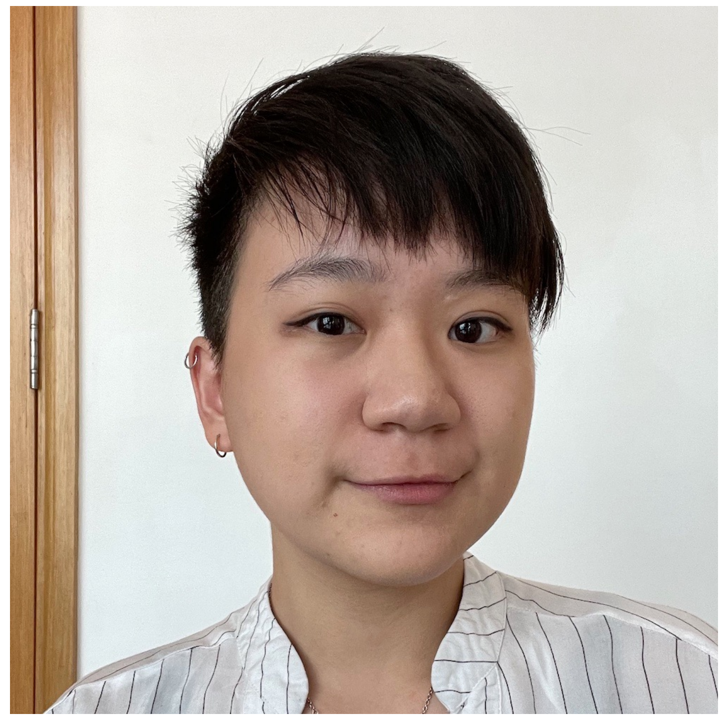
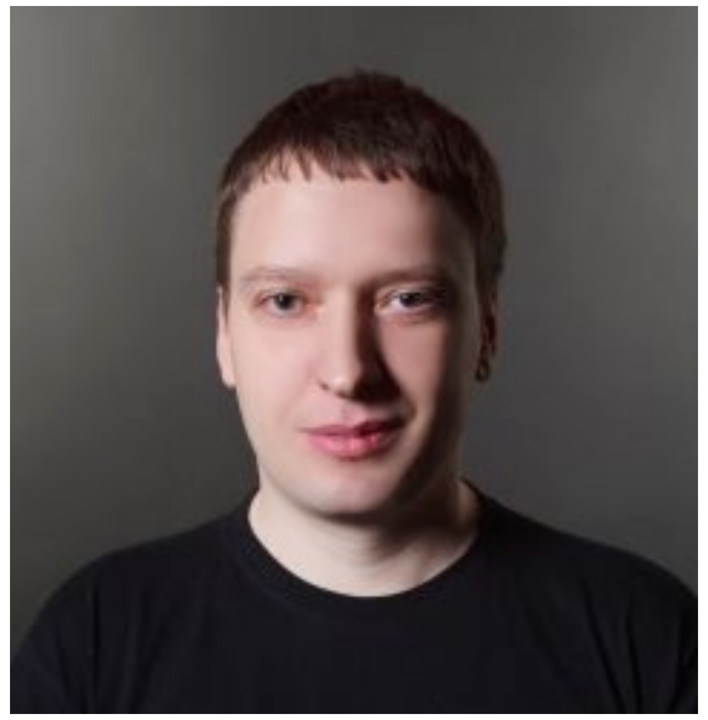
Random Testing of a Higher-Order Blockchain Language

(*Experience Report*)



Tram Hoang

National University of Singapore



Anton Trunov

Zilliqa

Leonidas Lampropoulos

University of Maryland, College Park

Ilya Sergey

National University of Singapore

ICFP 2022

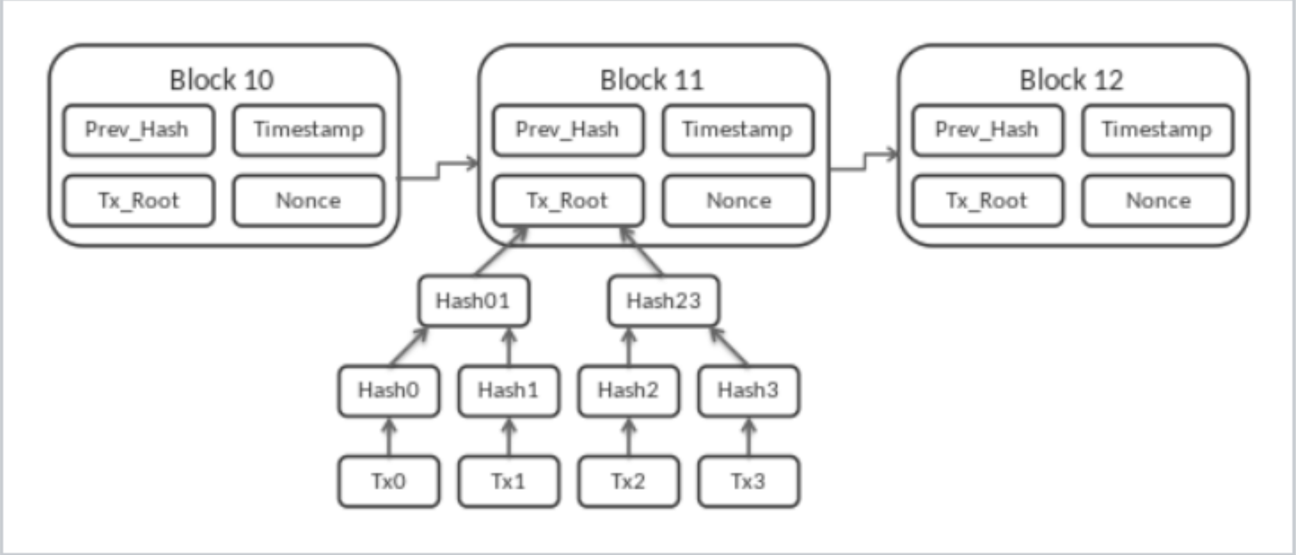
What is Blockchain?

Blockchain

From Wikipedia, the free encyclopedia

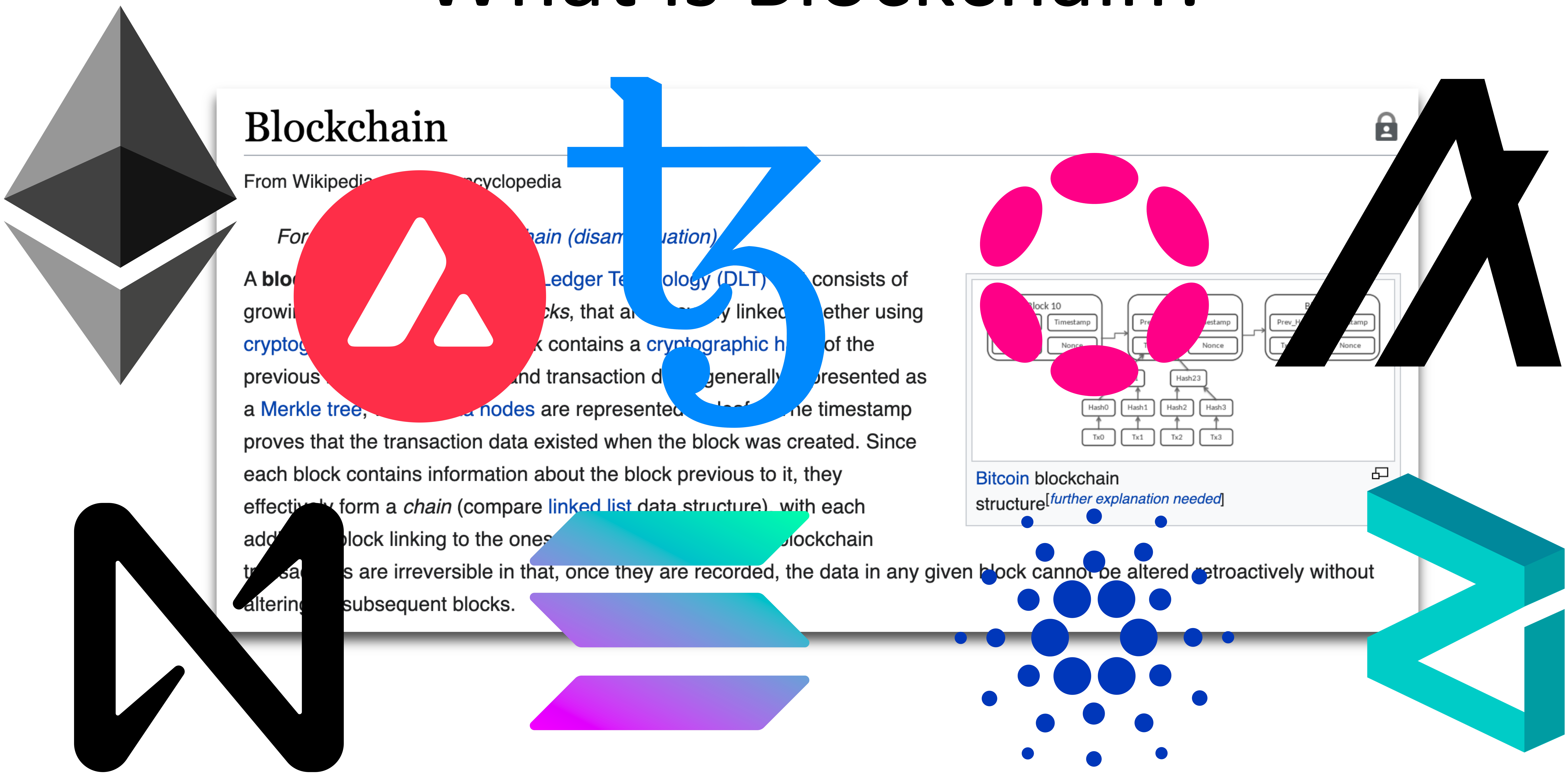
For other uses, see Block chain (disambiguation).

A **blockchain** is a type of Digital Ledger Technology (DLT) that consists of growing list of records, called *blocks*, that are securely linked together using cryptography.[1][2][3][4] Each block contains a cryptographic hash of the previous block, a timestamp, and transaction data (generally represented as a Merkle tree, where data nodes are represented by leafs). The timestamp proves that the transaction data existed when the block was created. Since each block contains information about the block previous to it, they effectively form a *chain* (compare linked list data structure), with each additional block linking to the ones before it. Consequently, blockchain transactions are irreversible in that, once they are recorded, the data in any given block cannot be altered retroactively without altering all subsequent blocks.



Bitcoin blockchain structure[*further explanation needed*]

What is Blockchain?



