+ DON'T TRUST, FORMALLY VERIFY

Jean-Frédéric Etienne, IOG
Romain Soulat, IOG

Live Demo

Simple Vesting Example

```
-- Vesting validator functions
def validScriptContext : POSIXTime -> ScriptContext -> Bool :=
    fun time sc =>
        (sc.transaction.validity_range.lower_bound <= time.time && time.time <= sc.transaction.validity_range.upper_bound)
        && (sc.transaction.validity_range.lower_bound <= sc.transaction.validity_range.upper_bound)

def signed_by : List VerificationKeyHash ->  VerificationKeyHash -> Bool :=
    fun keys key =>
            keys.contains key

def time_elapsed : ValidityRange -> POSIXTime -> Bool :=
    fun range time =>
        range.upper_bound >= time.time

def validator : VestingDatum -> VestingRedeemer
    fun  datum _ sc =>
        let transaction := sc.transaction;
        let purpose := sc.purpose;
        let signatories := transaction.signatori
        let v_range := transaction.validity_rang
        (purpose == Purpose.Spending) && (signec

theorem only_accept_if_signatory_and_time_elapse
        ∀ (datum: VestingDatum) (redeemer: Vesti

            ((validator datum redeemer c)
            &&
            (validScriptContext time c))

            →

            (c.transaction.signatories.contains datum.beneficiary
            &&
            time.time ≥ datum.lock_until.time) :=

            by sorry
```

```
30
31   theorem only_accept_if_signatory_and_time_elapsed :   ⊗ CEX Found
```

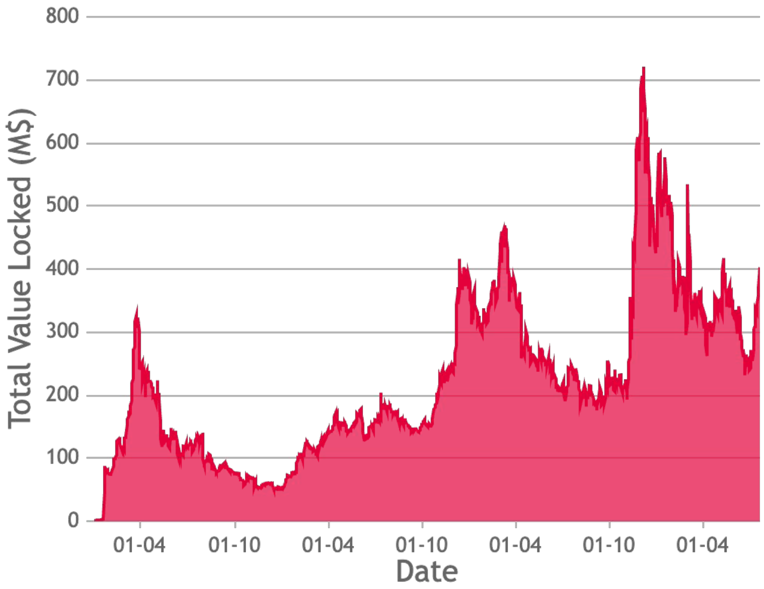
CEX Found only_accept_if_signatory_and_time_elapsed

```
CEX Found
datum: (DemoRareEvo.Vesting.VestingTypes.VestingDatum.mk  (DemoRareEvo.Vesting.PlutusLedgerApi.POSIXTime.mk 21239)  (DemoRareEvo.Vesting.PlutusLedger

redeemer: (DemoRareEvo.Vesting.VestingTypes.VestingRedeemer.mk 7719)

c: (DemoRareEvo.Vesting.PlutusLedgerApi.ScriptContext.mk  DemoRareEvo.Vesting.PlutusLedgerApi.Purpose.Spending  (DemoRareEvo.Vesting.PlutusLedgerApi.

time: (DemoRareEvo.Vesting.PlutusLedgerApi.POSIXTime.mk 21238)
```

+ Why Formal Verification?

Strong guarantees for critical software

Why Smart Contracts Need Strong Guarantees



Current verification approaches:

- Unit tests
- Integration tests
- Property Based Testing
- Manual audits

Very hard and expensive to test all scenarios, all possible values, …

Formal verification is the gold standard in many other industries:
Railway (SIL4), Aerospace (DAL-A), Chips, Cybersecurity (EAL7+)

Existing Approaches: Powerful But Specialized

Agda2hs
Deep mathematical proofs
Dedicated model
Requires a strong expertise in Agda
Manual proof

hs-to-coq
Deep mathematical proofs
Dedicated model
Requires a strong expertise in Coq/Rocq
Manual proof

LiquidHaskell
At source code level
Specific property types
Need to specify each function used
Scalability issues

SBV
(Almost) at the source code level
Automated reasoning
Provide tests for paths
Path explosion issue

+ Our vision

Write Specs. Push Button. Get Proofs.

Write Specs. Push Button. Get Proofs.

```
{-@ uniqueNFTToken:
   ∀ (p : OracleParams) (ocHash: ScriptHash) (currSym: CurrencySymbol),
     let hasNFTToken := fun utxo => TxOut.hasValue? utxo oracleNFToken currSym > 0;
     let validScriptHash := fun utxo => TxOut.scriptHash? utxo ocHash;
      ValidOracleParams p →
      State.Validators.hasScriptHash? (Validator.oracleContract p) ocHash →
      State.MintingPolicies.hasCurrencySymbol? (Minting.oracleMintingContract p ocHash) currSym →
      State.TxOuts.any hasNFTToken →
      State.TxOuts.sumOf (fun utxo => hashNFTToken utxo && validScriptHash utxo) = 1
```

