Safe Smart Contract Programming with Scilla

Ilya Sergey

Associate Professor, Yale-NUS College
Lead Language Designer, Zilliqa

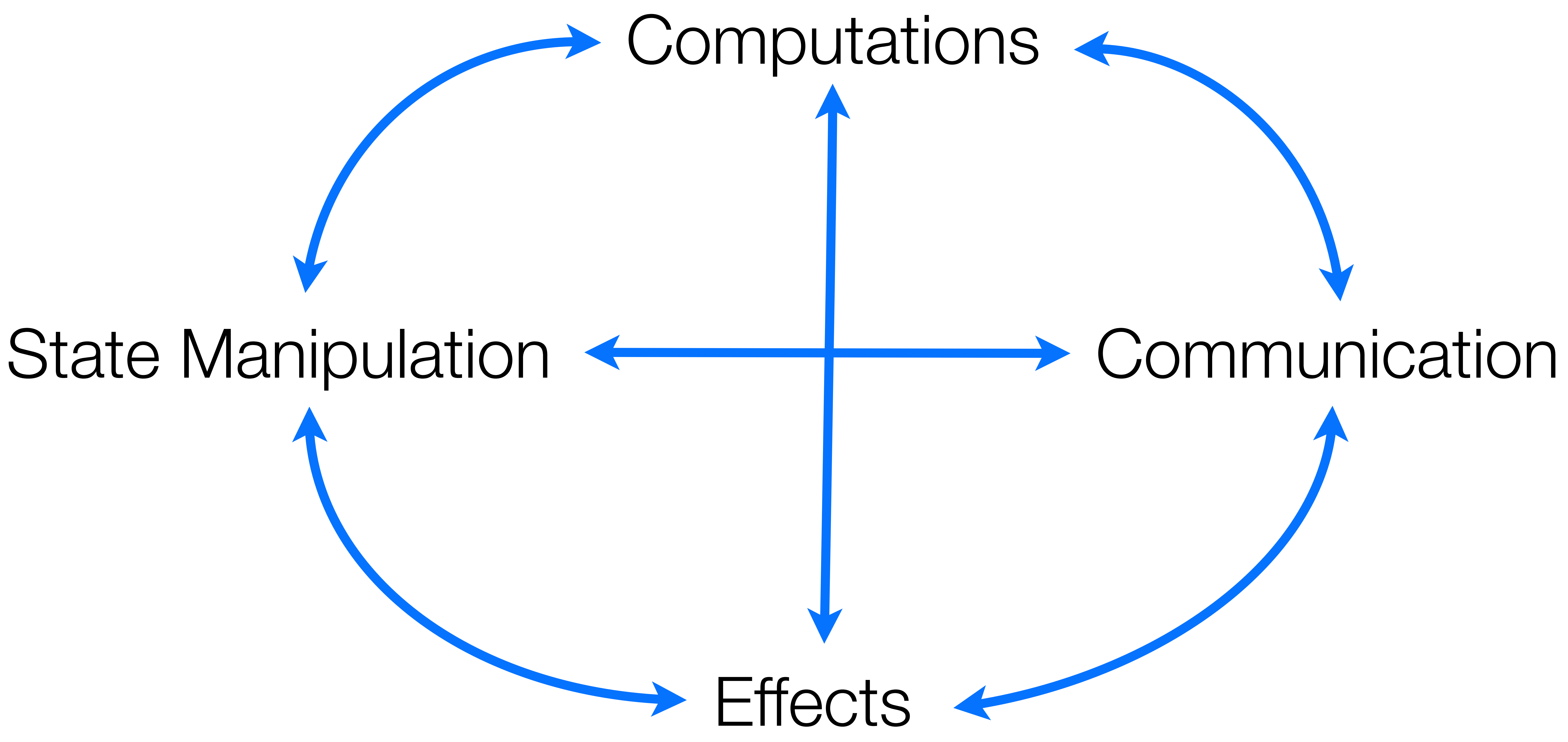
http://ilyasergey.net

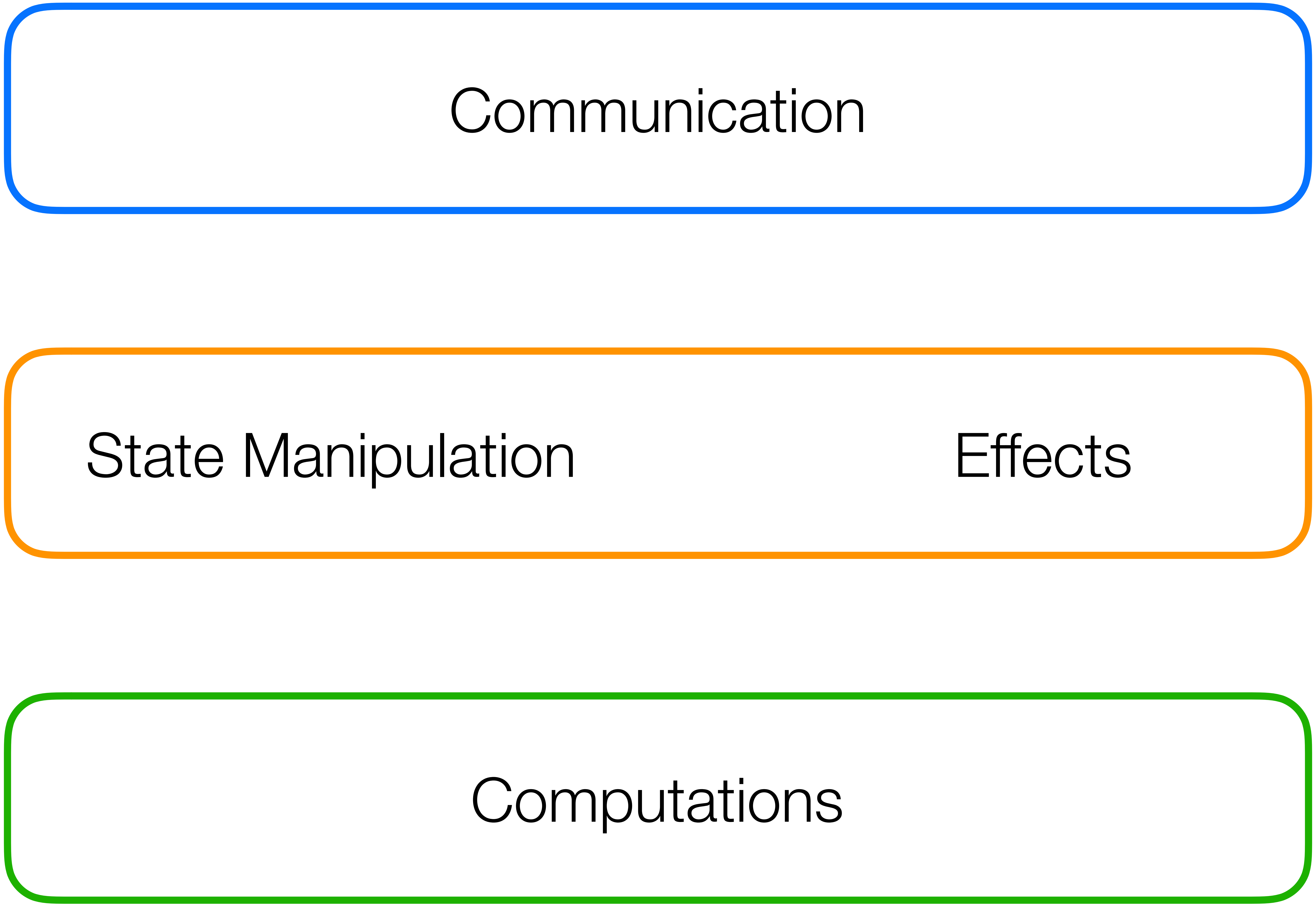
Smart Contracts

- *Stateful mutable* objects replicated via a consensus protocol
- State typically involves a stored amount of *funds/currency*
- One or more entry points: invoked *reactively* by a client *transaction*
- Main usages:
    - crowdfunding and ICO
    - multi-party accounting
    - voting and arbitration
    - puzzle-solving games with distribution of rewards
- Supporting platforms: **Ethereum**, **Tezos**, **Zilliqa**, …