ICFP 2019 Keynote

ICFP 2019 (series) / Keynotes and Reports /

Blockchains are functional

Track ICFP 2019 Keynotes and Reports

When **Mon 19 Aug 2019 09:00 - 10:00 at Aurora Borealis** - Monday Keynote Chair(s): Derek Dreyer

Abstract Functional programming and blockchains are a match made in heaven! The immutable and reproducible nature of distributed ledgers is mirrored in the semantic foundation of functional programming. Moreover, the concurrent and distributed operation calls for a programming model that carefully controls shared mutable state and side effects. Finally, the high financial stakes often associated with blockchains suggest the need for high assurance software and formal methods.

Nevertheless, most existing blockchains favour an object-oriented, imperative approach in both their implementation as well as in the contract programming layer that provides user-defined custom functionality on top of the basic ledger. On the one hand, this might appear surprising, given that it is widely understood that this style of programming is particularly risky in concurrent and distributed systems. On the other hand, blockchains are still in their infancy and little research has been conducted into associated programming language technology.

Manuel Chakravarty

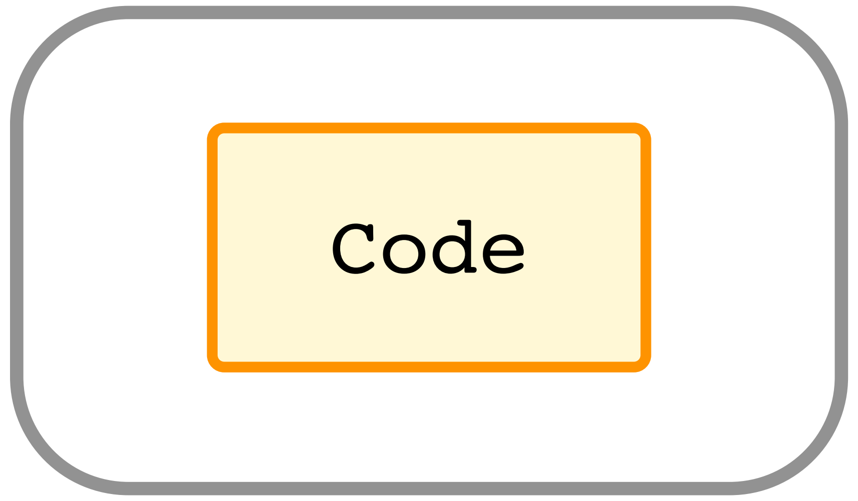
Tweag I/O & IOHK

Getting Your Code on a Blockchain

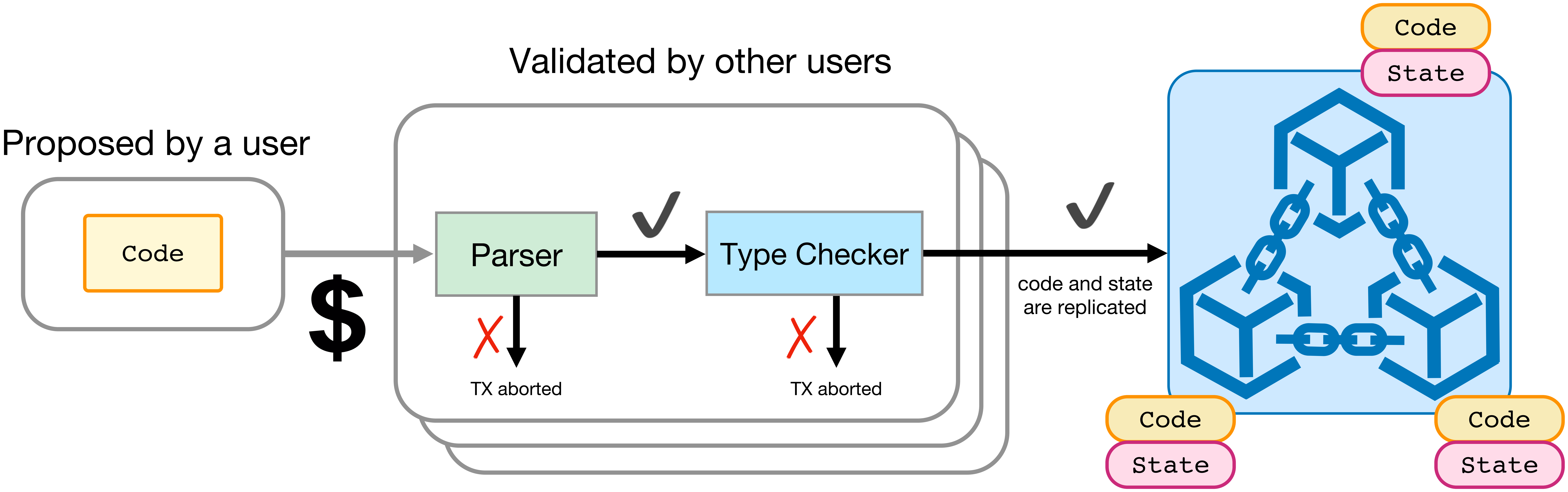
```
scilla_version 0
library FungibleToken
let min_int : Uint128 → Uint128 → Uint128 = (* ... *)
let le_int  : Uint128 → Uint128 → Bool    = (* ... *)
let one_msg : Msg → List Msg = (* Return singleton List with a message *)

contract FungibleToken
(owner : ByStr20, total_tokens : Uint128, decimals : Uint32, name : String, symbol : String)

field balances : Map ByStr20 Uint128 =
  let m = Emp ByStr20 Uint128 in builtin put m owner total_tokens
field allowed : Map ByStr20 (Map ByStr20 Uint128) = Emp ByStr20 (Map ByStr20 Uint128)

transition BalanceOf (tokenOwner : ByStr20)
  bal ← balances[tokenOwner];
  match bal with
  | Some v ⇒
    msg = {_tag : "BalanceOfResponse"; _recipient : _sender; address : tokenOwner; balance : v};
    msgs = one_msg msg; send msgs
  | None ⇒
    msg = {_tag : "BalanceOfResponse"; _recipient : _sender; address : tokenOwner; balance : zero};
    msgs = one_msg msg; send msgs
  end
end
transition TotalSupply () (* code omitted *) end
transition Transfer (to : ByStr20, tokens : Uint128) (* code omitted *) end
transition TransferFrom (from : ByStr20, to : ByStr20, tokens : Uint128) (* code omitted *) end
transition Approve (spender : ByStr20, tokens : Uint128) (* code omitted *) end
transition Allowance (tokenOwner : ByStr20, spender : ByStr20) (* code omitted *) end
```