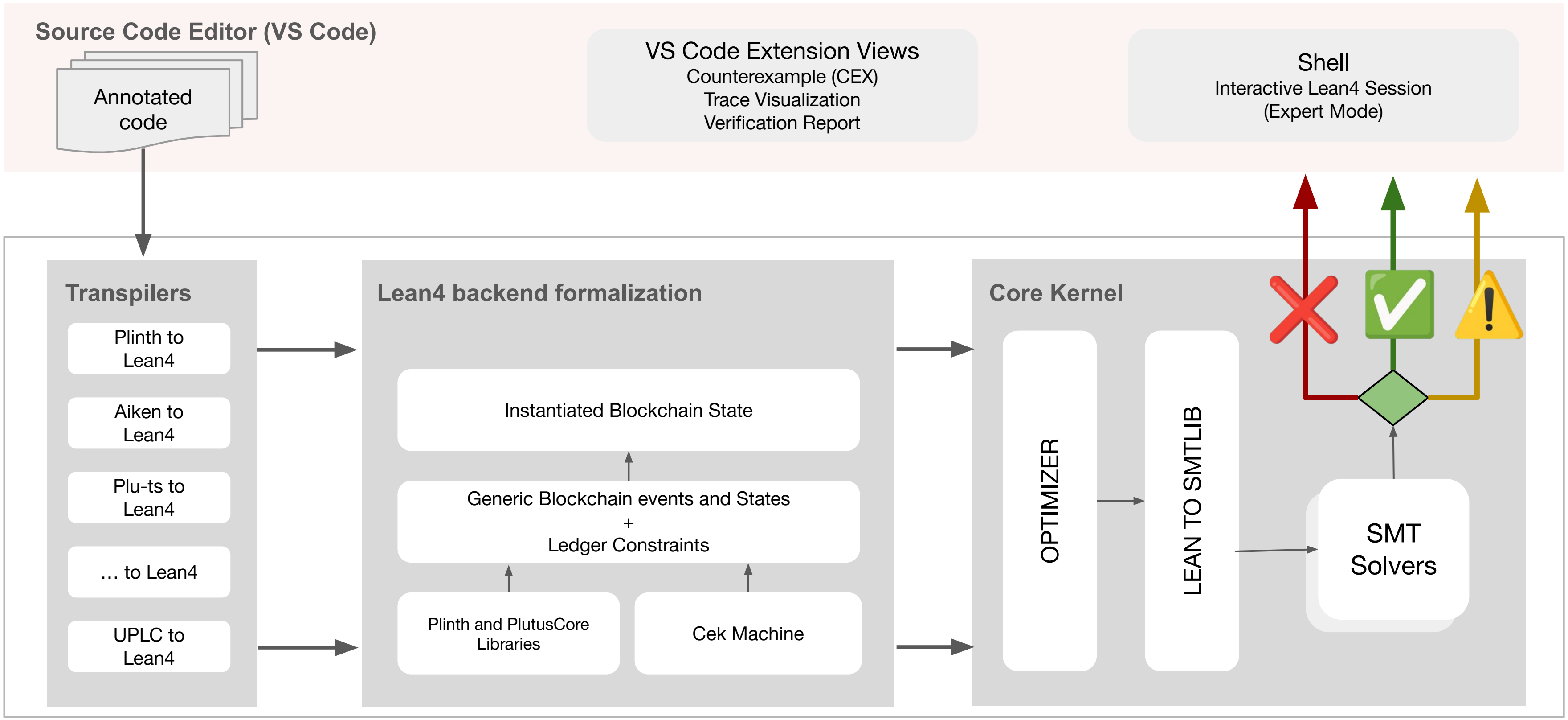
```
{-# INLINABLE validate #-}
validate :: EscrowParams DatumHash -> PaymentPubKeyHash -> Action -
validate EscrowParams{escrowDeadline, escrowTargets} contributor ac
    case action of
        Redeem ->
            traceIfFalse "escrowDeadline-after" (escrowDeadline `af
            && traceIfFalse "meetsTarget" (all (meetsTarget script(
        Refund ->
            traceIfFalse "escrowDeadline-before" ((escrowDeadline -
            && traceIfFalse "txSignedBy" (scriptContextTxInfo `txSi

typedValidator :: EscrowParams Datum -> V2.TypedValidator Escrow
typedValidator escrow = go (Haskell.fmap datumHash escrow) where
    go = V2.mkTypedValidatorParam @Escrow
        $$(PlutusTx.compile [|| validate ||])
        $$(PlutusTx.compile [|| wrap ||])
    wrap = Scripts.mkUntypedValidator
```

```
$ afv ./myContract.hs
```

Valid ✅

Falsified ❌

Behind Push-Button Formal Verification



+ Annotation Language

One Language to Spec' Them All

- Useful for all the stack of verification needs
  - Global blockchain state properties
  - Transaction related properties
  - Temporal properties
  - Simple properties for custom helper functions
- Applicable for all Cardano Smart Contracts (Plinth, Aiken, …)

Packaged in a set of libraries to allow easy property expression.

+ Cardano Blockchain Formalisation

From a generic solver to a specialized tool

Plinth and PlutusLedgerAPI

- Ease of transpilation of Plinth to Lean4 transpiler
- Introduction of the builtins for the Blockchain state formalization
- Facilitate the generation of correctness proof obligations for Typeclass instances
- Plinth-based smart contract can directly be specified in Lean4

