Program logics for ledgers

Orestis Melkonian, Wouter Swierstra, James Chapman

May 15th 2023, AIM XXXVI @ Delft

Motivation

- Local & modular reasoning for UTxO blockchain ledgers
- Entertain the following analogy with concurrency/PL:

Blockchain		Concurrency Theory
ledgers	$\leftrightarrow$	computer memory
memory locations	$\leftrightarrow$	accounts
data values	$\leftrightarrow$	account balances
smart contracts	$\leftrightarrow$	programs accessing memory

Approach

Investigate multiple semantics in different systems of increasing complexity

Operational

$$\frac{P(s) \quad (l, s) \rightsquigarrow^* s'}{Q(s')}$$

$\simeq$

Denotational

$$P \vdash Q \circ [\![l]\!]$$

$\simeq$ $\simeq$

Axiomatic

$$\{P\} l \{Q\}$$

Simple Model

```
module _ (Part : Type) ⦃ _ : DecEq Part ⦄ where


S = Map⟨ Part ↦ ℤ ⟩

record Tx : Type where
  constructor _—→⟨_⟩_
  field sender   : Part
        value    : ℤ
        receiver : Part
open Tx public
unquoteDecl DecEq-Tx = DERIVE DecEq [ quote Tx , DecEq-Tx ]

L = List Tx
```

Simple Model: Denotational Semantics

```
Domain = S → S

record Denotable (A : Type) : Type where
  field ⟦_⟧ : A → Domain
open Denotable {{...}} public

instance
  ⟦T⟧ : Denotable Tx
  ⟦T⟧ .⟦_⟧ (A —→⟨ v ⟩ B) s = s [ A ↝ _- v ] [ B ↝ _+ v ]

  ⟦L⟧ : Denotable L
  ⟦L⟧ .⟦_⟧ [] = id
  ⟦L⟧ .⟦_⟧ (t ∷ l) = ⟦ l ⟧ ∘ ⟦ t ⟧

comp : ∀ x → ⟦ l ++ l′ ⟧ x ≡ (⟦ l′ ⟧ ∘ ⟦ l ⟧) x
comp {l = []} _ = refl
comp {l = t ∷ l} x = comp {l} (⟦ t ⟧ x)
```

Simple Model: Operational Semantics

```
infix 0 _—→_
data _—→_ : L × S → S → Type where

  base :
    ─────────────
    ε , s —→ s

  step : let t = A —→⟨ v ⟩ B in

    l , ⟦ t ⟧ s —→ s′
    ────────────────────
    t :: l , s —→ s′
```

```
denot⇔oper :
  ⟦ l ⟧ s ≡ s′
  ══════════════
  l , s —→ s′
oper-comp :
  • l      , s  —→ s′
  • l′     , s′ —→ s″
   ─────────────────────
   l ++ l′ , s  —→ s″
```

Simple Model: Axiomatic Semantics (Hoare Logic)

```
Assertion = Pred₀ S
⟨_⟩_⟨_⟩ : Assertion → L → Assertion → Type
⟨ P ⟩ l ⟨ Q ⟩ = P ⊢ Q ∘ ⟦ l ⟧

hoare-base :
  ──────────────
  ⟨ P ⟩ [] ⟨ P ⟩
hoare-base = id

hoare-step :
  ⟨ P ⟩ l ⟨ Q ⟩
  ─────────────────────────
  ⟨ P ∘ ⟦ t ⟧ ⟩ t ∷ l ⟨ Q ⟩
hoare-step PlQ {_} = PlQ
```

Simple Model: Axiomatic Semantics (Hoare Logic)

```
consequence :
  • P′ ⊢ P
  • Q ⊢ Q′
  • ⟨ P ⟩ l ⟨ Q ⟩
  ─────────────────
  ⟨ P′ ⟩ l ⟨ Q′ ⟩
consequence ⊢P Q⊢ PlQ
  = Q⊢ ∘ PlQ ∘ ⊢P
```

```
hoare-step′ :
  • ⟨ P ⟩ l ⟨ Q ⟩
  • ⟨ Q ⟩ l′ ⟨ R ⟩
  ─────────────────────
  ⟨ P ⟩ l ++ l′ ⟨ R ⟩
hoare-step′ {P}{l}{Q}{l′}{R} PlQ QlR =
  begin P                       ⊢⟨ PlQ ⟩
        Q ∘ ⟦ l ⟧               ⊢⟨ QlR ⟩
        R ∘ (⟦ l′ ⟧ ∘ ⟦ l ⟧) ≐˘⟨ cong R ∘ comp {l} {l′} ⟩
        R ∘ ⟦ l ++ l′ ⟧       ∎ where open ⊢-Reasoning
```