Getting Your Code on a Blockchain

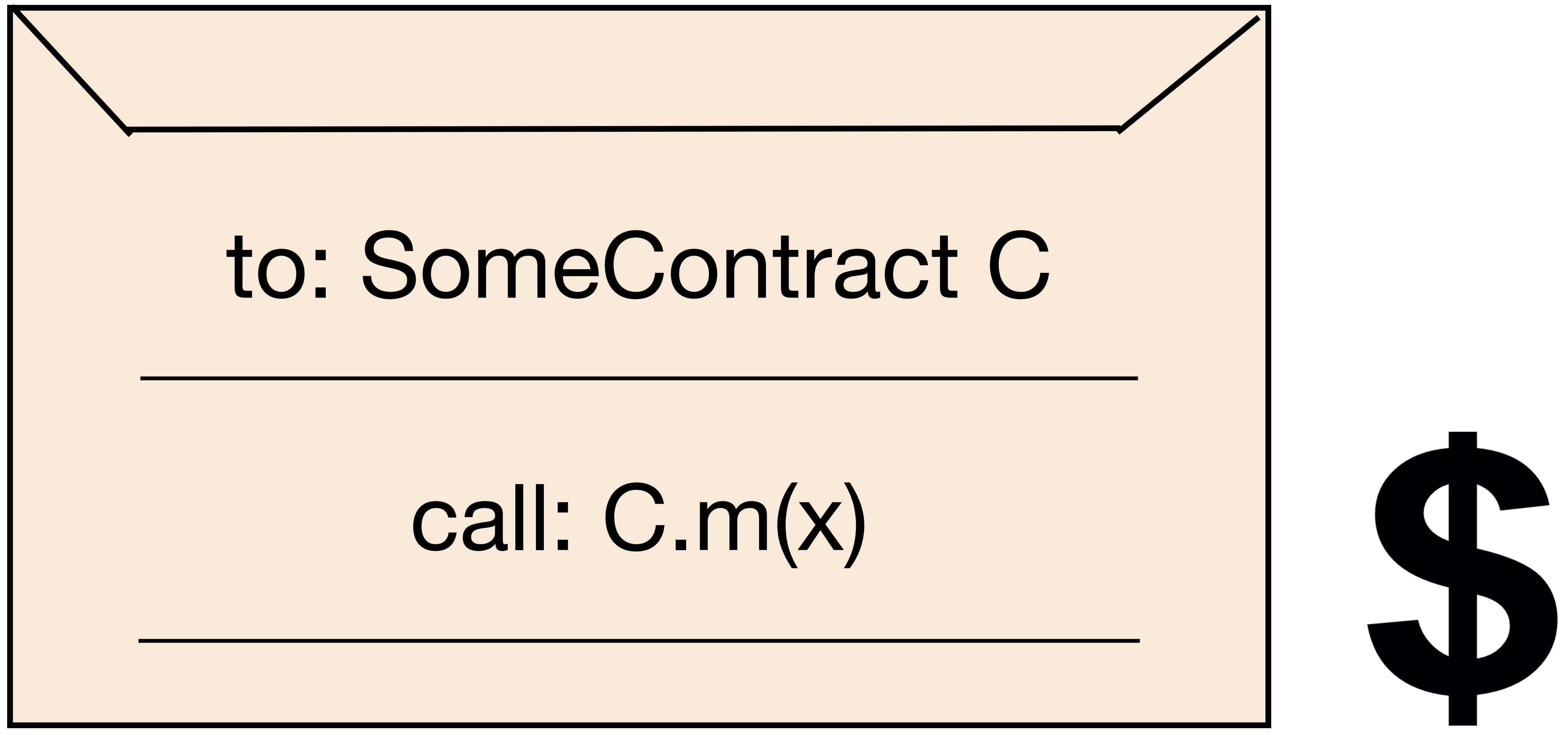
Proposed by a user



Getting Your Code on a Blockchain



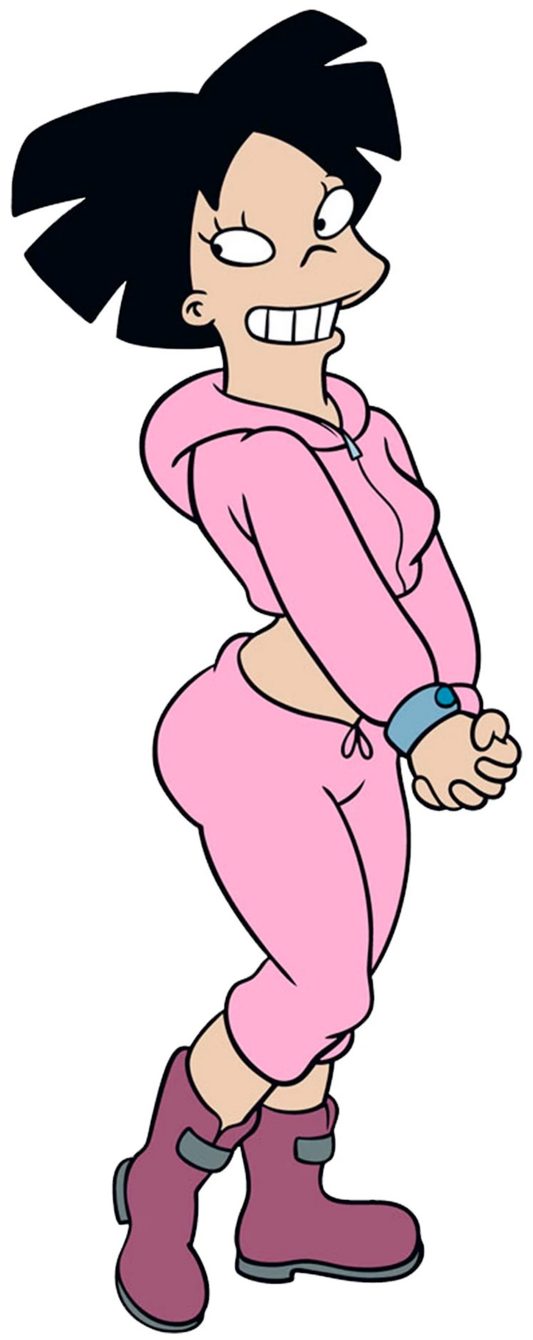
Using Your Code on a Blockchain



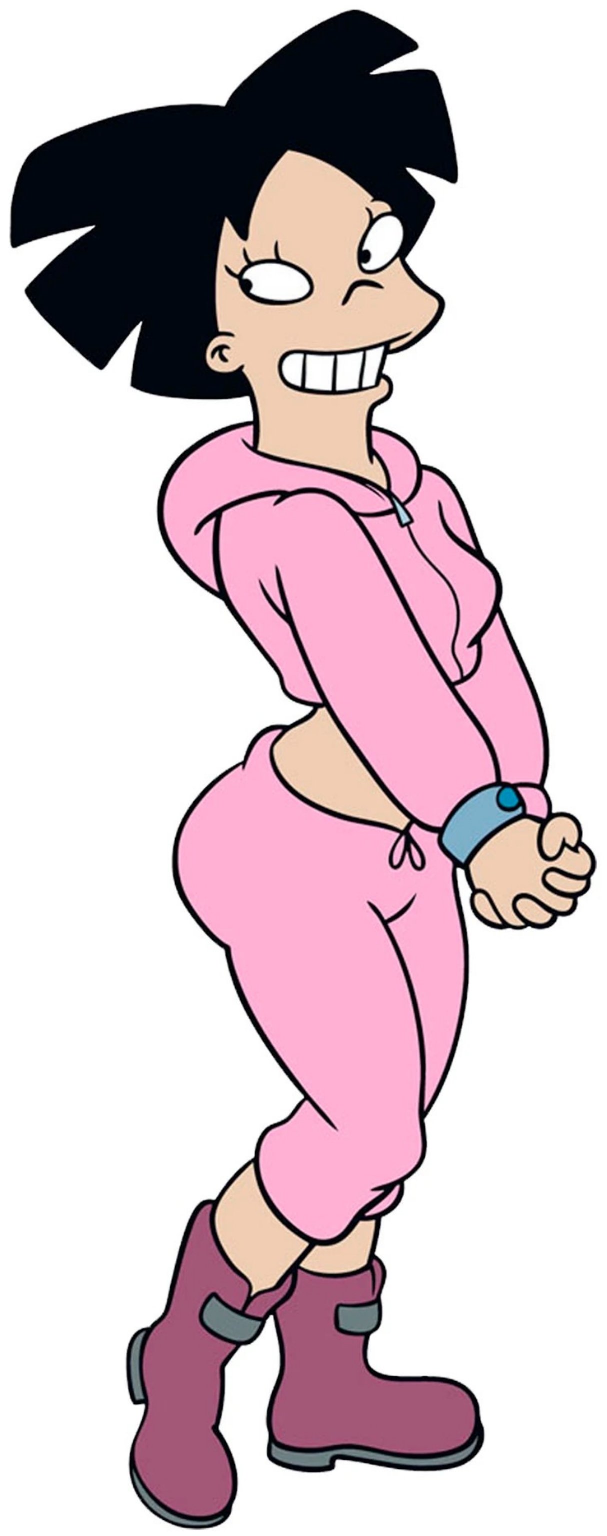
Using Your Code on a Blockchain

Using Your Code on a Blockchain

What Can Go Wrong?



Scenario 1: Static Semantics Bug

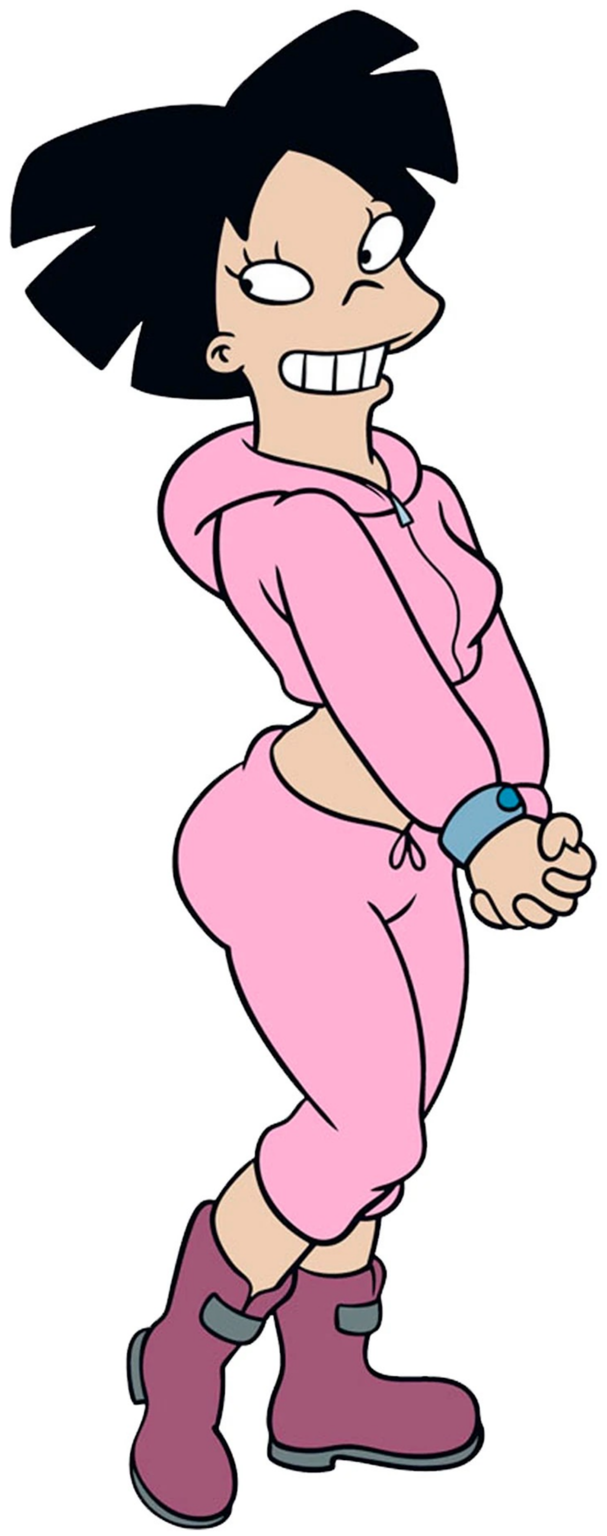


Fund_Amy.code

```
fun withdraw_donations _ =
  …
  for (b <- backer_accounts) do
    account_to_address(b) match
    | Some(addr) => …
    | None => …
  done
```

⊢ Fund_Amy OK => No Exceptions

Scenario 1: Static Semantics Bug



`Fund_Amy.code`

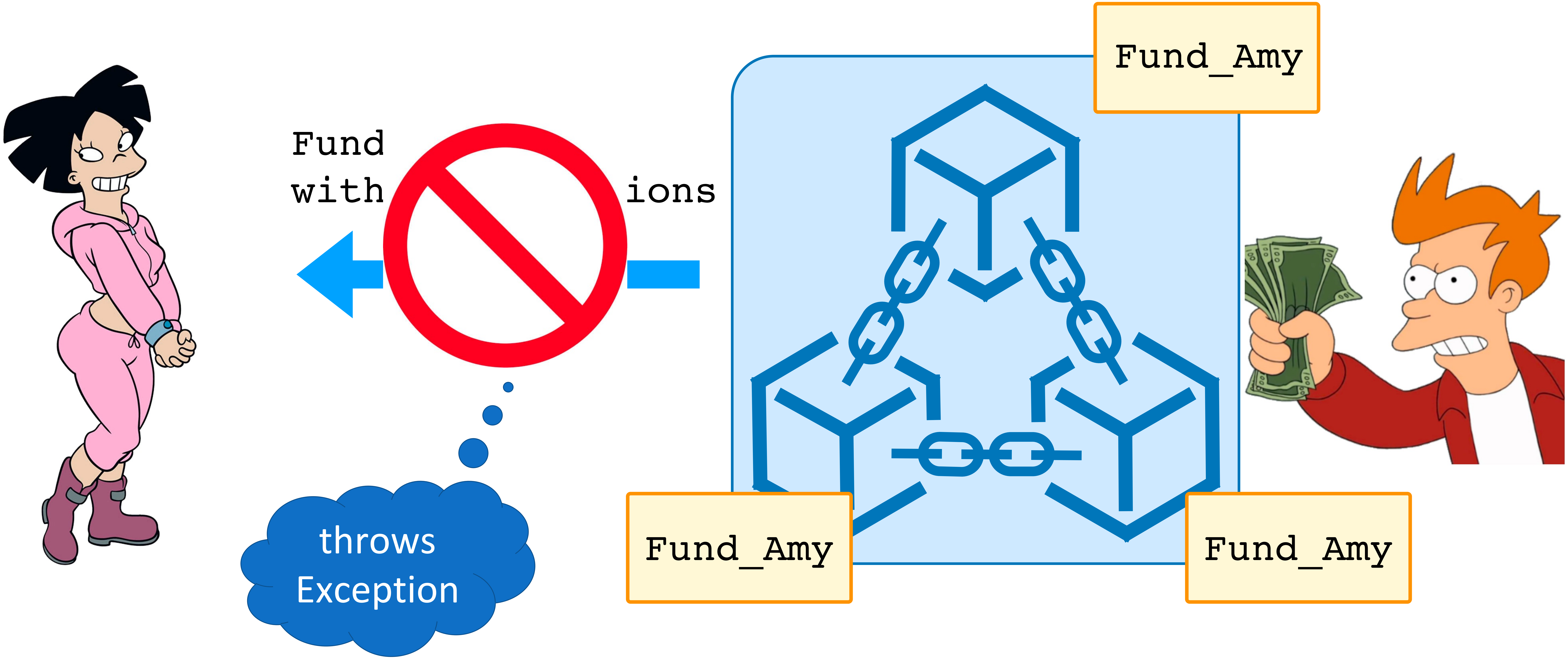
```
fun withdraw_donations _ =
  …
  for (b <- backer_accounts) d
    account_to_address(b) match
    | Some(addr) => …
    | None => …
  done
```

