -> Option<PromiseResult>;

    unsafe fn promise_create_call(&mut self, args: &PromiseCreateArgs) -> PromiseId;

    unsafe fn promise_attach_callback(
        &mut self,
        base: PromiseId,
        callback: &PromiseCreateArgs,
    ) -> PromiseId;

    unsafe fn promise_create_batch(&mut self, args: &PromiseBatchAction) -> PromiseId;

    fn promise_return(&mut self, promise: PromiseId);
}
```

Case example: Aurora Engine

```rust
pub fn submit<I: IO + Copy, E: Env, P: PromiseHandler>(
    io: I,
    env: &E,
    args: &SubmitArgs,
    state: EngineState,
    current_account_id: AccountId,
    relayer_address: Address,
    handler: &mut P,
) -> EngineResult<SubmitResult> {
    submit_with_alt_modexp::<_, _, _, AuroraModExp>( ···
}
```

Case example: Aurora Engine

- Advanced indexer functionality `eth_estimateGas`
  - Check how much gas an EVM transaction with take on Aurora by simulating the transaction
- https://github.com/aurora-is-near/borealis-engine-lib

```rust
pub fn estimate_gas(
    storage: &Storage,
    request: EthCallRequest,
    earliest_block_height: u64,
) -> (Result<SubmitResult, StateOrEngineError>, NonceStatus) {···
```

Set up environment variables

```rust
let env: Fixed = aurora_engine_sdk::env::Fixed {
    signer_account_id: default_account_id.clone(),
    current_account_id,
    predecessor_account_id: default_account_id,
    block_height,
    block_timestamp: block_metadata.timestamp,
    attached_deposit: 1,
    random_seed: block_metadata.random_seed,
    prepaid_gas: aurora_engine_types::types::NearGas::new(gas: 300),
};
storage &Storage
    .with_engine_access(block_height: block_height + 1, transaction_position: 0, input: &[], f: |io: EngineStateAccess<'_, '_, '_>| {···
    .result
```

Use state access in closure (elided)

Conclusion

- High level ideas:
  - Write business logic as pure code with abstract interfaces marking effectful boundaries
  - Re-use business logic in all applications which require it
- Rust specifics:
  - Use type generics with trait bounds
  - Use conditional compilation for different targets
- Near blockchain specifics:
  - Smart contracts and indexers share a codebase using Rust + Wasm tech stack
- Other possible applications:
  - Shared code between Web and Mobile versions of an application

Thank you!

- Michael Birch
  - Telegram: @birchmd
  - https://github.com/birchmd/
  - https://www.typedriven.ca/news/

- Aurora
  - https://aurora.dev/

- Near
  - https://near.org/