Almost a 1 to 1 mapping...

```haskell
class (Eq a) => Ord a where
  compare :: a -> a -> Ordering
  (<), (<=), (>), (>=) :: a -> a -> Bool
  max, min :: a -> a -> a

  {-# INLINEABLE compare #-}
  compare x y =
    if x == y
      then EQ
      else
        if x <= y
          then LT
          else GT

  {-# INLINEABLE (<) #-}
  x < y = case compare x y of LT -> True; _ -> False
  {-# INLINEABLE (<=) #-}
  x <= y = case compare x y of GT -> False; _ ->
True{-# INLINEABLE (>) #-}
  x > y = case compare x y of GT -> True; _ -> False
  {-# INLINEABLE (>=) #-}
  x >= y = case compare x y of LT -> False; _ ->
True
  {-# INLINEABLE max #-}
  max x y = if x <= y then y else x
  {-# INLINEABLE min #-}
  min x y = if x <= y then x else y
  {-# MINIMAL compare | (<=) #-}
```

```lean
class Ord' (a : Type) extends Eq a where
  leq : a -> a -> Bool
  compare (x : a) (y : a) : Ordering :=
    if x == y then Ordering.EQ
    else if leq x y then Ordering.LT
    else Ordering.GT

class Ord (a : Type) extends Ord' a where

  lt (x : a) (y : a) : Bool :=
    match compare x y with
    | Ordering.LT => true
    | _ => false

  leq x y :=
    match compare x y with
    | Ordering.GT => false
    | _ => true

  gt (x : a) (y : a) : Bool :=
    match compare x y with
    | Ordering.GT => true
    | _ => false

  geq (x : a) (y : a) : Bool :=
    match compare x y with
    | Ordering.LT => false
    | _ => true

  max (x : a) (y : a) : a := if leq x y then y else x

  min (x : a) (y : a) : a := if leq x y then x else y
```

.. but with a lot more verification

```
/-- Properties on 'leq' that need to be provided for each Ord instance -/
eq_leq_left : ∀ (x y : α), x ==ₚ y → leq x y
eq_leq_right : ∀ (x y : α), x ==ₚ y → leq y x
eq_geq_left : ∀ (x y : α), x ==ₚ y → geq x y
eq_geq_right : ∀ (x y : α), x ==ₚ y → geq y x
leq_reflexive : ∀ (x : α), leq x x
leq_antisymmetric : ∀ (x y : α), leq x y → leq y x → x ==ₚ y
leq_transitive : ∀ (x y z : α), leq x y → leq y z → leq x z
leq_imp_eq_or_lt : ∀ (x y : α), leq x y → ( x ==ₚ y ||ₚ lt x y )
leq_geq_iff : ∀ (x y : α), leq x y = geq y x
leq_not_lt_iff : ∀ (x y : α), notₚ (lt y x) = leq x y
leq_not_gt_iff : ∀ (x y : α), notₚ (gt x y) = leq x y

/-- Properties on 'lt' that need to be provided for each Ord instance -/
eq_imp_not_lt : ∀ (x y : α), x ==ₚ y → ( notₚ (lt x y) &&ₚ notₚ (lt y x) )
lt_not_reflexive : ∀ (x : α), notₚ (lt x x)
lt_antisymmetric : ∀ (x y : α), lt x y → notₚ (lt y x)
lt_transitive : ∀ (x y z : α), lt x y → lt y z -> lt x z
lt_imp_leq : ∀ (x y : α), lt x y → leq x y
lt_gt_iff : ∀ (x y : α), lt x y = gt y x
lt_imp_not_gt : ∀ (x y : α), lt x y → notₚ (gt x y)
lt_imp_not_eq : ∀ (x y : α), lt x y → x /=ₚ y
lt_not_leq_iff : ∀ (x y : α), notₚ (leq y x) = lt x y
lt_not_geq_iff : ∀ (x y : α), notₚ (geq x y) = lt x y

/-- Properties on 'geq' that need to be provided for each Ord instance -/
geq_reflexive : ∀ (x : α), geq x x
geq_antisymmetric : ∀ (x y : α), geq x y → geq y x → x ==ₚ y
geq_transitive : ∀ (x y z : α), geq x y → geq y z → geq x z
geq_imp_eq_or_gt : ∀ (x y : α), geq x y → ( x ==ₚ y ||ₚ gt x y )
geq_not_gt_iff : ∀ (x y : α), notₚ (gt y x) = geq x y
geq_not_lt_iff : ∀ (x y : α), notₚ (lt x y) = geq x y
geq_and_leq_imp_eq : ∀ (x y : α), geq x y → leq x y → x ==ₚ y

/-- Properties on 'gt' that need to be provided for each Ord instance -/
eq_imp_not_gt : ∀ (x y : α), x ==ₚ y → ( notₚ (gt x y) &&ₚ notₚ (gt y x) )
gt_not_reflexive : ∀ (x : α), notₚ (gt x x)
gt_antisymmetric : ∀ (x y : α), gt x y → notₚ (gt y x)
gt_transitive : ∀ (x y z : α), gt x y → gt y z -> gt x z
gt_imp_geq : ∀ (x y : α), gt x y → geq x y
gt_imp_not_lt : ∀ (x y : α), gt x y → notₚ (lt x y)
gt_imp_not_eq : ∀ (x y : α), gt x y → x /=ₚ y
gt_not_leq_iff : ∀ (x y : α), notₚ (leq x y) = gt x y
gt_not_geq_iff : ∀ (x y : α), notₚ (geq y x) = gt x y
```

```
/-- Propreties on 'min' that need to be provided for each Ord instance -/
min_reduce : ∀ (x : α), min x x = x := by simp
leq_min_left : ∀ (x y : α), leq x y → min x y = x := by simp; intros; contradiction