Throws an exception if b is ill-formed

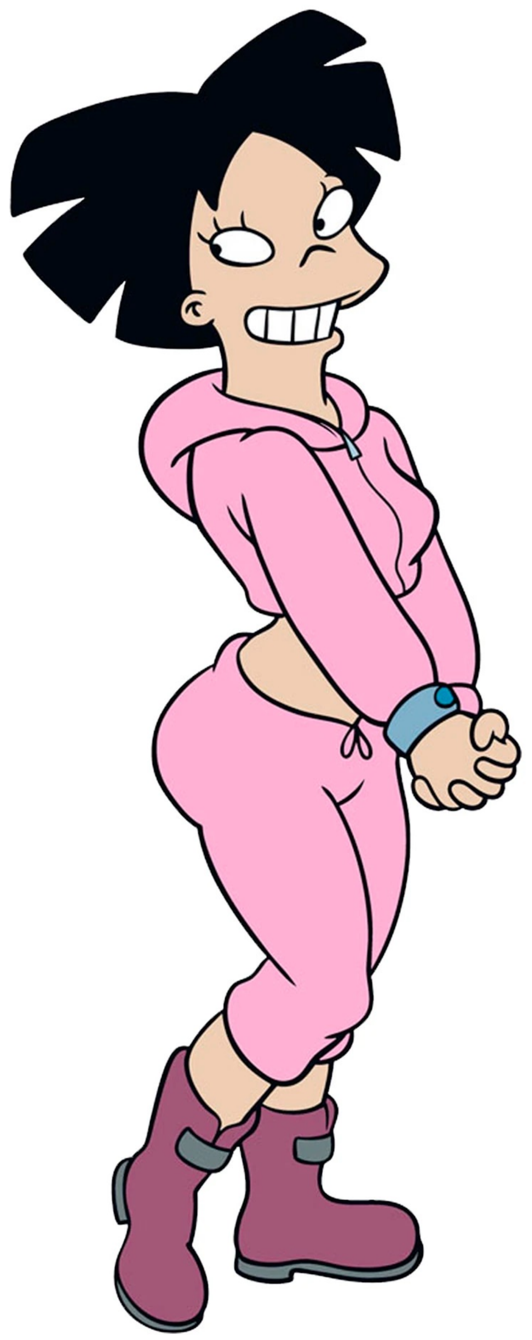
⊢ Fund_Amy OK => No Exceptions

WRONG!

Scenario 1: Static Semantics Bug



What Else Can Go Wrong?



Scenario 2: Cost Semantics Bug

Interpreter.ml

```
fun eval_arith_operation op args =
  …
  | Power base x =>
    let res = compute_pow base x in
    let gas_charged = log2 x in
    (res, gas_charged)
```

Scenario 2: Cost Semantics Bug

Interprete

```
fun eval_arith_operation                    =
  …
  | Power base x =>
    let res = compute_pow base x in
    let gas_charged = log2 x in
    (res, gas_charged)
```

Replicated by many users to re-validate

Takes O(x) in Leela's implementation

Scenario 2: Cost Semantics Bug

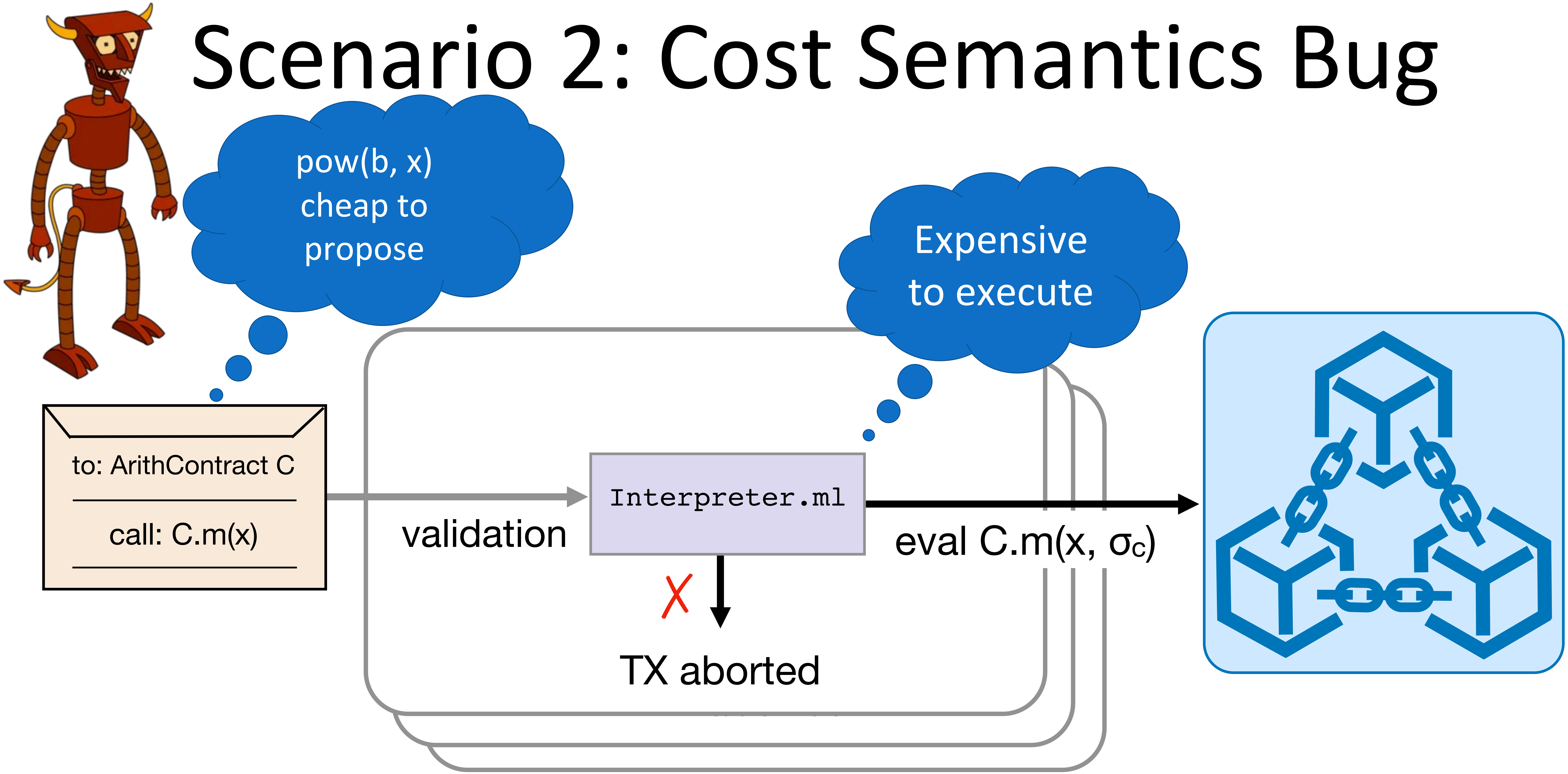
Interpreter.ml

```
fun eval_arith_operation op args =
  …
  | Power base x =>
    let res = compute_pow base x in
    let gas_charged = log2 x in
    (res, gas_charged)
```