Smart Contracts in a Nutshell

Computations	obtaining values from inputs
State Manipulation	changing contract's fields
Effects	accepting funds, logging events
Communication	sending funds, calling other contracts


Computations
State Manipulation
Communication
Effects


Communication
State Manipulation
Effects
Computations

Scilla

Smart Contract Intermediate-Level Language

Principled model for computations	**System** ***F*** with small extensions
Not Turing-complete	Only *structural recursion*/iteration
Explicit Effects	*State-transformer* semantics
Communication	Contracts are *autonomous actors*

Scilla Pragmatics

- Open source: github.com/Zilliqa/scilla
- Intentionally minimalistic: a *small language* is easier to reason about
- Implemented in OCaml (and a bit of C++), ~6 kLOC
- *Reference evaluator* is only ~350 LOC
- *Mostly* purely functional, Statically Typed
- Inspired by OCaml, Haskell, Scala, and Erlang

Statically Typed

Haskell Curry

- Types describe the *sets* of programs
- *Well-typed programs don't go wrong.*
  - No applying an **Int** (as a function) to a **String**
  - No adding **List** to **Bool**
  - No mishandled/forgotten arguments
  - No *ill-formed messages*
  - *etc.*

Robin Milner

Follow the code!

github.com/ilyasergey/scilla-demo

Types

t	::=	p	*Primitive types*
		C t_1 … t_n	*Algebraic data types*
		t_1 -> t_2	*Functions*
		‘A	*Type variables*
		forall ‘A . t	*Polymorphic types*
		Map t_1 t_2	*Maps*

Types

t	::=	p	*Primitive types*
		C t_1 … t_n	*Algebraic data types*
		t_1 -> t_2	*Functions*
		‘A	*Type variables*
		forall ‘A . t	*Polymorphic types*
		Map t_1 t_2	*Maps*

Primitive types and Values

p ::= Int32, Int64, Int128, Int256

Uint32, Uint64, Uint128, Uint256

String

ByStrX, ByStr

BNum

Message

Types

t	::=	p	*Primitive types*
		C t_1 … t_n	*Algebraic data types*
		t_1 -> t_2	*Functions*
		‘A	*Type variables*
		forall ‘A . t	*Polymorphic types*
		Map t_1 t_2	*Maps*

Types

t	::=	p	*Primitive types*
		C t_1 … t_n	*Algebraic data types*
		t_1 -> t_2	*Functions*
		‘A	*Type variables*
		forall ‘A . t	*Polymorphic types*
		Map t_1 t_2	*Maps*

Structural Recursion in Scilla

Natural numbers (not Ints!)

```
nat_rec : forall α . α -> (nat -> α -> α) -> nat -> α
```

Result type

Value for 0

number of iterations

constructing the next value

final result

Structural Recursion with Lists

$$\texttt{list\_rec} : \textbf{forall}\ \alpha\ \beta.\ \beta\ \texttt{->}\ \underbrace{(\alpha\ \texttt{->}\ \texttt{list}\ \alpha\ \texttt{->}\ \beta\ \texttt{->}\ \beta)}_{\text{Iterator for non-empty list}}\ \texttt{->}\ \texttt{list}\ \alpha\ \texttt{->}\ \beta$$

Element type

Result type

Value for Nil

argument list

argument list

Types

t	::=	p	*Primitive types*
		C t_1 … t_n	*Algebraic data types*
		t_1 -> t_2	*Functions*
		‘A	*Type variables*
		forall ‘A . t	*Polymorphic types*
		Map t_1 t_2	*Maps*

Expressions (pure)

Expression	e	::=	f	simple expression
			let x ⟨: T⟩ = f **in** e	let-form
Simple expression	f	::=	l	primitive literal
			x	variable
			{ ⟨*entry*⟩$_k$ }	Message
			fun (x : T) => e	function
			builtin b ⟨x_k⟩	built-in application
			x ⟨x_k⟩	application
			tfun α => e	type function
			@x T	type instantiation
			C ⟨ {⟨T_k⟩} ⟩ ⟨x_k⟩	constructor instantiation
			match x **with** ⟨ | sel_k ⟩ **end**	pattern matching
Selector	*sel*	::=	*pat* => e	
Pattern	*pat*	::=	x	variable binding
			C ⟨pat_k⟩	constructor pattern
			(*pat*)	paranthesized pattern
			_	wildcard pattern
Message entrry	*entry*	::=	$b : x$	
Name	b			identifier