Simple Model: Separation Logic

```agda
emp : Assertion
emp m = ∀ k → m k ≡ ε

_∗_ : Op₂ Assertion
(P ∗ Q) s = ∃ λ s₁ → ∃ λ s₂ → ⟨ s₁ ◇ s₂ ⟩≡ s × P s₁ × Q s₂


∗↔ : P ∗ Q ⊢ Q ∗ P
∗↔ (s₁ , s₂ , ≡s , Ps₁ , Qs₂) = s₂ , s₁ , ◇≡-comm {x = s₁}{s₂} ≡s , Qs₂ , Ps₁

∗↝ : P ∗ Q ∗ R ⊢ (P ∗ Q) ∗ R
∗↝ {x = s} (s₁ , s₂₃ , ≡s , Ps₁ , (s₂ , s₃ , ≡s₂₃ , Qs₂ , Rs₃)) =
  (s₁ ◇ s₂) , s₃ , ◇≈-assocʳ {m₁ = s₁} ≡s ≡s₂₃ , (s₁ , s₂ , ≈-refl , Ps₁ , Qs₂) , Rs₃

↜∗ : (P ∗ Q) ∗ R ⊢ P ∗ Q ∗ R
↜∗ {x = s} (s₁₂ , s₃ , ≡s , (s₁ , s₂ , ≡s₁₂ , Ps₁ , Qs₂) , Rs₃) =
  s₁ , s₂ ◇ s₃ , ◇≈-assocˡ {m₁ = s₁}{s₂} ≡s ≡s₁₂ , Ps₁ , (s₂ , s₃ , ≈-refl , Qs₂ , Rs₃)
```

Simple Model: Frame Rule

```
◇-⟦⟧ :
  ⟨ s₁ ◇ s₂ ⟩≡ s
  ────────────────────────
  ⟨ ⟦ l ⟧ s₁ ◇ s₂ ⟩≡ ⟦ l ⟧ s
[FRAME] :
  ⟨ P ⟩ l ⟨ Q ⟩
  ────────────────────────
  ⟨ P ∗ R ⟩ l ⟨ Q ∗ R ⟩
[FRAME] {l = l} PlQ (s₁ , s₂ , ≡s , Ps₁ , Rs₂) =
  ⟦ l ⟧ s₁ , s₂ , ◇-⟦⟧ {l = l} ≡s , PlQ Ps₁ , Rs₂
```

Simple Model: Concurrent Separation Logic

$\diamond$-interleave :

- $l_1 \parallel l_2 \equiv l$
- $\langle s_1 \diamond s_2 \rangle \equiv s$

$$\frac{}{\langle [\![l_1]\!]\, s_1 \diamond [\![l_2]\!]\, s_2 \rangle \equiv [\![l]\!]\, s}$$

[PAR] :

- $l_1 \parallel l_2 \equiv l$
- $\langle P_1 \rangle\, l_1 \, \langle Q_1 \rangle$
- $\langle P_2 \rangle\, l_2 \, \langle Q_2 \rangle$

$$\frac{}{\langle P_1 * P_2 \rangle\, l \, \langle Q_1 * Q_2 \rangle}$$

$[PAR]\ \{l_1\}\ \{l_2\}\ \{l\} \equiv l\ Pl_1Q\ Pl_2Q\ \{s\}\ (s_1\ ,\ s_2\ ,\ \equiv s\ ,\ Ps_1\ ,\ Ps_2) =$
$\quad [\![l_1]\!]\, s_1\ ,\ [\![l_2]\!]\, s_2\ ,\ \diamond\text{-interleave} \equiv l \equiv s\ ,\ Pl_1Q\ Ps_1\ ,\ Pl_2Q\ Ps_2$