leq_min_right : ∀ (x y : α), leq y x → min x y = y
lt_min_left : ∀ (x y : α), lt x y → min x y = x
lt_min_right : ∀ (x y : α), lt y x → min x y = y
geq_min_left : ∀ (x y : α), geq y x → min x y = x := by simp; intros; contradiction
geq_min_right : ∀ (x y : α), geq x y → min x y = y
gt_min_left : ∀ (x y : α), gt y x → min x y = x
gt_min_right : ∀ (x y : α), gt x y → min x y = y

/-- Propreties on 'max' that need to be provided for each Ord instance -/
max_reduce : ∀ (x : α), max x x = x := by simp
leq_max_left : ∀ (x y : α), leq y x → max x y = x
leq_max_right : ∀ (x y : α), leq x y → max x y = y := by simp; intros; contradiction
lt_max_left : ∀ (x y : α), lt y x → max x y = x
lt_max_right : ∀ (x y : α), lt x y → max x y = y
geq_max_left : ∀ (x y : α), geq x y → max x y = x
geq_max_right : ∀ (x y : α), geq y x → max x y = y := by simp; intros; contradiction
gt_max_left : ∀ (x y : α), gt x y → max x y = x
gt_max_right : ∀ (x y : α), gt y x → max x y = y

/-- Propreties on 'compare' that need to be provided for each Ord instance -/
compare_eq_left : ∀ (x y : α), x ==ₚ y → compare x y = Ordering.EQ
compare_eq_right : ∀ (x y : α), x ==ₚ y → compare y x = Ordering.EQ
compare_imp_eq : ∀ (x y : α), compare x y = Ordering.EQ → x ==ₚ y
compare_lt_left : ∀ (x y : α), lt x y → compare x y = Ordering.LT
compare_lt_right : ∀ (x y : α), lt y x → compare x y = Ordering.GT
compare_imp_lt : ∀ (x y : α), compare x y = Ordering.LT → lt x y
compare_leq_left : ∀ (x y : α), leq x y → ( compare x y = Ordering.EQ ∨ compare x y = Ordering.LT )
compare_leq_right : ∀ (x y : α), leq y x → ( compare x y = Ordering.EQ ∨ compare x y = Ordering.GT )
compare_imp_leq : ∀ (x y : α), ( compare x y = Ordering.EQ ∨ compare x y = Ordering.LT ) → leq x y
compare_leq_neq : ∀ (x y : α), leq x y → ¬ (compare x y = Ordering.EQ) -> lt x y
compare_leq_eq : ∀ (x y : α), leq x y → (compare x y = Ordering.EQ) -> x ==ₚ y
compare_gt_left : ∀ (x y : α), gt x y → compare x y = Ordering.GT
compare_gt_right : ∀ (x y : α), gt y x → compare x y = Ordering.LT
compare_imp_gt : ∀ (x y : α), compare x y = Ordering.GT → gt x y
compare_not_lt_imp_geq : ∀ (x y : α), compare x y ≠ Ordering.LT → geq x y
compare_not_gt_imp_leq : ∀ (x y : α), compare x y ≠ Ordering.GT → leq x y
compare_geq_left : ∀ (x y : α), geq x y → ( compare x y = Ordering.EQ ∨ compare x y = Ordering.GT )
compare_geq_right : ∀ (x y : α), geq y x → ( compare x y = Ordering.EQ ∨ compare x y = Ordering.LT )
compare_imp_geq : ∀ (x y : α), ( compare x y = Ordering.EQ ∨ compare x y = Ordering.GT ) → geq x y
compare_geq_neq : ∀ (x y : α), geq x y → ¬ (compare x y = Ordering.EQ) -> gt x y
compare_geq_eq : ∀ (x y : α), geq x y → (compare x y = Ordering.EQ) -> x ==ₚ y
compare_equality_imp_eq : ∀ (x y : α), compare x y = compare y x → x ==ₚ y
compare_eq_imp_not_gt : ∀ (x y : α), compare x y = Ordering.EQ → ¬ (compare x y = Ordering.GT)
compare_eq_imp_not_lt : ∀ (x y : α), compare x y = Ordering.EQ → ¬ (compare x y = Ordering.LT)
compare_lt_imp_not_eq : ∀ (x y : α), compare x y = Ordering.LT → ¬ (compare x y = Ordering.EQ)
compare_lt_imp_not_gt : ∀ (x y : α), compare x y = Ordering.LT → ¬ (compare x y = Ordering.GT)
compare_lt_imp_not_lt : ∀ (x y : α), compare x y = Ordering.LT → ¬ (compare y x = Ordering.LT)
compare_lt_imp_gt : ∀ (x y : α), compare x y = Ordering.LT → compare y x = Ordering.GT
compare_gt_imp_not_eq : ∀ (x y : α), compare x y = Ordering.GT → ¬ (compare x y = Ordering.EQ)
compare_gt_imp_not_lt : ∀ (x y : α), compare x y = Ordering.GT → ¬ (compare x y = Ordering.LT)
compare_gt_imp_not_gt : ∀ (x y : α), compare x y = Ordering.GT → ¬ (compare y x = Ordering.GT)
compare_gt_imp_lt : ∀ (x y : α), compare x y = Ordering.GT → compare y x = Ordering.LT
compare_refl_eq : ∀ (x : α), compare x x = Ordering.EQ
compare_refl_not_gt : ∀ (x : α), compare x x ≠ Ordering.GT
compare_refl_not_lt : ∀ (x : α), compare x x ≠ Ordering.LT
compare_antisymmetric_lt_lt : ∀ (x y : α), compare x y ≠ Ordering.LT → compare y x ≠ Ordering.LT → x ==
```