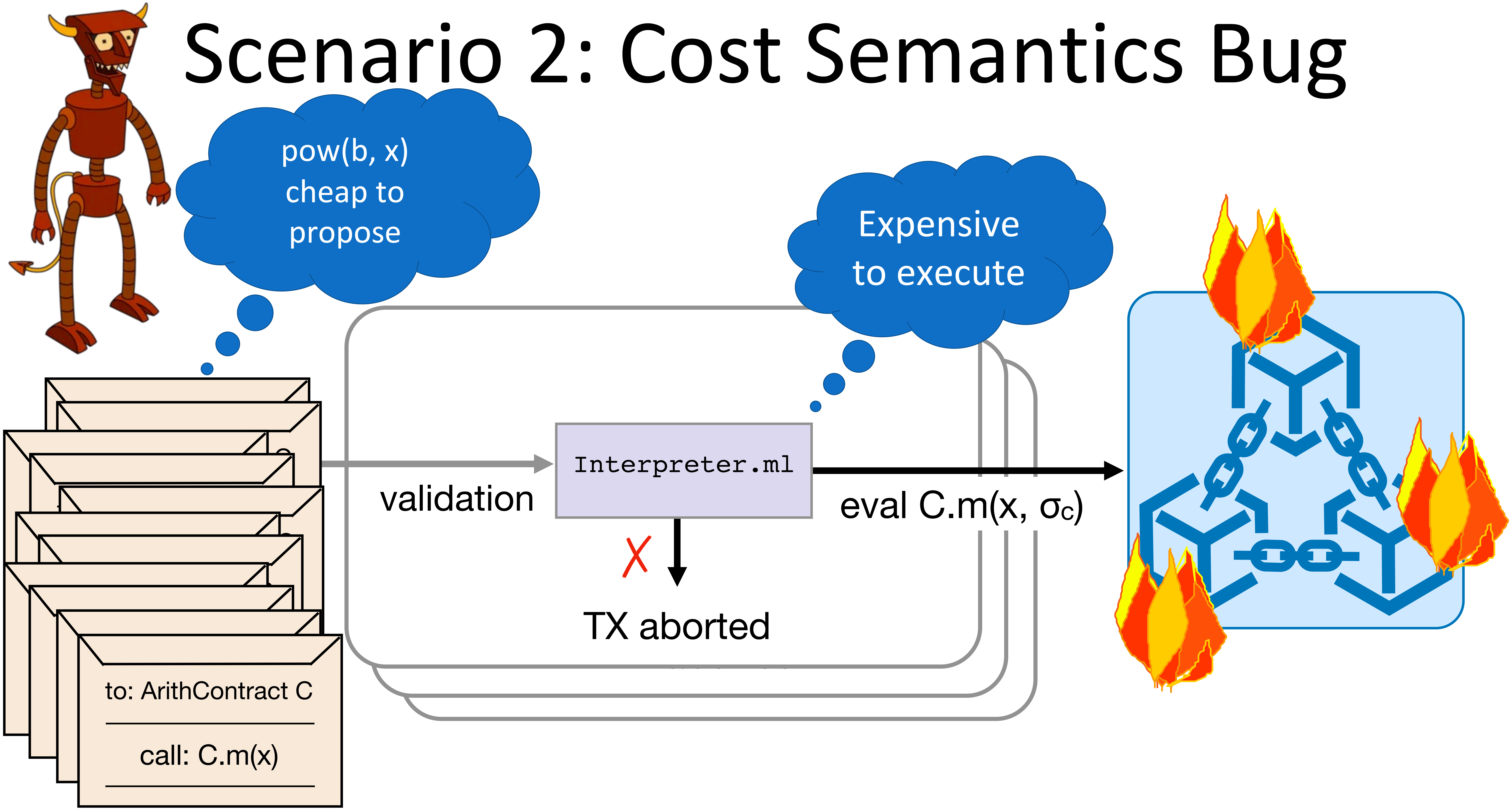
Takes O(x) in Leela’s implementation

Charges for computing power

Scenario 2: Cost Semantics Bug



Scenario 2: Cost Semantics Bug

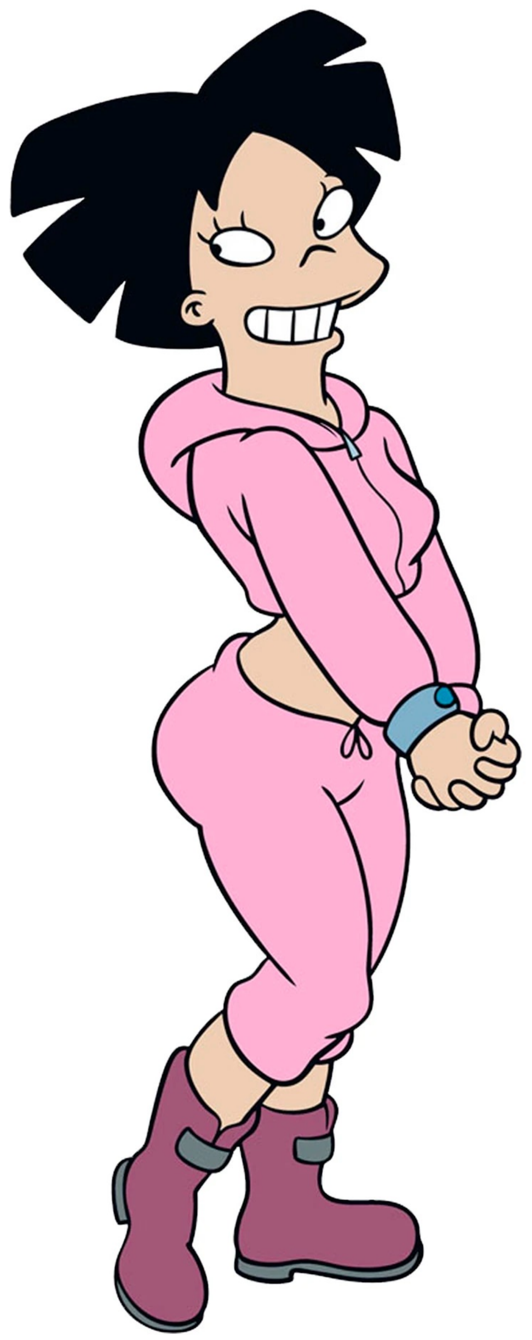


Scenario 2: Cost Semantics Bug

Interpreter.ml

```
fun eval_arith_operation op args =
  …
  | Power base x =>
    let res = compute_pow base x in
    let gas_charged = log2 x in
    (res, gas_charged)
```

What Else Can Go Wrong?



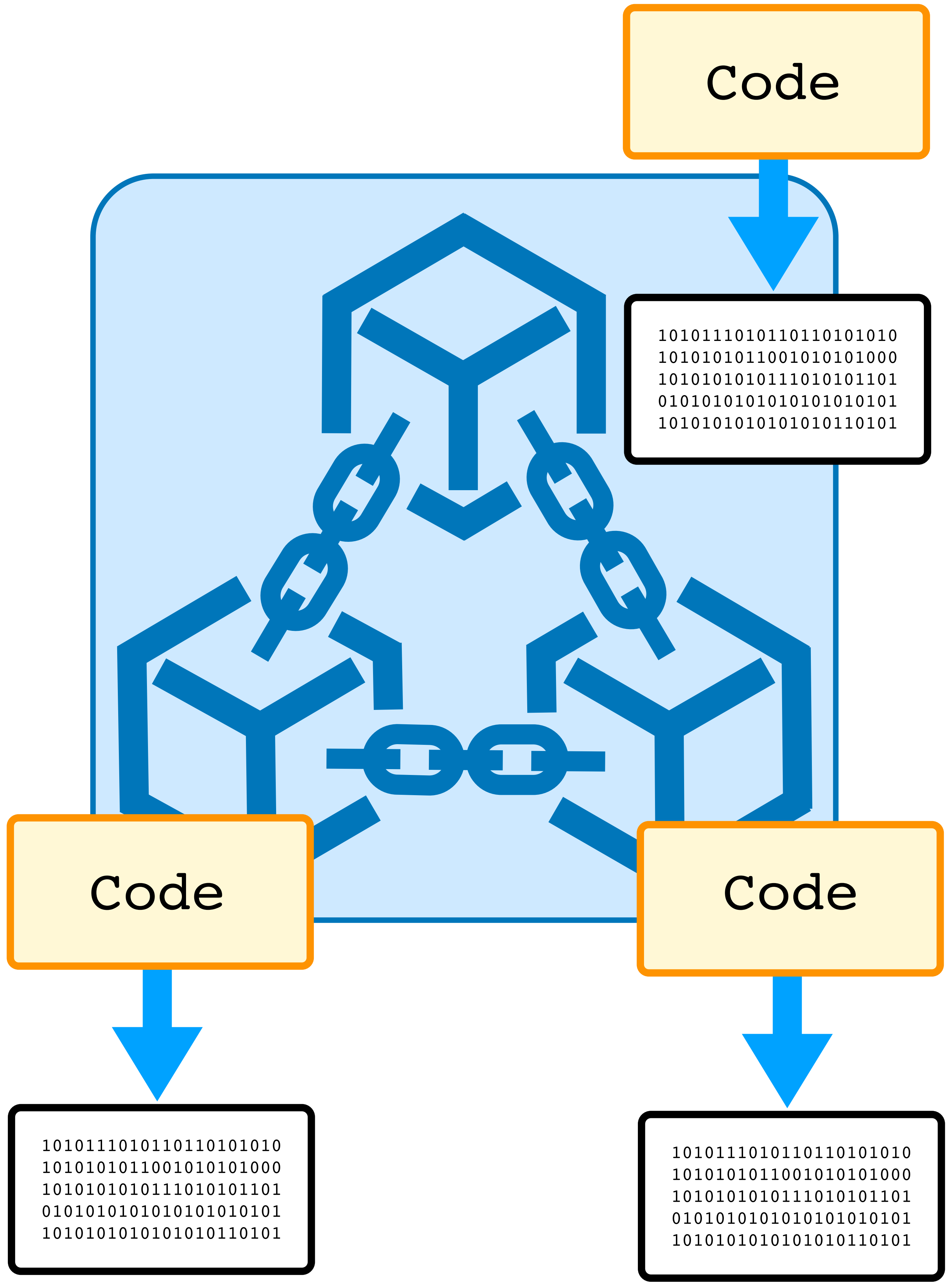
Scenario 3: A Compiler Exploit

```
scilla_version 0
library FungibleToken
let min_int : Uint128 → Uint128 → Uint128 = (* ... *)
let le_int  : Uint128 → Uint128 → Bool    = (* ... *)
let one_msg : Msg → List Msg = (* Return singleton List with a message *)

contract FungibleToken
(owner : ByStr20, total_tokens : Uint128, decimals : Uint32, name : String, symbol : String)

field balances : Map ByStr20 Uint128 =
  let m = Emp ByStr20 Uint128 in builtin put m owner total_tokens
field allowed : Map ByStr20 (Map ByStr20 Uint128) = Emp ByStr20 (Map ByStr20 Uint128)

transition BalanceOf (tokenOwner : ByStr20)
  bal ← balances[tokenOwner];
  match bal with
  | Some v ⇒
    msg = {_tag : "BalanceOfResponse"; _recipient : _sender; address : tokenOwner; balance : v};
    msgs = one_msg msg; send msgs
  | None ⇒
    msg = {_tag : "BalanceOfResponse"; _recipient : _sender; address : tokenOwner; balance : zero};
    msgs = one_msg msg; send msgs
  end
end
transition TotalSupply () (* code omitted *) end
transition Transfer (to : ByStr20, tokens : Uint128) (* code omitted *) end
transition TransferFrom (from : ByStr20, to : ByStr20, tokens : Uint128) (* code omitted *) end
transition Approve (spender : ByStr20, tokens : Uint128) (* code omitted *) end
transition Allowance (tokenOwner : ByStr20, spender : ByStr20) (* code omitted *) end
```

10x speedup
in transaction
processing!

```
10101110101101101010100
10101010110010101010000
10101010101110101011101
01010101010101010101010
10101010101010101110101
```