Simple Model: Example derivation (monolithic)

```
A B C D : Part
t₁ = A —→⟨ 1ℤ ⟩ B; t₂ = D —→⟨ 1ℤ ⟩ C; t₃ = B —→⟨ 1ℤ ⟩ A; t₄ = C —→⟨ 1ℤ ⟩ D
t₁₋₄ = L ∋ ⟦ t₁ , t₂ , t₃ , t₄ ⟧

_ : ⟨ A ↦ 1ℤ ∗ B ↦ 0ℤ ∗ C ↦ 0ℤ ∗ D ↦ 1ℤ ⟩ t₁₋₄ ⟨ A ↦ 1ℤ ∗ B ↦ 0ℤ ∗ C ↦ 0ℤ ∗ D ↦ 1ℤ ⟩
_ = begin A ↦ 1ℤ ∗ B ↦ 0ℤ   ∗ C ↦ 0ℤ ∗ D ↦ 1ℤ ~⟪ ∗↝ ⟩
          (A ↦ 1ℤ ∗ B ↦ 0ℤ) ∗ C ↦ 0ℤ ∗ D ↦ 1ℤ ~⟨ t₁ :- [FRAME] (C ↦ 0ℤ ∗ D ↦ 1ℤ) (A ↝ B) ⟩
          (A ↦ 0ℤ ∗ B ↦ 1ℤ) ∗ C ↦ 0ℤ ∗ D ↦ 1ℤ ~⟪ ∗↔ ⟩
          (C ↦ 0ℤ ∗ D ↦ 1ℤ) ∗ A ↦ 0ℤ ∗ B ↦ 1ℤ ~⟨ t₂ :- [FRAME] (A ↦ 0ℤ ∗ B ↦ 1ℤ) (C ↜ D) ⟩
          (C ↦ 1ℤ ∗ D ↦ 0ℤ) ∗ A ↦ 0ℤ ∗ B ↦ 1ℤ ~⟪ ∗↔ ⟩
          (A ↦ 0ℤ ∗ B ↦ 1ℤ) ∗ C ↦ 1ℤ ∗ D ↦ 0ℤ ~⟨ t₃ :- [FRAME] (C ↦ 1ℤ ∗ D ↦ 0ℤ) (A ↜ B) ⟩
          (A ↦ 1ℤ ∗ B ↦ 0ℤ) ∗ C ↦ 1ℤ ∗ D ↦ 0ℤ ~⟪ ∗↔ ⟩
          (C ↦ 1ℤ ∗ D ↦ 0ℤ) ∗ A ↦ 1ℤ ∗ B ↦ 0ℤ ~⟨ t₄ :- [FRAME] (A ↦ 1ℤ ∗ B ↦ 0ℤ) (C ↝ D) ⟩
          (C ↦ 0ℤ ∗ D ↦ 1ℤ) ∗ A ↦ 1ℤ ∗ B ↦ 0ℤ ~⟪ ∗↔ ⟩
          (A ↦ 1ℤ ∗ B ↦ 0ℤ) ∗ C ↦ 0ℤ ∗ D ↦ 1ℤ ~⟪ ↜∗ ⟩
          A ↦ 1ℤ ∗ B ↦ 0ℤ   ∗ C ↦ 0ℤ ∗ D ↦ 1ℤ ∎
```