PlutusCore with formal verification

```lean
inductive Data where
  | Constr : Integer → List Data → Data
  | Map : List (Data × Data) → Data
  | List : List Data → Data
  | I : Integer → Data
  | B : ByteString → Data

mutual
  private def dataStr : Data → String
    | .Constr idx fields => constrStr idx fields
    | .Map mxs => mapStr "" mxs
    | .List xs => listDataStr "" xs
    | .I i => s!"(I {i})"
    | .B bs => s!"(B {bs})"

  private def constrStr : Integer → List Data → String
   | idx, fields =>  s!"(Constr {idx} [{listDataStr ""
fields}])"
  private def listDataStr (acc : String) : List Data → String
    | [] => s!"(List [{acc}])"
    | h :: tl =>
        let hStr := dataStr h
        if acc.isEmpty
        then listDataStr hStr tl
        else listDataStr s!"{acc}, {hStr}" tl

  private def mapStr (acc : String) : List (Data × Data) →
String|[] => s!"(Map [{acc}])"
    | (x, y) :: tl =>
       let hstr := s!"({dataStr x}, {dataStr y})"
       if acc.isEmpty
       then mapStr hstr tl
       else mapStr s!"{acc}, {hstr}" tl
end
```

```lean
@[simp] theorem Data.beq_iff_eq (x y : Data) : x == y ↔ x = y := by
  simp [BEq.beq]
  apply Iff.intro
  . apply eqData_true_imp_eq
  . intro h
    rw [h]
    apply eqData_reflexive

@[simp] theorem Data.not_beq_iff_not_eq (x y : Data) : x != y ↔ x ≠ y := by simp [BEq.beq]

@[simp] theorem chooseData_constr
  (idx : Integer) (xs : List Data) (tc : α) (tm : α) (tl : α) (ti : α) (tb : α) :
  UPLC.chooseData (Data.Constr idx xs) tc tm tl ti tb = tc := rfl

@[simp] theorem chooseData_map
  (xs : List (Data × Data)) (tc : α) (tm : α) (tl : α) (ti : α) (tb : α) :
  UPLC.chooseData (Data.Map xs) tc tm tl ti tb = tm := rfl

@[simp] theorem chooseData_list
  (xs : List Data) (tc : α) (tm : α) (tl : α) (ti : α) (tb : α) :
  UPLC.chooseData (Data.List xs) tc tm tl ti tb = tl := rfl

@[simp] theorem chooseData_i
  (i : Integer) (tc : α) (tm : α) (tl : α) (ti : α) (tb : α) :
  UPLC.chooseData (Data.I i) tc tm tl ti tb = ti := rfl

@[simp] theorem chooseData_b
  (bs : ByteString) (tc : α) (tm : α) (tl : α) (ti : α) (tb : α) :
  UPLC.chooseData (Data.B bs) tc tm tl ti tb = tb := rfl
```