Scenario 3: A Replicated Compiler Exploit



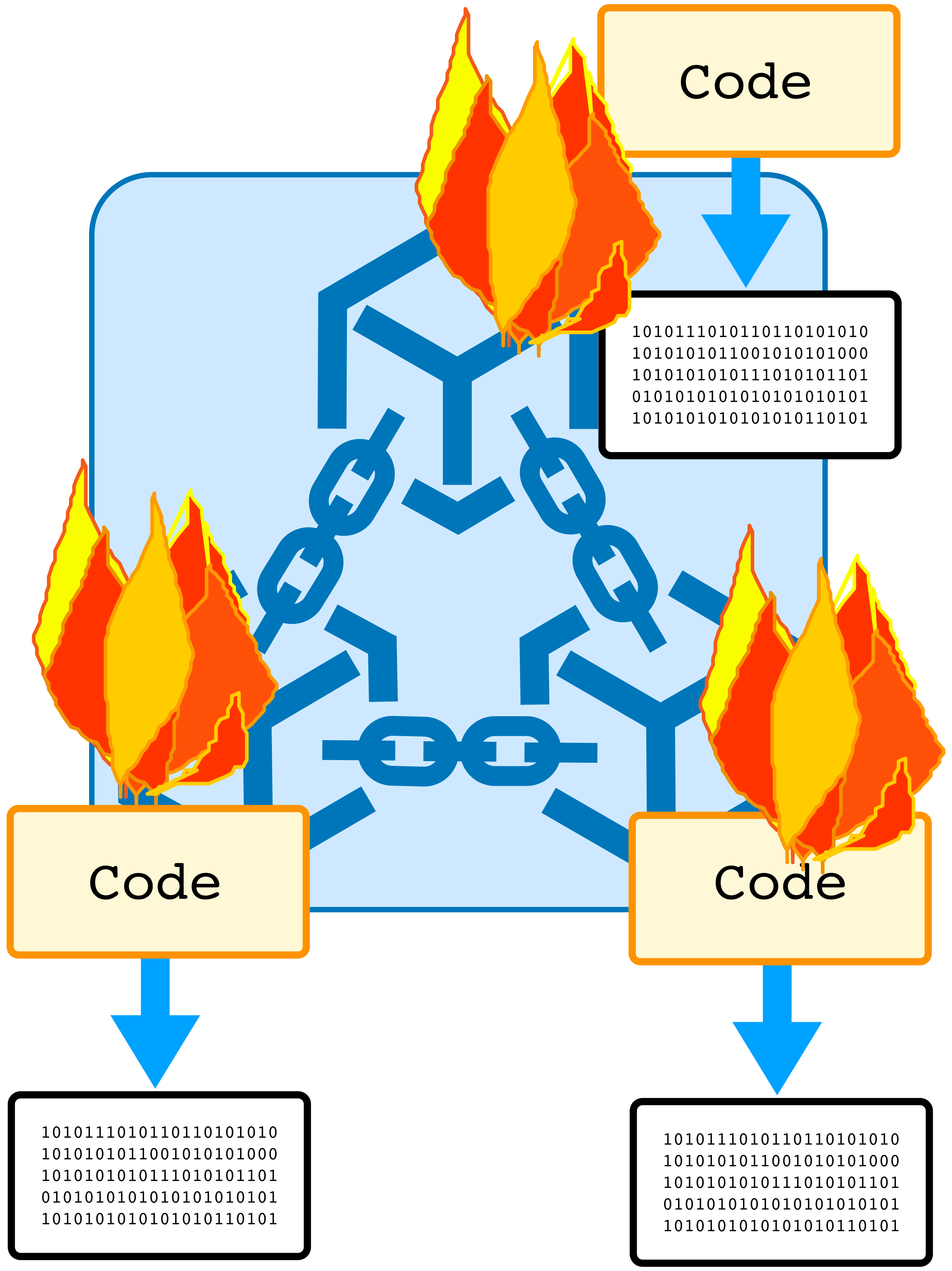
Scenario 3: A Replicated Compiler Exploit

```
scilla_version 0
library FungibleToken
let min_int : Uint128 → Uint128 → Uint128 = (* ... *)
let le_int  : Uint128 → Uint128 → Bool    = (* ... *)
let one_msg : Msg → List Msg = (* Return singleton List with a message *)

contract FungibleToken
(owner : ByStr20, total_tokens : Uint128, decimals : Uint32, name : String, symbol : String)

field balances : Map ByStr20 Uint128 =
  let m = Emp ByStr20 Uint128 in builtin put m owner total_tokens
field allowed : Map ByStr20 (Map ByStr20 Uint128) = Emp ByStr20 (Map ByStr20 Uint128)

transition BalanceOf (tokenOwner : ByStr20)
  bal ← balances[tokenOwner];
  match bal with
  | Some v ⇒
    msg = {_tag : "BalanceOfResponse"; _recipient : _sender; address : tokenOwner; balance : v};
    msgs = one_msg msg; send msgs
  | None ⇒
    msg = {_tag : "BalanceOfResponse"; _recipient : _sender; address : tokenOwner; balance : zero};
    msgs = one_msg msg; send msgs
  end
end
transition TotalSupply () (* code omitted *) end
transition Transfer (to : ByStr20, tokens : Uint128) (* code omitted *) end
transition TransferFrom (from : ByStr20, to : ByStr20, tokens : Uint128) (* code omitted *) end
transition Approve (spender : ByStr20, tokens : Uint128) (* code omitted *) end
transition Allowance (tokenOwner : ByStr20, spender : ByStr20) (* code omitted *) end
```

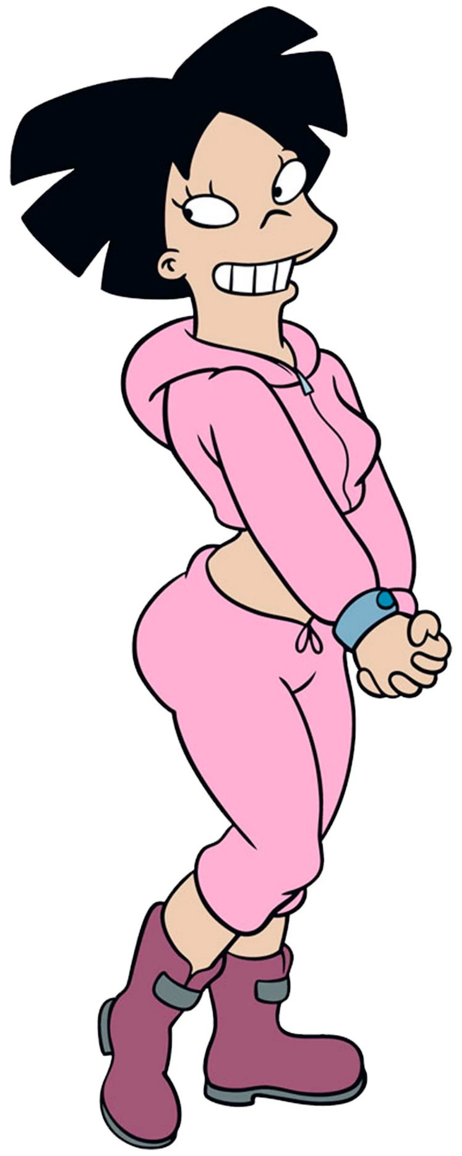
$O(n^3)$
in the
program size

```
1010111010110110101010
1010101011001010101000
1010101010111010101101
0101010101010101010101
1010101010101010110101
```

Scenario 3: A Replicated Compiler Exploit



Language-Layer Bugs



- **Type Checker & Interpreter**:
  static guarantees are not ensured at runtime

- **Reference Interpreter:**
  cost semantics is misaligned with runtime costs

- **Compiler:**
  Compilation cost is not linear in the program size

This Talk

Catching Bugs

in the Blockchain Language Layer

using Property-Based Testing

The Rest of the Talk

- The Language
- The Testing Framework
- Generating Random Programs
- Semantic Harness for Testing
- Found Bugs

The Rest of the Talk

- The Language
- The Testing Framework
- Generating Random Programs
- Semantic Harness for Testing
- Found Bugs

Scilla

Smart Contract Intermediate Level Language

SCILLA

- Principled Model for Computation
  - System F + extensions
- Not Turing Complete
  - Only structural recursion/iteration
- Explicit Effects
  - State Transformer Semantics
- Communication
  - Contracts are autonomous actors

Used by zilliqa

OOPSLA'19

Artifacts Available acm
Artifacts Evaluated acm Reusable
Artifacts Evaluated acm Functional

185

Safer Smart Contract Programming with Scilla

ILYA SERGEY, Yale-NUS College, Singapore and National University of Singapore, Singapore
VAIVASWATHA NAGARAJ, Zilliqa Research, India
JACOB JOHANNSEN, Zilliqa Research, Denmark
AMRIT KUMAR, Zilliqa Research, United Kingdom
ANTON TRUNOV, Zilliqa Research, Russia
KEN CHAN GUAN HAO, Zilliqa Research, Malaysia

The rise of programmable open distributed consensus platforms based on the blockchain technology has aroused a lot of interest in replicated stateful computations, *aka smart contracts*. As blockchains are used predominantly in financial applications, smart contracts frequently manage millions of dollars worth of virtual coins. Since smart contracts cannot be updated once deployed, the ability to reason about their correctness becomes a critical task. Yet, the de facto implementation standard, pioneered by the Ethereum platform, dictates smart contracts to be deployed in a low-level language, which renders independent audit and formal verification of deployed code infeasible in practice.

Scilla

Pure Fragment: System F

Variables: $x, y, \ldots, X, Y, \ldots$

Primitives: $p ::= Int32 \mid Int64 \mid \ldots \mid Uint32 \mid \ldots \mid$
$\mid String \mid ByStr \mid ByStrX \mid Message \mid \ldots$

Constants : $c ::= 0 \mid 1 \mid \text{" "} \mid \ldots$

Types: $\sigma, \tau \ldots ::= p \mid \sigma \rightarrow \tau \mid X \mid \forall X.\tau$

Terms: $\mathrm{e}, \ldots ::= x \mid c \mid e_1 e_2 \mid \lambda x : \sigma.\mathrm{e}$
$\mid \mathrm{e}\ \tau \mid \Lambda \mathrm{X}.\mathrm{e} \mid \{\mathrm{f} : \mathrm{e}, \ldots\}$
$\mid \mathrm{C}\ \mathrm{e}_1 \ldots \mathrm{e}_\mathrm{n} \mid \mathrm{match\ e\ with} < \mathrm{pat} \Rightarrow \mathrm{sel} > \mathrm{end}$

Scilla

Pure Fragment: System F

Variables: $x, y, \ldots, X, Y, \ldots$

Primitives: $p ::= Int32 \mid Int64 \mid \ldots \mid Uint32 \mid \ldots \mid$
$\mid String \mid ByStr \mid ByStrX \mid Message \mid \ \ldots$

Constants : $c ::= 0 \mid 1 \mid " " \mid \ \ldots$

Types: $\sigma, \tau \ldots ::= \boldsymbol{p} \mid \boldsymbol{\sigma} \ \boldsymbol{\to} \boldsymbol{\tau} \mid \boldsymbol{X} \mid \forall \boldsymbol{X}. \boldsymbol{\tau}$

Terms: $e, \ldots ::= \boldsymbol{x} \mid \boldsymbol{c} \mid \boldsymbol{e_1 e_2} \mid \boldsymbol{\lambda x : \sigma.} \ \mathbf{e}$
$\mid \mathbf{e} \ \boldsymbol{\tau} \mid \boldsymbol{\Lambda} \mathbf{X}. \ \mathbf{e} \mid \{\mathrm{f} : \mathrm{e}, \ldots\}$
$\mid \mathrm{C} \ \mathrm{e}_1 \ldots \mathrm{e}_\mathrm{n} \mid \mathrm{match} \ \mathrm{e} \ \mathrm{with} < \mathrm{pat} \Rightarrow \mathrm{sel} > \mathrm{end}$