Simple Model: Example derivation (modular)

```
_ : ⟨ A ↦ 1ℤ ∗ B ↦ 0ℤ ∗ C ↦ 0ℤ ∗ D ↦ 1ℤ ⟩ t₁-₄ ⟨ A ↦ 1ℤ ∗ B ↦ 0ℤ ∗ C ↦ 0ℤ ∗ D ↦ 1ℤ ⟩
_ = begin A ↦ 1ℤ ∗ B ↦ 0ℤ   ∗ C ↦ 0ℤ ∗ D ↦ 1ℤ ~⟪ ∗↝ ⟩
         (A ↦ 1ℤ ∗ B ↦ 0ℤ) ∗ C ↦ 0ℤ ∗ D ↦ 1ℤ ~⟨ t₁-₄ :- [PAR] auto H₁ H₂ ⟩++
         (A ↦ 1ℤ ∗ B ↦ 0ℤ) ∗ C ↦ 0ℤ ∗ D ↦ 1ℤ ~⟪ ↜∗ ⟩
          A ↦ 1ℤ ∗ B ↦ 0ℤ   ∗ C ↦ 0ℤ ∗ D ↦ 1ℤ ∎
  where
    H₁ : ℝ⟨ A ↦ 1ℤ ∗ B ↦ 0ℤ ⟩ t₁ ∷ t₃ ∷ [] ⟨ A ↦ 1ℤ ∗ B ↦ 0ℤ ⟩
    H₁ = A ↦ 1ℤ ∗ B ↦ 0ℤ ~⟨ t₁ :- A ↝ B ⟩
         A ↦ 0ℤ ∗ B ↦ 1ℤ ~⟨ t₃ :- A ↜ B ⟩
         A ↦ 1ℤ ∗ B ↦ 0ℤ ∎
    H₂ : ℝ⟨ C ↦ 0ℤ ∗ D ↦ 1ℤ ⟩ t₂ ∷ t₄ ∷ [] ⟨ C ↦ 0ℤ ∗ D ↦ 1ℤ ⟩
    H₂ = C ↦ 0ℤ ∗ D ↦ 1ℤ ~⟨ t₂ :- C ↜ D ⟩
         C ↦ 1ℤ ∗ D ↦ 0ℤ ~⟨ t₄ :- C ↝ D ⟩
         C ↦ 0ℤ ∗ D ↦ 1ℤ ∎
```

Adding Partiality

```
S = Map⟨ Part ↦ ℕ ⟩

Domain = S → Maybe S

⟦T⟧ : Denotable Tx
⟦T⟧ .⟦_⟧ t s = M.when (isValidTx t s) (⟦ t ⟧₀ s)

⟦L⟧ : Denotable L
⟦L⟧ .⟦_⟧ [] s = just s
⟦L⟧ .⟦_⟧ (t ∷ l) = ⟦ t ⟧ >=> ⟦ l ⟧

comp : ∀ x → ⟦ l ++ l′ ⟧ x ≡ (⟦ l ⟧ >=> ⟦ l′ ⟧) x
```

Adding Partiality: Operational Semantics

```
infix 0 _—→_
data _—→_ : L × S → S → Type where

  base :
    ────────────
    ε , s —→ s

  step :
    • IsValidTx t s
    • l , ⟦ t ⟧₀ s —→ s′
      ──────────────────
      t ∷ l , s —→ s′
denot⇔oper :
  ⟦ l ⟧ s ≡ just s′
  ═════════════════
  l , s —→ s′
```

Adding Partiality: Lifting Predicates for Hoare Logic

```
weak↑ strong↑ : Pred₀ S → Pred₀ (Maybe S)
weak↑   = M.All.All
strong↑ = M.Any.Any

_↑∘_ : Pred₀ S → (S → Maybe S) → Pred₀ S
P ↑∘ f = strong↑ P ∘ f
⟨_⟩_⟨_⟩ : Assertion → L → Assertion → Type
⟨ P ⟩ ι ⟨ Q ⟩ = P ⊢ Q ↑∘ ⟦ ι ⟧
```

Adding Partiality: Frame Rule

```
◇-⟦⟧ : ∀ s₁′ →
  • ⟦ l ⟧ s₁ ≡ just s₁′
  • ⟨ s₁ ◇ s₂ ⟩≡ s
    ────────────────────────────────
    (⟨ s₁′ ◇ s₂ ⟩≡_ ↑∘ ⟦ l ⟧) s
[FRAME] : ∀ R →
  ⟨ P ⟩ l ⟨ Q ⟩
  ────────────────────────
  ⟨ P ∗ R ⟩ l ⟨ Q ∗ R ⟩
```

Adding Partiality: Parallel Rule

◇-interleave :