CEK Machine reimplemented

```
def step (Sigma : State) : State :=
  match Sigma with
  |                                 s; ρ ▷ u(var x)              => s ◁ ρ⟦x⟧ If x is bound in ρ
  |                                 s; ρ ▷ u(con T c)            => s ◁ v⟨con T c⟩
  |                                 s; ρ ▷ u(lam x, M)           => s ◁ v⟨lam x, M, ρ⟩
  |                                 s; ρ ▷ u(delay M)            => s ◁ v⟨delay M, ρ⟩
  |                                 s; ρ ▷ u(force M)            =>  (@f(force ␣) · s); ρ ▷ M
  |                                 s; ρ ▷ u[M ∘_ N]             => (@f[␣ (N, ρ)] · s); ρ ▷ M
  |                                 s; ρ ▷ u(constr i (M · Ms))  => (@f(constr i, [] ␣ (Ms, ρ)) · s); ρ ▷ M
  |                                 s; ρ ▷ u(constr i [])        => s ◁ v⟨constr i, []⟩
  |                                 s; ρ ▷ u(case N, Ms)         => (@f(case ␣ (Ms, ρ)) · s); ρ ▷ N
  |                                 s; ρ ▷ u(builtin b)          => s ◁ v⟨builtin b, [], α(b)⟩
  |                                 s; ρ ▷ u(error)              => ◆
  |                                   [] ◁ V                     => ▢ V
  |                    (@f[␣ (M, ρ)] · s) ◁ V                     => (@f[V ␣] · s); ρ ▷ M
  |             (@f[v⟨lam x, M, ρ⟩ ␣] · s) ◁ V                     => s; ρ⟦x ↦ V⟧ ▷ M
  |                         (@f[␣ V] · s) ◁ v⟨lam x, M, ρ⟩        => s; ρ⟦x ↦ V⟧ ▷ M
  |  (@f[v⟨builtin b, Vs, ι ⊙ η⟩ ␣] · s) ◁ V                     => (s ◁ v⟨builtin b, Vs :· V, η⟩) If ι ∈ 𝓤 ∪ 𝓥
  |                         (@f[␣ V] · s) ◁ v⟨builtin b, Vs, ι ⊙ η⟩ => (s ◁ v⟨builtin b, Vs :· V, η⟩) If ι ∈ 𝓤 ∪ 𝓥
  |    (@f[v⟨builtin b, Vs, a[ι]⟩ ␣] · s) ◁ V                     => (Eval_CEK(s, b, Vs :· V)) If ι ∈ 𝓤 ∪ 𝓥
  |                         (@f[␣ V] · s) ◁ v⟨builtin b, Vs, a[ι]⟩  => (Eval_CEK(s, b, Vs :· V)) If ι ∈ 𝓤 ∪ 𝓥
  |                     (@f(force ␣) · s) ◁ v⟨delay M, ρ⟩          => s; ρ ▷ M
  |                     (@f(force ␣) · s) ◁ v⟨builtin b, Vs, ι ⊙ η⟩ => (s ◁ v⟨builtin b, Vs, η⟩) If ι ∈ 𝓠
  |                     (@f(force ␣) · s) ◁ v⟨builtin b, Vs, a[ι]⟩  => (Eval_CEK(s, b, Vs)) If ι ∈ 𝓠
  | (@f(constr i, Vs ␣ (M · Ms, ρ)) · s) ◁ V                     => (@f(constr i, Vs :· V ␣ (Ms, ρ)) · s); ρ ▷ M
  |     (@f(constr i, Vs ␣ ([], ρ)) · s) ◁ V                     => s ◁ v⟨constr i, Vs :· V⟩
  |            (@f(case ␣ (Ms, ρ)) · s) ◁ v⟨constr i, Vs⟩         => unfoldCase s i Ms Vs ρ
  | _ => ◆
```

+ Optimization, Normalization, SMT-translation

Cornerstone for scalability

Internal representation gets big, too big if not managed carefully

- Need to reduce internal representation complexity before querying the SMT solver
- Minimizes (or even removes!) user intervention and the need for manual proof
- Speeds up proof and dramatically improves scalability

Cornerstone for scalability

Key normalizations

- Aggressive constant propagation
- Arithmetic, boolean, and propositional simplification
- If-then-else simplification
- Match and recursive function equivalence detection
- Beta reduction and non-recursive function/lambda applications
- Structural equivalence on expressions
- Cone of influence computation and variable elimination