Scilla

Stateful Fragment

s ::=
  | x <- f
  | f := x
  | **let** x = e
  | **event** m
  | **send** ms

Scilla

Contract definition

Library of pure functions

Immutable contract parameters

Mutable fields

Transitions

Interaction via messages

```
scilla_version 0
library FungibleToken
let min_int : Uint128 → Uint128 → Uin          .. *)
let le_int  : Uint128 → Uint128 → Bool     = (* ... *)
let one_msg : Msg → List Msg = (* Return singleton List wi

contract FungibleToken
(owner : ByStr20, total_tokens : Uint128, decimals : Uint3  na          symbol : String)

field balances : Map ByStr20 Uint128 =
  let m = Emp ByStr20 Uint128 in builtin put m owner total_tokens
field allowed : Map ByStr20 (Map ByStr20 Uint128) = Emp ByStr20 (Map ByStr20 Uint128)

transition BalanceOf (tokenOwner : ByStr20)
  bal ← balances[tokenOwner];
  match bal with
  | Some v ⇒
    msg = {_tag : "BalanceOfResponse"; _recipient : _sender; address : tokenOwner; balance : v};
    msgs = one_msg msg; send msgs
  | None ⇒
    msg = {_tag : "BalanceOfResponse"; _recipient : _sender; address : tokenOwner; balance : zero};
    msgs = one_msg msg; send msgs
  end
end
transition TotalSupply () (* code omitted *) end
transition Transfer (to : ByStr20, tokens : Uint128) (* code omitted *) end
transition TransferFrom (from : ByStr20, to : ByStr20, tokens : Uint128) (* code omitted *) end
transition Approve (spender : ByStr20, tokens : Uint128) (* code omitted *) end
transition Allowance (tokenOwner : ByStr20, spender : ByStr20) (* code omitted *) end
```

Scilla

Monadic Interpreter (in OCaml)

Monadic
cost & failure
tracking

```
let rec exp_eval (e, loc) env = match e with
  let open EvalMonad.Let_syntax in
  | Literal l -> return (l, env)
  | Var i ->
      let%bind v = Env.lookup env i in return (v, env)
  | Let (i, _, lhs, rhs) ->
      let%bind lval, _ = wrap_eval lhs env (e, U) in
      let env' = Env.bind env (get_id i) lval in
      wrap_eval rhs env' (e, E lval)
  | GasExpr (g, e') ->
      let thunk () = exp_eval e' env in
      let%bind cost = eval_gas_charge env g in
      checkwrap thunk (Uint64.of_int cost)
        ("Insufficient gas")
  | Fixpoint (g, _, body) -> (* Other cases *)
```

The Rest of the Talk

- The Language
- The Testing Framework
- Generating Interesting Programs
- Semantic Harness for Testing
- Found Bugs

The Rest of the Talk

- The Language
- The Testing Framework
- Generating Interesting Programs
- Semantic Harness for Testing
- Found Bugs

Property-Based Testing

- Programmer writes *properties* of software system or component as a function from inputs to Booleans
- Tool *generates* many random inputs and applies the function to each one
- Famously embodied in **Haskell QuickCheck**

John Hughes

Koen Claessen

Testing Language Properties

```
Theorem preservation :
  forall Γ e τ e’,
    Γ |- e : τ -> e => e’ -> Γ |- e’ : τ.
```

Tricky part:
we only need
well-typed terms

Testing Language Properties

```
Theorem preservation :
  forall Γ e τ e’,
   Γ |- e : τ -> e => e’ -> Γ |- e’ : τ.
```

Non-Solution

1. Generate an *environment*, *term*, and *type*
2. Check if the term e is *well-typed*
3. If it’s not, start over (and again…)

Testing Language Properties

```
Theorem preservation :
  forall Γ e τ e’,
   Γ |- e : τ -> e => e’ -> Γ |- e’ : τ.
```

Solution

Write a generator that produces *well-typed terms!*

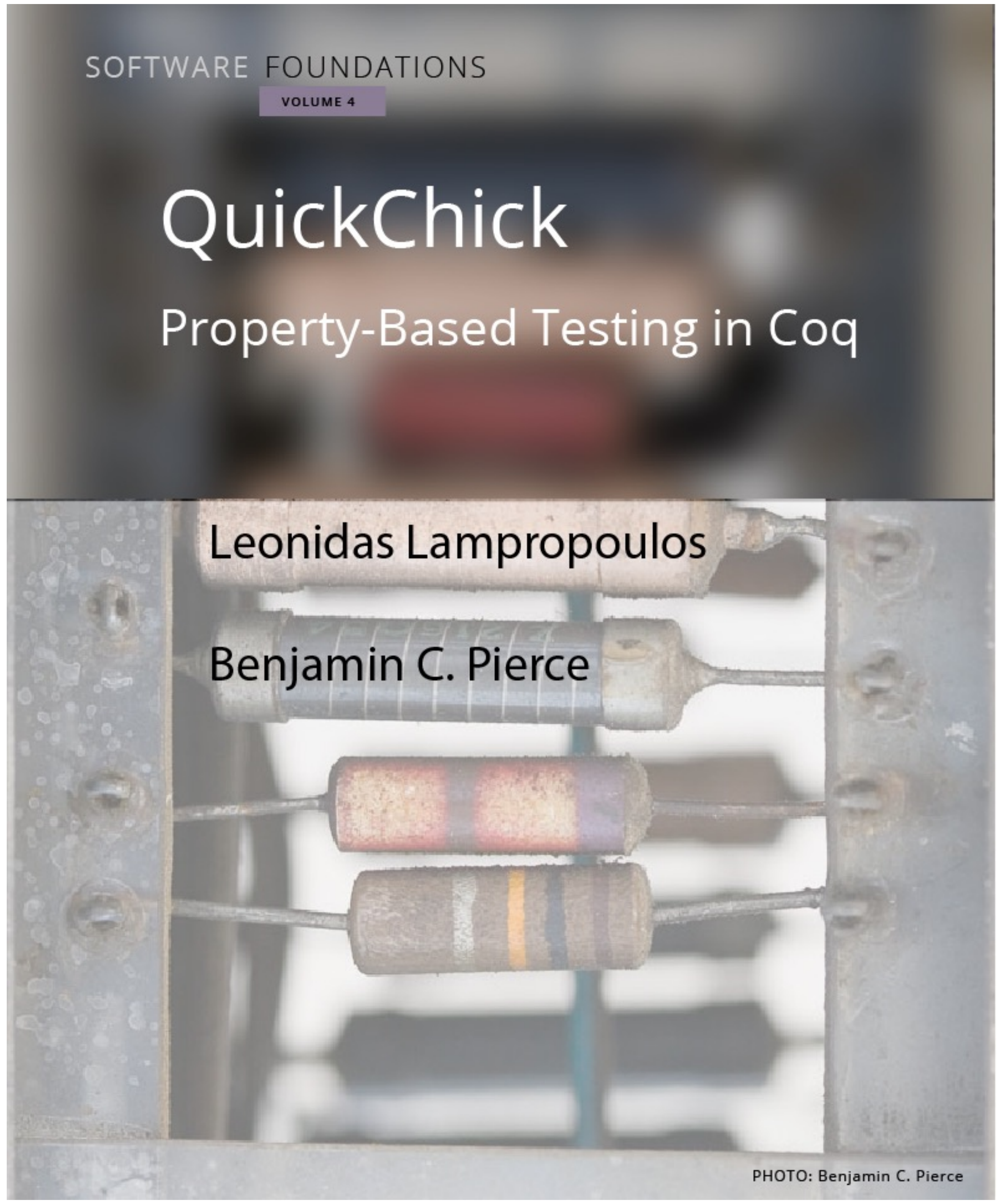
Testing Language Properties

Write a generator that produces *well-typed terms.*

This is a difficult and long-studied problem!

- CSmith [Yang et al., PLDI ‘11]
- Testing GHC’s strictness analyser [Palka et al., AST ‘11]
- Testing Noninterference, Quickly [Hritcu et al., ICFP ‘13]

A Tool for the Job: QuickChick



… used in practice to facilitate verification ([***JFP 2016***], DeepWeb, Vellvm)
… in many places
(UPenn, UMD, Princeton, MIT, INRIA)
… taught in courses and summer schools (UMD 631, DeepSpec Summer Schools)

A Tool for the Job: QuickChick

- Easy to define generators for hierarchical data (ASTs)
- Good integration with OCaml via Coq extraction
- Ability to do fuzzing-like, feedback-based generation

A Tool for the Job: QuickChick

- Easy to define generators for hierarchical data (ASTs)
- Good integration with OCaml via Coq extraction
- Ability to do fuzzing-like, feedback-based generation

  (ended up not using much, at least so far)

The Rest of the Talk

- The Language
- The Testing Framework
- Generating Interesting Programs
- Semantic Harness for Testing
- Found Bugs

The Rest of the Talk

- The Language
- The Testing Framework
- Generating Interesting Programs
- Semantic Harness for Testing
- Found Bugs

Scilla

Pure Fragment: System F

Variables: $x, y, \ldots, X, Y, \ldots$
Primitives: $p ::= Int32 \mid Int64 \mid \ldots \mid Uint32 \mid \ldots \mid$
$\mid String \mid ByStr \mid ByStrX \mid Message \mid \ \ldots$
Constants : $c ::= 0 \mid 1 \mid " \ " \mid \ \ldots$
Types: $\sigma, \tau \ldots ::= \boldsymbol{p} \mid \boldsymbol{\sigma \to \tau} \mid \boldsymbol{X} \mid \boldsymbol{\forall X. \tau}$
Terms: $e, \ldots ::= \boldsymbol{x} \mid \boldsymbol{c} \mid \boldsymbol{e_1 e_2} \mid \boldsymbol{\lambda x : \sigma. \mathrm{e}}$
$\mid \mathbf{e}\ \boldsymbol{\tau} \mid \boldsymbol{\Lambda}\mathbf{X}.\mathbf{e} \mid \{\mathrm{f} : \mathrm{e}, \ldots\}$
$\mid \mathrm{C}\ \mathrm{e}_1 \ldots \mathrm{e}_\mathrm{n} \mid \mathrm{match\ e\ with} < \mathrm{pat} \Rightarrow \mathrm{sel} > \mathrm{end}$

Generating System F Terms

$$\frac{\Gamma; X, \Delta \vdash e : \tau}{\Gamma; \Delta \vdash \Lambda X. e : \forall X. \tau}$$

$$\frac{\Gamma; \Delta \vdash e : \forall X. \tau}{\Gamma; \Delta \vdash e\ \tau' : \tau[\tau' / X]}$$

Generating System F Terms

$$\frac{\Gamma; X, \Delta \vdash e : \tau}{\Gamma; \Delta \vdash \Lambda X.\, e : \forall X.\, \tau}$$

Generating System F Terms

$$\frac{\Gamma; X, \Delta \vdash e : \tau}{\Gamma; \Delta \vdash \Lambda X.\, e : \forall X.\, \tau}$$

Generating System F Terms

$$\frac{\Gamma; X, \Delta \vdash e : \tau}{\Gamma; \Delta \vdash \Lambda X. e : \forall X. \tau}$$

Generating System F Terms

$$\frac{\Gamma; \Delta \vdash \mathrm{e} : \forall X.\, \tau}{\Gamma; \Delta \vdash \mathrm{e}\ \tau' : \tau[\tau' / X]}$$

Generating System F Terms

$$\frac{\Gamma; \Delta \vdash \mathrm{e} : \forall X.\,\tau \qquad \sigma = \tau[\tau' / X]}{\Gamma; \Delta \vdash \mathrm{e}\;\tau' : \sigma}$$

Generating System F Terms

How do we generate τ, τ′, and X such that this equality holds?

$$\frac{\Gamma;\Delta \vdash e : \forall X.\tau \qquad \sigma = \tau[\tau'/X]}{\Gamma;\Delta \vdash e\ \tau' : \sigma}$$

Un-substitution

Idea: Produce a distribution of closed "sub-types" τ' of σ to abstract!*

$$\frac{\Gamma; \Delta \vdash \mathrm{e} : \forall X.\, \tau \qquad \sigma = \tau[\tau' / X]}{\Gamma; \Delta \vdash \mathrm{e}\ \tau' : \sigma}$$

- Pick a closed sub-type τ' of σ
- Traverse σ and abstract τ' with X

*Details are a bit tricky (need keep track of closedness and frequencies): see the paper for the algorithm.