- ($l_1 \parallel l_2 \equiv l$)
- $\langle\, s_1 \diamond s_2 \,\rangle \equiv s$
- $[\![\, l_1 \,]\!]\; s_1 \equiv \mathsf{just}\; s_1{}'$
- $[\![\, l_2 \,]\!]\; s_2 \equiv \mathsf{just}\; s_2{}'$

$$\frac{}{\exists\, \lambda\, s' \to ([\![\, l \,]\!]\; s \equiv \mathsf{just}\; s') \times (\langle\, s_1{}' \diamond s_2{}' \,\rangle \equiv s')}$$

[PAR] :

- $l_1 \parallel l_2 \equiv l$
- $\langle\, P_1 \,\rangle\; l_1\; \langle\, Q_1 \,\rangle$
- $\langle\, P_2 \,\rangle\; l_2\; \langle\, Q_2 \,\rangle$

$$\frac{}{\langle\, P_1 * P_2 \,\rangle\; l\; \langle\, Q_1 * Q_2 \,\rangle}$$

Adding Partiality: Example derivation (monolithic)

```
A B C D : Part
t₁ = A —→⟨ 1 ⟩ B; t₂ = D —→⟨ 1 ⟩ C; t₃ = B —→⟨ 1 ⟩ A; t₄ = C —→⟨ 1 ⟩ D
t₁₋₄ = L ∋ ⟦ t₁ , t₂ , t₃ , t₄ ⟧

_ : ⟨ A ↦ 1 ∗ B ↦ 0 ∗ C ↦ 0 ∗ D ↦ 1 ⟩ t₁₋₄ ⟨ A ↦ 1 ∗ B ↦ 0 ∗ C ↦ 0 ∗ D ↦ 1 ⟩
_ = begin A ↦ 1 ∗ B ↦ 0   ∗ C ↦ 0 ∗ D ↦ 1 ~⟪ ∗↝ ⟩
          (A ↦ 1 ∗ B ↦ 0) ∗ C ↦ 0 ∗ D ↦ 1 ~⟨ t₁ :- [FRAME] (C ↦ 0 ∗ D ↦ 1) (A ↝ B) ⟩
          (A ↦ 0 ∗ B ↦ 1) ∗ C ↦ 0 ∗ D ↦ 1 ~⟪ ∗↔ ⟩
          (C ↦ 0 ∗ D ↦ 1) ∗ A ↦ 0 ∗ B ↦ 1 ~⟨ t₂ :- [FRAME] (A ↦ 0 ∗ B ↦ 1) (C ↜ D) ⟩
          (C ↦ 1 ∗ D ↦ 0) ∗ A ↦ 0 ∗ B ↦ 1 ~⟪ ∗↔ ⟩
          (A ↦ 0 ∗ B ↦ 1) ∗ C ↦ 1 ∗ D ↦ 0 ~⟨ t₃ :- [FRAME] (C ↦ 1 ∗ D ↦ 0) (A ↜ B) ⟩
          (A ↦ 1 ∗ B ↦ 0) ∗ C ↦ 1 ∗ D ↦ 0 ~⟪ ∗↔ ⟩
          (C ↦ 1 ∗ D ↦ 0) ∗ A ↦ 1 ∗ B ↦ 0 ~⟨ t₄ :- [FRAME] (A ↦ 1 ∗ B ↦ 0) (C ↝ D) ⟩
          (C ↦ 0 ∗ D ↦ 1) ∗ A ↦ 1 ∗ B ↦ 0 ~⟪ ∗↔ ⟩
          (A ↦ 1 ∗ B ↦ 0) ∗ C ↦ 0 ∗ D ↦ 1 ~⟪ ↜∗ ⟩
          A ↦ 1 ∗ B ↦ 0   ∗ C ↦ 0 ∗ D ↦ 1 ∎
```