+ Sending everything to Z3

Efficient encoding to SMTLib

- Inductive data types (including mutually inductive)
- Recursive function (including mutually recursive)
- Inductive proof schemas
- Quantified functions, higher-order functions, lambda terms
- Counterexample generation support for recursive types/functions

Moving from Lean4 to SMTLib for automated reasoning

Simple translation example

```
#solve (dump-smt-lib: 1) (only-smt-lib: 1) [∀ (a b c : Nat), (a + b) * c = c * a + b * c]
```

⇓

```
(define-sort Nat () Int)
(define-fun @isNat ((@x Nat)) Bool (<= 0 @x))
(declare-const $0 Nat)
(declare-const $1 Nat)
(declare-const $2 Nat)
(assert (not (=> (@isNat $0)
                 (=> (@isNat $1)
                     (=> (@isNat $2)
                         (= (+ (* $0 $2) (* $1 $2)) (* $2 (+ $0
$1))))))))
```

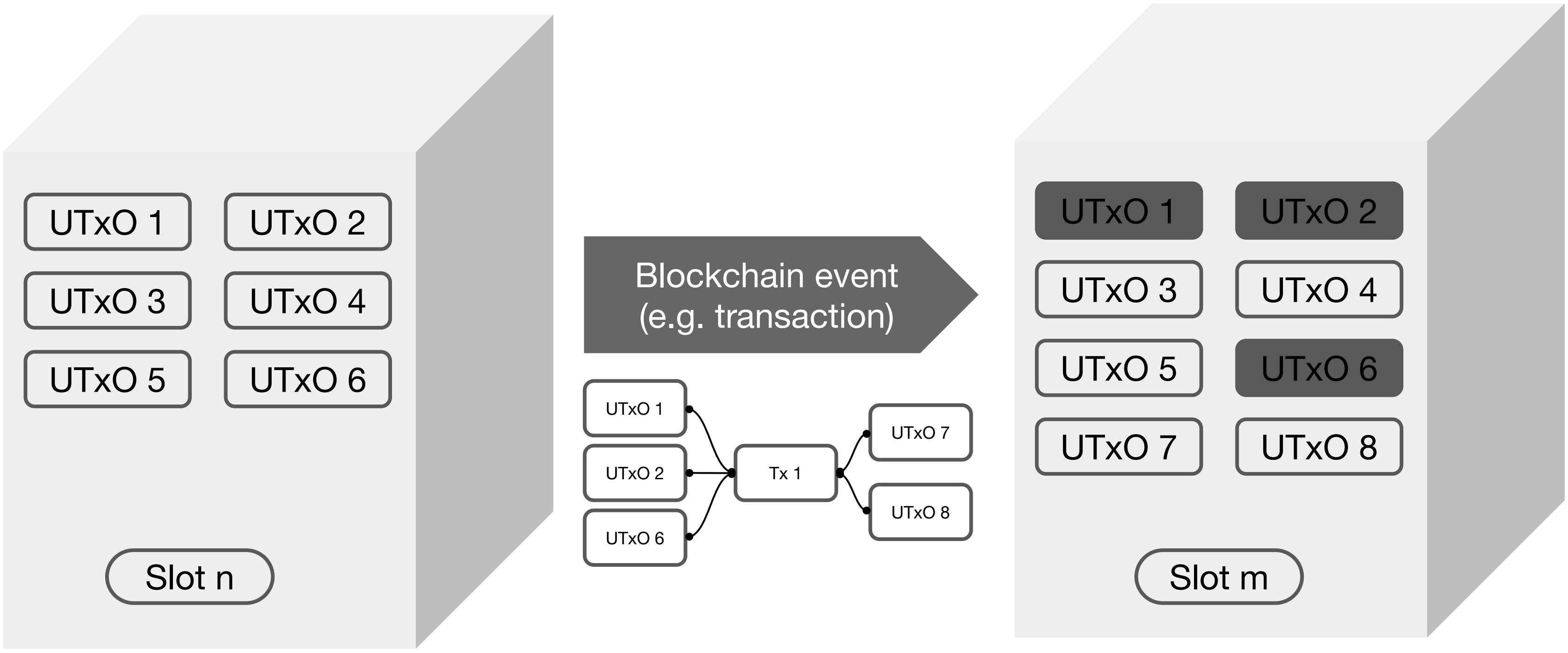
+ Live Demo: simpleAdd UPLC

Optimizing away the CEK machine

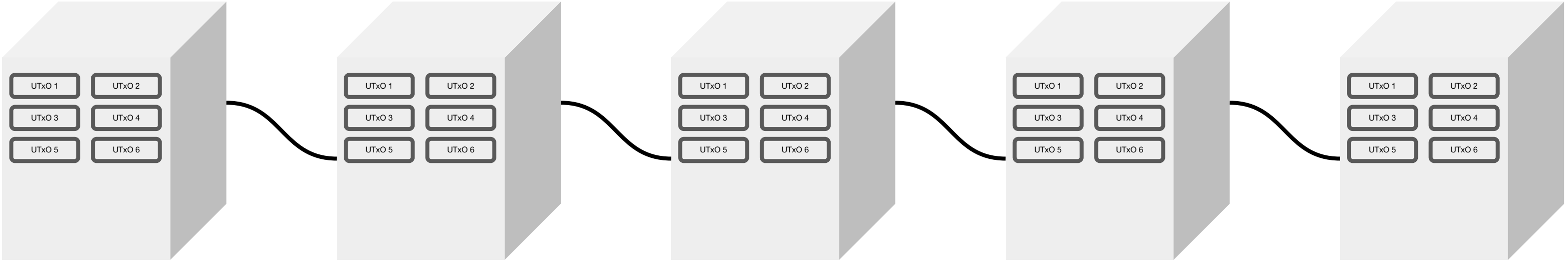
+ State Machine

Bringing steps to the proofs

Blockchain as a State Machine

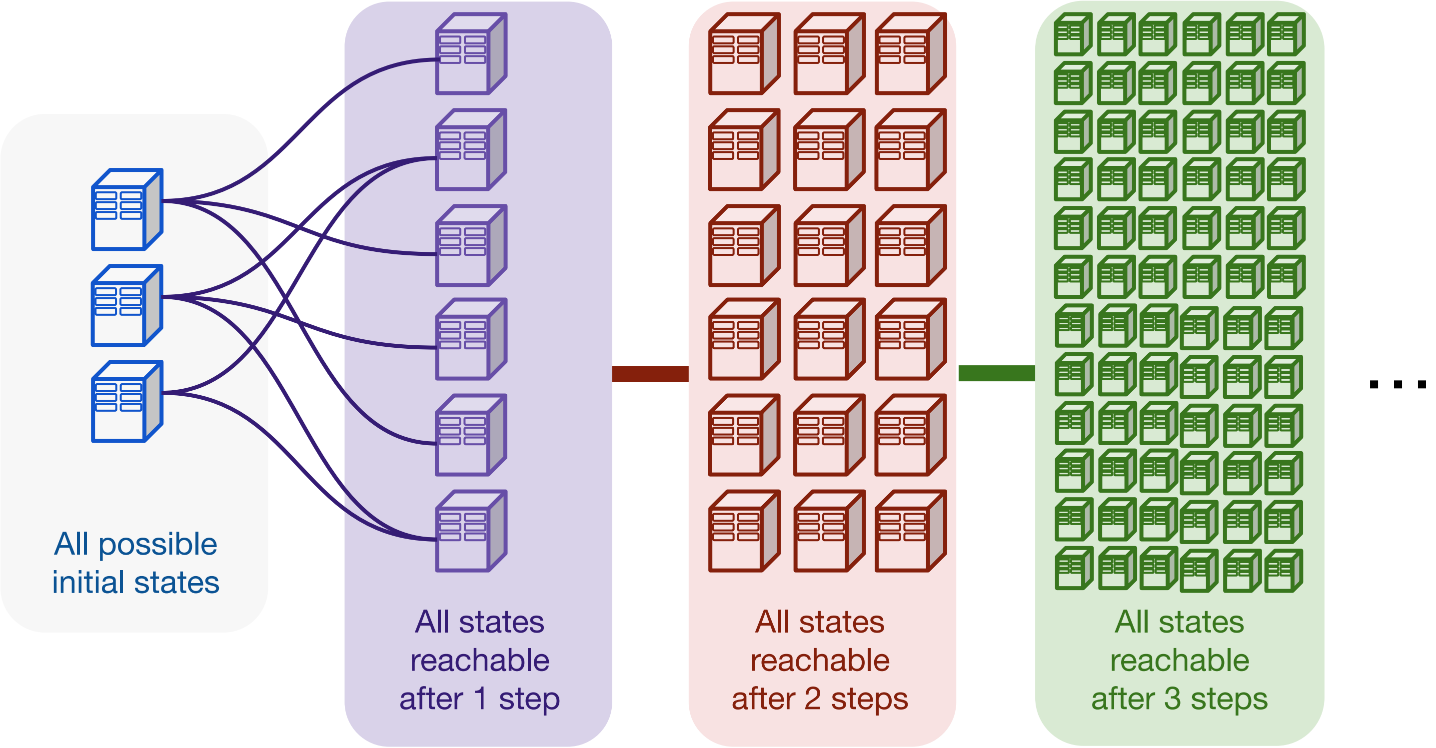


From a step to a trace of execution

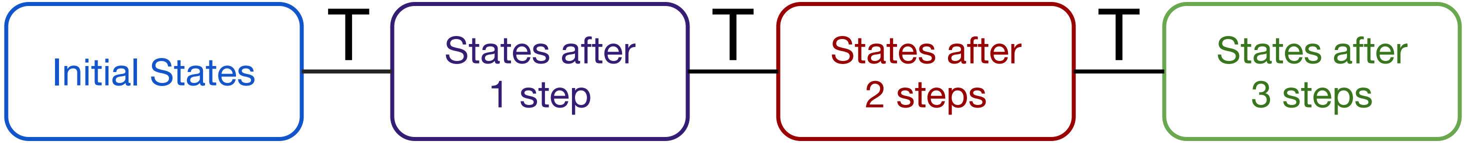


And now we have traces of execution !

Forest of all executions



Bounded Model Checking



$$\forall(s_0, s_1, s_2, s_3)$$

$$\boxed{I(s_0)} \wedge \boxed{T(s_0, s_1)} \wedge \boxed{T(s_1, s_2)} \wedge \boxed{T(s_2, s_3)} \wedge \neg P(s3)$$

Fast but incomplete: Very useful for bug finding

Induction

"Safe" state $\xrightarrow{T}$ "Safe" states after 1 step $\xrightarrow{T}$ "Safe" States after 2 steps $\xrightarrow{T}$ States after 3 steps

$$\forall(s_0, s_1, s_2, s_3)$$

$$\boxed{P(s_0)} \wedge \boxed{T(s_0, s_1) \wedge P(s_1)} \wedge \boxed{T(s_1, s_2) \wedge P(s_2)} \wedge \boxed{T(s_2, s_3)}$$