The Rest of the Talk

- The Language
- The Testing Framework
- Generating Interesting Programs
- Semantic Harness for Testing
- Bugs

The Rest of the Talk

- The Language
- The Testing Framework
- Generating Interesting Programs
- Semantic Harness for Testing
- Bugs

Testing Control- and Type-Flow Analysis in Scilla Compiler

- The analysis correctly over-approximates the *flow of values to variables*
  - use case: **function inlining**

- It also over-approximates the flow of *ground types to type variables*
  - use case: **monomorphization**

- To test over-approximation, we need **collecting semantics**

Monadic Interpreters to the Rescue

```
let rec exp_eval (e, loc) env = match e with
  let open EvalMonad.Let_syntax in
  | Literal l -> return (l, env)
  | Var i ->
      let%bind v = Env.lookup env i in return (v, env)
  | Let (i, _, lhs, rhs) ->
      let%bind lval, _ = wrap_eval lhs env (e, U) in
      let env' = Env.bind env (get_id i) lval in
      wrap_eval rhs env' (e, E lval)
  | GasExpr (g, e') ->
      let thunk () = exp_eval e' env in
      let%bind cost = eval_gas_charge env g in
      checkwrap thunk (Uint64.of_int cost)
        ("Insufficient gas")
  | Fixpoint (g, _, body) -> (* Other cases *)
```

PLDI'13

Monadic Abstract Interpreters

Ilya Sergey, IMDEA Software Institute, Spain, ilya.sergey@imdea.org
Dominique Devriese, iMinds – DistriNet, KU Leuven, Belgium, dominique.devriese@cs.kuleuven.be
Matthew Might, University of Utah, USA, might@cs.utah.edu
Jan Midtgaard, Aarhus University, Denmark, jmi@cs.au.dk
David Darais, Harvard University, USA, darais@seas.harvard.edu
Dave Clarke, Frank Piessens, iMinds – DistriNet, KU Leuven, Belgium, {firstname.lastname}@cs.kuleuven.be

ICFP'17

Abstracting Definitional Interpreters (Functional Pearl)

DAVID DARAIS, Univeristy of Maryland, USA
NICHOLAS LABICH, Univeristy of Maryland, USA
PHÚC C. NGUYỄN, Univeristy of Maryland, USA
DAVID VAN HORN, Univeristy of Maryland, USA

Implemented *State-Collecting Semantics* for **Flows-To Information** as a monad instance

The Rest of the Talk

- The Language
- The Testing Framework
- Generating Interesting Programs
- Semantic Harness for Testing
- Bugs

The Rest of the Talk

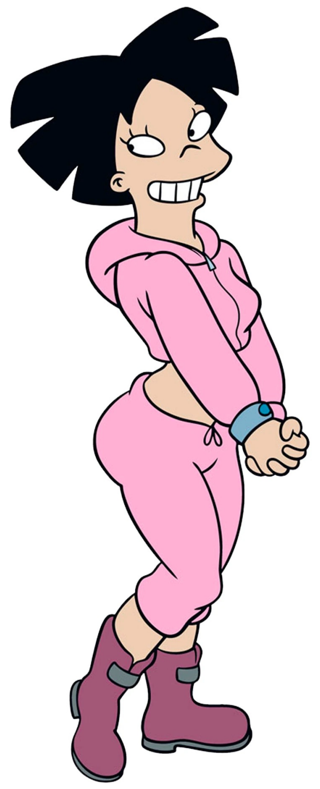
- The Language
- The Testing Framework
- Generating Interesting Programs
- Semantic Harness for Testing
- Bugs

Bugs Found!

ID	Short bug description	Status
	Type checking and type inference	
#1	Closure values could be used as map keys	known
#2	Type variables were not properly shadowed; the bug allowed for encoding non well-formed recursion	known
#3	Type checker allowed for hashing closure values	new
#4	Type checker allowed for hashing polymorphically-typed values	new
#5	Sub-types of address type `ByteString` were implicitly up-cast to type `ByteString`	new
	Definitional interpreter	
#6	Conversion between `bech32` and `ByStr20` datatypes threw an exception	new
#7	Cryptographic built-in operations `ecdsa_verify` and `ecdsa_recover_pk` were throwing exceptions	new
#8	Cryptographic built-in `ecdsa_recover_pk` could abort Scilla interpreter with an OS-level exception	new
#9	The interpreter inadequately charged gas for the `power` arithmetic operation	new
	Type-flow analysis	
#10	Type-flow analysis does not terminate on programs that make use of impredicative polymorphism	known

Bugs Found!

ID	Short bug description	Status
	Type checking and type inference	
#1	Closure values could be used as map keys	known
#2	Type variables were not properly shadowed; the bug allowed for encoding non well-formed recursion	known
#3	Type checker allowed for hashing closure values	new
#4	Type checker allowed for hashing polymorphically-typed values	new
#5	Sub-types of address type `ByteString` were implicitly up-cast to type `ByteString`	new
	Definitional interpreter	
#6	Conversion between `bech32` and `ByStr20` datatypes threw an exception	new
#7	Cryptographic built-in operations `ecdsa_verify` and `ecdsa_recover_pk` were throwing exceptions	new
#8	Cryptographic built-in `ecdsa_recover_pk` could abort Scilla interpreter with an OS-level exception	new
#9	The interpreter inadequately charged gas for the `power` arithmetic operation	new
	Type-flow analysis	
#10	Type-flow analysis does not terminate on programs that make use of impredicative polymorphism	known



Bugs Found!

ID	Short bug description	Status
	Type checking and type inference	
#1	Closure values could be used as map keys	known
#2	Type variables were not properly shadowed; the bug allowed for encoding non well-formed recursion	known
#3	Type checker allowed for hashing closure values	new
#4	Type checker allowed for hashing polymorphically-typed values	new
#5	Sub-types of address type `ByteString` were implicitly up-cast to type `ByteString`	new
	Definitional interpreter	
#6	Conversion between `bech32` and `ByStr20` datatypes threw an exception	new
#7	Cryptographic built-in operations `ecdsa_verify` and `ecdsa_recover_pk` were throwing exceptions	new
#8	Cryptographic built-in `ecdsa_recover_pk` could abort Scilla interpreter with an OS-level exception	new
#9	The interpreter inadequately charged gas for the power arithmetic operation	new
	Type-flow analysis	
#10	Type-flow analysis does not terminate on programs that make use of impredicative polymorphism	known

Bugs Found!

ID	Short bug description	Status
	Type checking and type inference	
#1	Closure values could be used as map keys	known
#2	Type variables were not properly shadowed; the bug allowed for encoding non well-formed recursion	known
#3	Type checker allowed for hashing closure values	new
#4	Type checker allowed for hashing polymorphically-typed values	new
#5	Sub-types of address type `ByteString` were implicitly up-cast to type `ByteString`	new
	Definitional interpreter	
#6	Conversion between `bech32` and `ByStr20` datatypes threw an exception	new
#7	Cryptographic built-in operations `ecdsa_verify` and `ecdsa_recover_pk` were throwing exceptions	new
#8	Cryptographic built-in `ecdsa_recover_pk` could abort Scilla interpreter with an OS-level exception	new
#9	The interpreter inadequately charged gas for the `power` arithmetic operation	new
	Type-flow analysis	
#10	Type-flow analysis does not terminate on programs that make use of impredicative polymorphism	known

Type Variable Shadowing Bug (#2)

```
let a =
  let b =
    let c =
      let d = tfun 'V ⇒ fun (v1 : 'V) ⇒ tfun 'V ⇒ v1
      in @d (ByStr32)
    in
    let e = 0x0000000000000000000000000000000000000000000000000000000000000002
    in c e
  in @b (forall 'V. Nat)
in @a Nat
```

: ByStr32

This program should be ill-typed!

To Take Away

Artifacts Available acm V1.1

Artifacts Evaluated acm Reusable V1.1

Random Testing of a Higher-Order Blockchain Language (Experience Report)

122

TRAM HOANG, National University of Singapore, Singapore
ANTON TRUNOV, Zilliqa Research, Russia
LEONIDAS LAMPROPOULOS, University of Maryland, USA
ILYA SERGEY, National University of Singapore, Singapore

We describe our experience of using property-based testing—an approach for automatically generating random inputs to check executable program specifications—in a development of a higher-order smart contract language that powers a state-of-the-art blockchain with thousands of active daily users.

We outline the process of integrating QuickChick—a framework for property-based testing built on top of the Coq proof assistant—into a real-world language implementation in OCaml. We discuss the challenges we have encountered when generating well-typed programs for a realistic higher-order smart contract language, which mixes purely functional and imperative computations and features runtime resource accounting. We describe the set of the language implementation properties that we tested, as well as the semantic harness required to enable their validation. The properties range from the standard type safety to the soundness of a control- and type-flow analysis used by the optimizing compiler. Finally, we present the list of bugs discovered and rediscovered with the help of QuickChick and discuss their severity and possible ramifications.

- We've tested the *language layer* (based on System F) of a real-world blockchain with **QuickChick** and found several critical bugs.
- We've introduced *un-substitution:* a simple technique to generate *well-typed* System F terms.
- We've used *monadic interpreters* methodology of implementing *collecting semantics*.
- Check out our *artifact* for the QuickChick test harness and examples!

QuickChick

SCILLA

Thanks!