Adding Partiality: Example derivation (modular)

```
_ : ⟨ A ↦ 1 ∗ B ↦ 0 ∗ C ↦ 0 ∗ D ↦ 1 ⟩ t₁₋₄ ⟨ A ↦ 1 ∗ B ↦ 0 ∗ C ↦ 0 ∗ D ↦ 1 ⟩
_ = begin A ↦ 1 ∗ B ↦ 0   ∗ C ↦ 0 ∗ D ↦ 1 ~⟪ ∗↝ ⟩
         (A ↦ 1 ∗ B ↦ 0) ∗ C ↦ 0 ∗ D ↦ 1 ~⟨ t₁₋₄ :- [PAR] auto H₁ H₂ ⟩++
         (A ↦ 1 ∗ B ↦ 0) ∗ C ↦ 0 ∗ D ↦ 1 ~⟪ ↜∗ ⟩
          A ↦ 1 ∗ B ↦ 0   ∗ C ↦ 0 ∗ D ↦ 1 ∎
  where
    H₁ : ⟨ A ↦ 1 ∗ B ↦ 0 ⟩ t₁ ∷ t₃ ∷ [] ⟨ A ↦ 1 ∗ B ↦ 0 ⟩
    H₁ = begin A ↦ 1 ∗ B ↦ 0 ~⟨ t₁ :- A ↝ B ⟩
               A ↦ 0 ∗ B ↦ 1 ~⟨ t₃ :- A ↜ B ⟩
               A ↦ 1 ∗ B ↦ 0 ∎
    H₂ : ⟨ C ↦ 0 ∗ D ↦ 1 ⟩ t₂ ∷ t₄ ∷ [] ⟨ C ↦ 0 ∗ D ↦ 1 ⟩
    H₂ = begin C ↦ 0 ∗ D ↦ 1 ~⟨ t₂ :- C ↜ D ⟩
               C ↦ 1 ∗ D ↦ 0 ~⟨ t₄ :- C ↝ D ⟩
               C ↦ 0 ∗ D ↦ 1 ∎
```

UTxO: Barebones Setup

```
S = Map⟨ TxOutputRef ↦ TxOutput ⟩

record IsValidTx (tx : Tx) (utxos : S) : Type where
  field
    noDoubleSpending :
      ·Unique (outputRefs tx)

    validOutputRefs :
      ∀[ ref ∈ outputRefs tx ] (ref ∈ᵈ utxos)
    preservesValues :
      tx .forge + ∑ resolvedInputs (value ∘ proj₂) ≡ ∑ (tx .outputs) value

    allInputsValidate :
      ∀[ i ∈ tx .inputs ] T (i .validator txInfo (i .redeemer))

    validateValidHashes :
      ∀[ (i , o) ∈ resolvedInputs ] (o .address ≡ i .validator ♯)
```

UTxO: Denotational Semantics

```
instance
  ⟦T⟧ : Denotable Tx
  ⟦T⟧ .⟦_⟧ tx s = M.when (isValidTx tx s) (s ─ outputRefs tx ∪ utxoTx tx)

  ⟦L⟧ : Denotable L
  ⟦L⟧ .⟦_⟧ [] s = just s
  ⟦L⟧ .⟦_⟧ (t ∷ l) = ⟦ t ⟧ >=> ⟦ l ⟧
comp : ∀ x → ⟦ l ++ l′ ⟧ x ≡ (⟦ l ⟧ >=> ⟦ l′ ⟧) x
comp {l = []} x = refl
comp {l = t ∷ l} x with ⟦ t ⟧ x
... | nothing = refl
... | just s = comp {l} s
```

UTxO: Separation via Disjointness

```
_∗_ : Op₂ Assertion
(P ∗ Q) s = ∃ λ s₁ → ∃ λ s₂ → ⟨ s₁ ⊎ s₂ ⟩≡ s × P s₁ × Q s₂
```