$$\implies P(s_3)$$

Very powerful to prove invariants for unbounded traces of execution

+ Live Demo: Escrow.lean

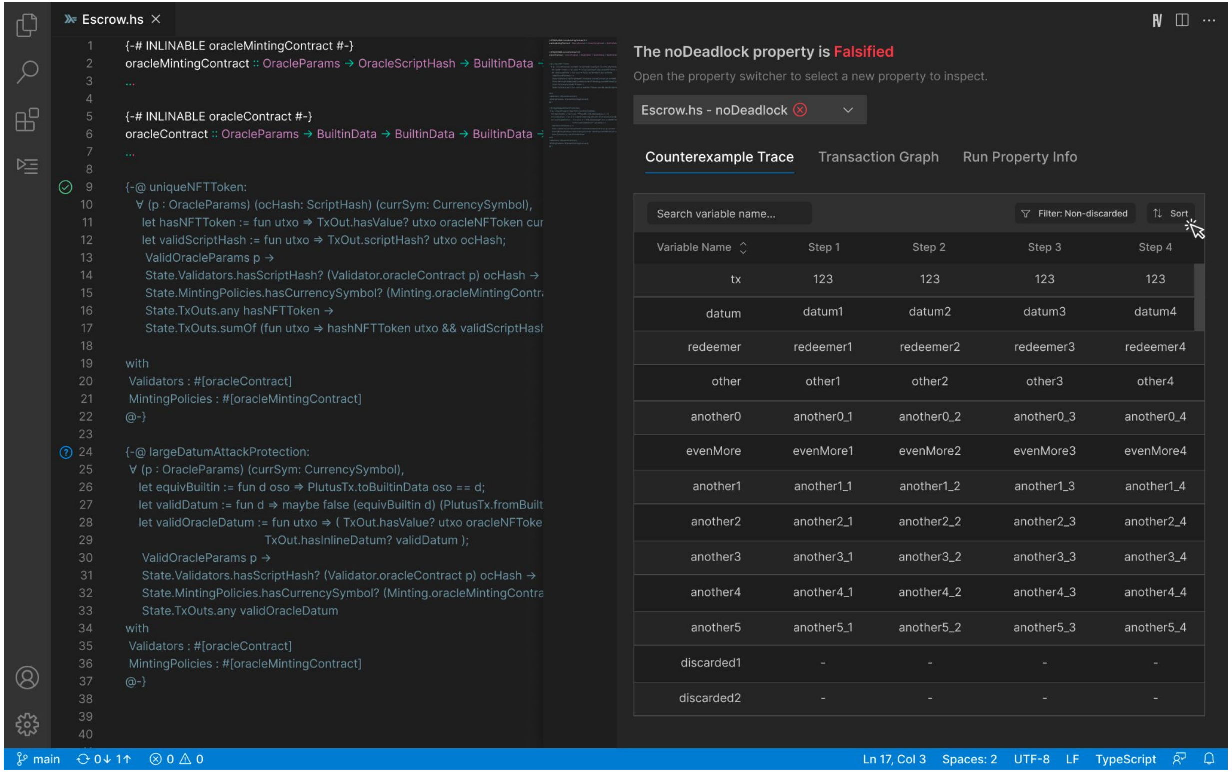
Traces of execution to weird cases

+ Current & Future

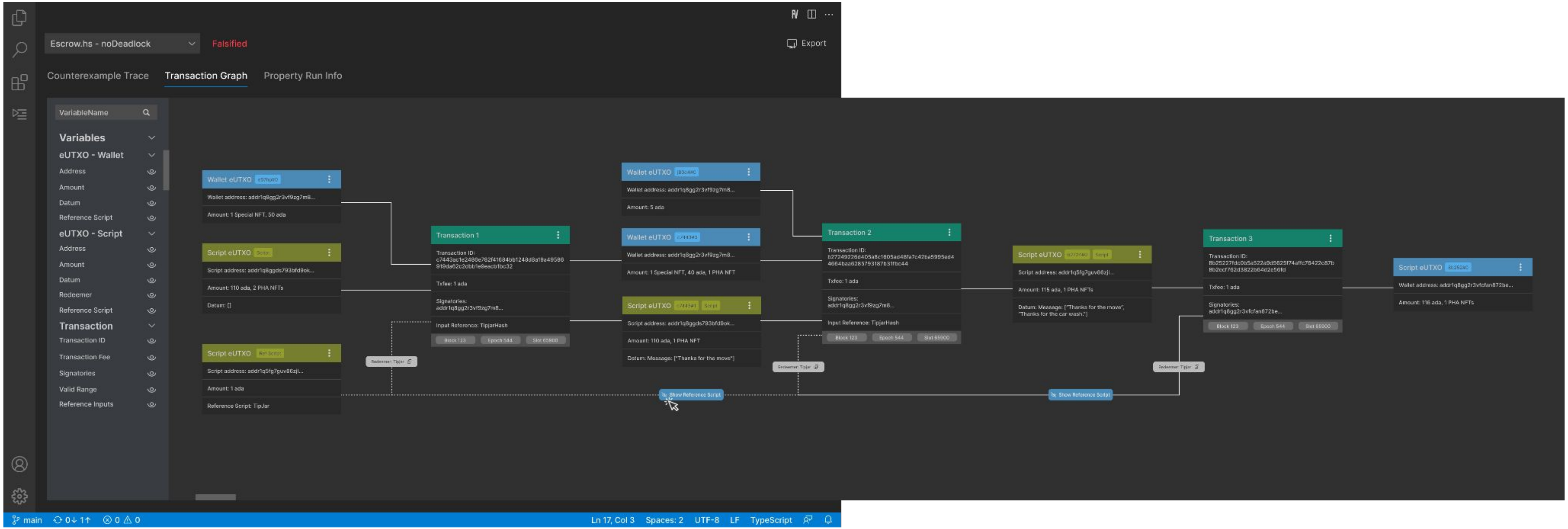
Conclusion

- Application of formal verification at the source code level
- Empowering smart contract developers to use formal verification with minimal effort
- Cost and time efficient verification with the already formalized Cardano context
- Easy debugging with counterexamples for every failed property
- Integration into VS Code for integration into traditional development workflows
- CLI tool for integration in CI/CD or uses outside VS Code

Counter Example exploration in VS Code



Trace visualization in VS Code



Roadmap

2025

Stable version

Transpilation from Plinth

Automated trace reasoning

UPLC equivalence checking

VS Code integration

2026

Extended support

Transpilation from Aiken

Automated common attacks verification

Scalability to complex DApps

Continuous updates and improvements

2027

Other chains

Midnight

Continuous updates and improvements

+GET PROVING_

Contact us:
afv_earlyaccessprogram@iohk.io