Statements (effectful)

`s ::=`	`x <- f`	read from mutable field
	`f := x`	store to a field
	`x = e`	assign a pure expression
	`match` `x` **`with`** `⟨pat => s⟩` **`end`**	pattern matching and branching
	`x <- &B`	read from blockchain state
	`accept`	accept incoming payment
	`event` `m`	create a single event
	`send` `ms`	send list of messages

Statement Semantics

$$[\![s]\!] : BlockchainState \to Configuration \to Configuration$$

$BlockchainState$ Immutable global data (block number *etc.*)

$$Configuration = Env \times Fields \times Balance \times Incoming \times Emitted$$

Immutable bindings

Mutable fields

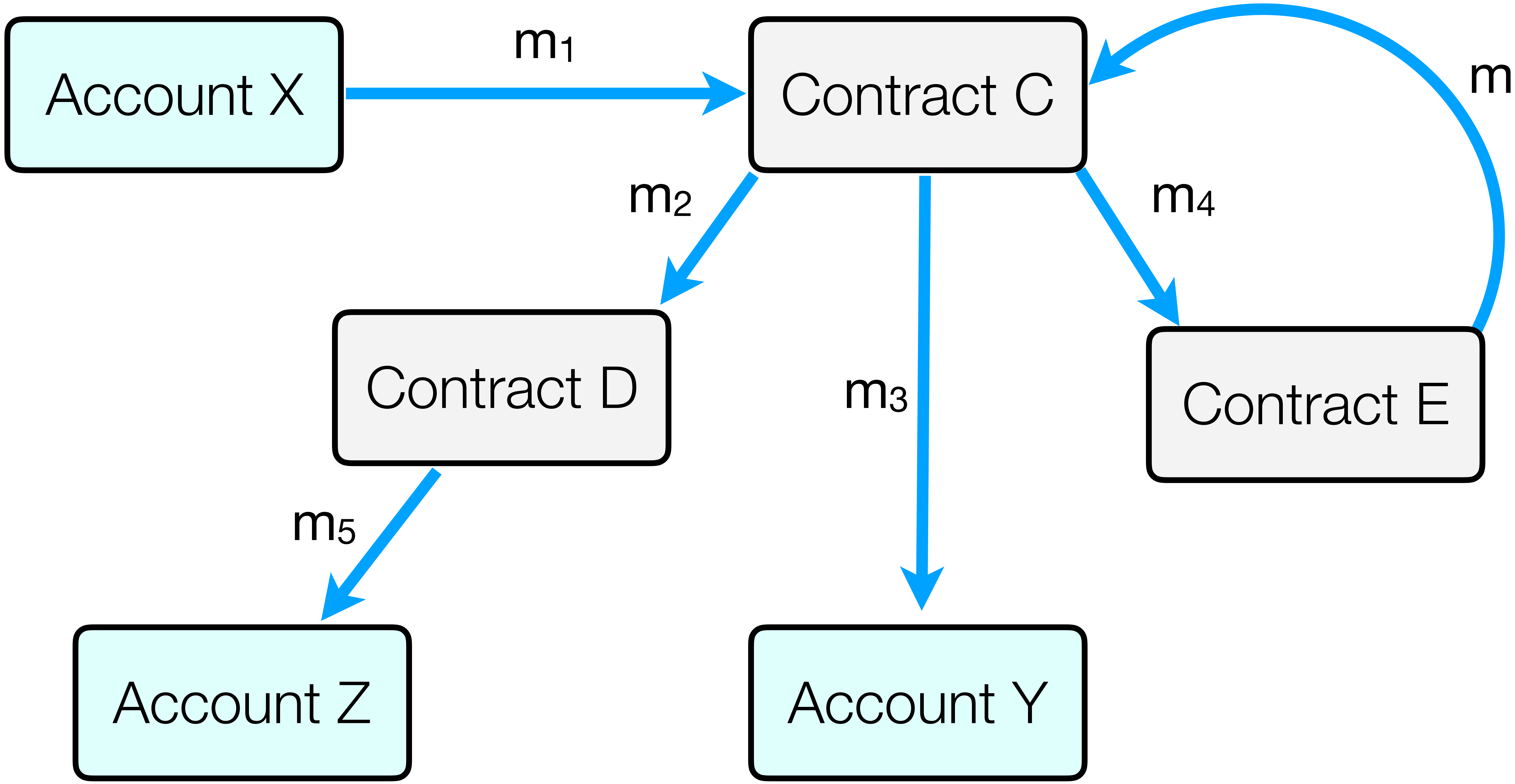
Contract's own funds

Funds sent to contract

Messages and events to be sent

Global Execution Model

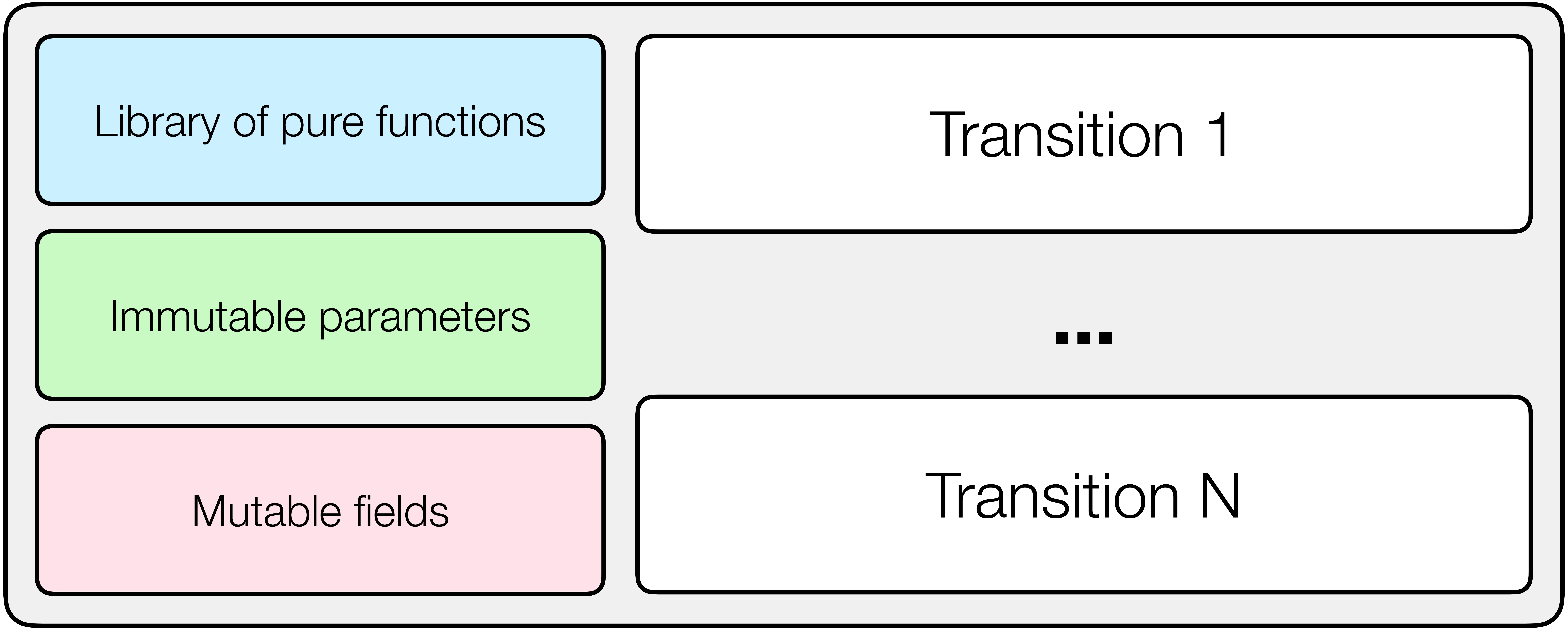
Global Execution Model



Putting it All Together

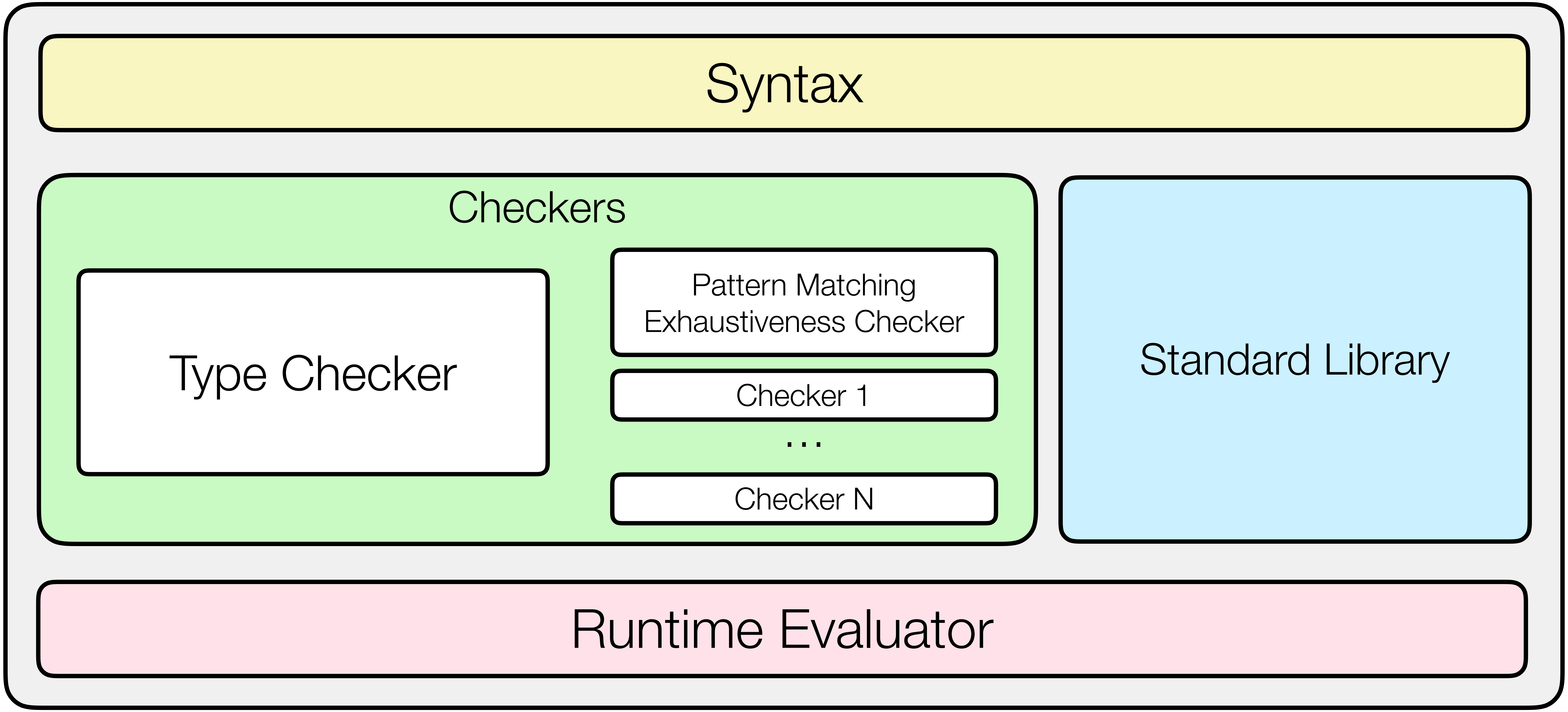
- Scilla contracts are (infinite) *State-Transition Systems*
- Interaction *between* contracts via sending/receiving *messages*
- Messages trigger (effectful) *transitions* (sequences of *statements*)
- A contract can *send messages* to other contracts via **`send`** statement
- Most computations are done via *pure expressions*, no storable closures
- Contract's state is immutable parameters, mutable fields, balance

Contract Structure



Demo

Scilla as a Framework



How can you contribute?

- Implementing contracts in Scilla
- Tooling support for better user experience
- Language Infrastructure and Checkers

Jacob Johannsen

Amrit Kumar

Edison Lim

Vaivaswatha Nagaraj

Ilya Sergey

Ian Tan

Han Wen Chua

More resources

- http://scilla-lang.org
- https://github.com/Zilliqa/scilla

Thanks!