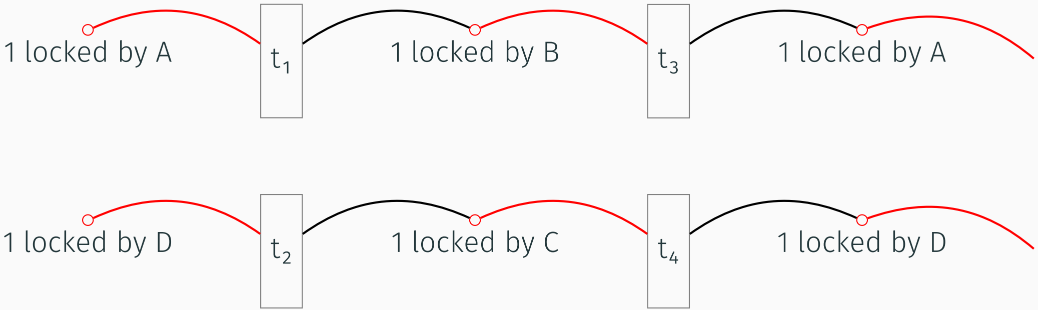
```
⊎-⟦⟧ : ∀ s₁′ →
  • ⟦ l ⟧ s₁ ≡ just s₁′
  • ⟨ s₁ ⊎ s₂ ⟩≡ s
  ────────────────────────────
  (⟨ s₁′ ⊎ s₂ ⟩≡_ ↑∘ ⟦ l ⟧) s

[FRAME] : ∀ R →
  • l ♯ R
  • ⟨ P ⟩ l ⟨ Q ⟩
  ──────────────────
  ⟨ P ∗ R ⟩ l ⟨ Q ∗ R ⟩
```

```
[PAR] :
  • l₁ ♯ P₂
  • l₂ ♯ P₁
  • l₁ ∥ l₂ ≡ l
  • ⟨ P₁ ⟩ l₁ ⟨ Q₁ ⟩
  • ⟨ P₂ ⟩ l₂ ⟨ Q₂ ⟩
  ─────────────────────────────
  ⟨ P₁ ∗ P₂ ⟩ l ⟨ Q₁ ∗ Q₂ ⟩
```

UTxO: Example transaction graph

```
A B C D : Address
t₁₋₄ = L ∋ ⟦ t₁ , t₂ , t₃ , t₄ ⟧
```



UTxO: Example derivation (monolithic)

```
_ : ⟨ t₀₀ ↦ 1 at A ∗ t₀₁ ↦ 1 at D ⟩ t₁₋₄ ⟨ t₃₀ ↦ 1 at A ∗ t₄₀ ↦ 1 at D ⟩
_ = begin t₀₀ ↦ 1 at A ∗ t₀₁ ↦ 1 at D ~⟨ t₁ :- [FRAME] (t₀₁ ↦ 1 at D) t₁♯ ⋯ ⟩
          t₁₀ ↦ 1 at B ∗ t₀₁ ↦ 1 at D ~⟪ ∗↔ ⟩
          t₀₁ ↦ 1 at D ∗ t₁₀ ↦ 1 at B ~⟨ t₂ :- [FRAME] (t₁₀ ↦ 1 at B) t₂♯ ⋯ ⟩
          t₂₀ ↦ 1 at C ∗ t₁₀ ↦ 1 at B ~⟪ ∗↔ ⟩
          t₁₀ ↦ 1 at B ∗ t₂₀ ↦ 1 at C ~⟨ t₃ :- [FRAME] (t₂₀ ↦ 1 at C) t₃♯ ⋯ ⟩
          t₃₀ ↦ 1 at A ∗ t₂₀ ↦ 1 at C ~⟪ ∗↔ ⟩
          t₂₀ ↦ 1 at C ∗ t₃₀ ↦ 1 at A ~⟨ t₄ :- [FRAME] (t₃₀ ↦ 1 at A) t₄♯ ⋯ ⟩
          t₄₀ ↦ 1 at D ∗ t₃₀ ↦ 1 at A ~⟪ ∗↔ ⟩
          t₃₀ ↦ 1 at A ∗ t₄₀ ↦ 1 at D ∎
  where postulate t₁♯ : [ t₁ ] ♯ (t₀₁ ↦ 1 at D)
                  t₂♯ : [ t₂ ] ♯ (t₁₀ ↦ 1 at B)
                  t₃♯ : [ t₃ ] ♯ (t₂₀ ↦ 1 at C)
                  t₄♯ : [ t₄ ] ♯ (t₃₀ ↦ 1 at A)
```

UTxO: Example derivation (modular)

```
_ : ⟨ t₀₀ ↦ 1 at A ∗ t₀₁ ↦ 1 at D ⟩ t₁₋₄ ⟨ t₃₀ ↦ 1 at A ∗ t₄₀ ↦ 1 at D ⟩
_ = begin t₀₀ ↦ 1 at A ∗ t₀₁ ↦ 1 at D ~⟨ t₁₋₄ :- [PAR] … auto H₁ H₂ ⟩++
          t₃₀ ↦ 1 at A ∗ t₄₀ ↦ 1 at D ∎
  where
    H₁ : ⟨ t₀₀ ↦ 1 at A ⟩ t₁ ∷ t₃ ∷ [] ⟨ t₃₀ ↦ 1 at A ⟩
    H₁ = begin t₀₀ ↦ 1 at A ~⟨ t₁ :- … ⟩
               t₁₀ ↦ 1 at B ~⟨ t₃ :- … ⟩
               t₃₀ ↦ 1 at A ∎
    H₂ : ⟨ t₀₁ ↦ 1 at D ⟩ t₂ ∷ t₄ ∷ [] ⟨ t₄₀ ↦ 1 at D ⟩
    H₂ = begin t₀₁ ↦ 1 at D ~⟨ t₂ :- … ⟩
               t₂₀ ↦ 1 at C ~⟨ t₄ :- … ⟩
               t₄₀ ↦ 1 at D ∎
```

Abstract UTxO: Setup

```
S = Bag⟨ TxOutput ⟩

record IsValidTx (tx : Tx) (utxos : S) : Type where
  field
    validOutputRefs :
      stxoTx tx ⊆ˢ utxos

    preservesValues :
      tx .forge + ∑ (tx .inputs) (value ∘ outputRef) ≡ ∑ (tx .outputs) value

    allInputsValidate :
      ∀[ i ∈ tx .inputs ] T (i .validator txInfo (i .redeemer))

    validateValidHashes :
      ∀[ i ∈ tx .inputs ] (i .outputRef .address ≡ i .validator ♯)
```

Abstract UTxO: Denotational Semantics

```
instance
  ⟦T⟧ : Denotable Tx
  ⟦T⟧ .⟦_⟧ tx s = M.when (isValidTx tx s) (s ─ stxoTx tx ∪ utxoTx tx)

  ⟦L⟧ : Denotable L
  ⟦L⟧ .⟦_⟧ [] s = just s
  ⟦L⟧ .⟦_⟧ (t ∷ l) = ⟦ t ⟧ >=> ⟦ l ⟧
```

Abstract UTxO: Monoidal Separation once again

```
_∗_ : Op₂ Assertion
(P ∗ Q) s = ∃ λ s₁ → ∃ λ s₂ → ⟨ s₁ ◇ s₂ ⟩≡ s × P s₁ × Q s₂

∗↔ : P ∗ Q ⊢ Q ∗ P
∗↔ {x = s} (s₁ , s₂ , ≡s , Ps₁ , Qs₂) = s₂ , s₁ , ◇≡-comm {s = s}{s₁}{s₂} ≡s , Qs₂ , Ps₁

∗↝ : P ∗ Q ∗ R ⊢ (P ∗ Q) ∗ R
∗↝ {x = s} (s₁ , s₂₃ , ≡s , Ps₁ , (s₂ , s₃ , ≡s₂₃ , Qs₂ , Rs₃)) =
  let ≡s₁₂ = ◇≈-assocʳ {s₁ = s₁}{s₂₃}{s}{s₂}{s₃} ≡s ≡s₂₃ in
  (s₁ ◇ s₂) , s₃ , ≡s₁₂ , (s₁ , s₂ , ≈-refl {x = s₁ ∪ s₂} , Ps₁ , Qs₂) , Rs₃

↜∗ : (P ∗ Q) ∗ R ⊢ P ∗ Q ∗ R
↜∗ {x = s} (s₁₂ , s₃ , ≡s , (s₁ , s₂ , ≡s₁₂ , Ps₁ , Qs₂) , Rs₃) =
  let ≡s₂₃ = ◇≈-assocˡ {s₁₂ = s₁₂}{s₃}{s}{s₁}{s₂} ≡s ≡s₁₂ in
  s₁ , s₂ ◇ s₃ , ≡s₂₃ , Ps₁ , (s₂ , s₃ , ≈-refl {x = s₂ ∪ s₃}, Qs₂ , Rs₃)
```

Abstract UTxO: Separation Logic Rules

[FRAME] : $\forall R \rightarrow$

$$\frac{\langle P \rangle\, l\, \langle Q \rangle}{\langle P * R \rangle\, l\, \langle Q * R \rangle}$$

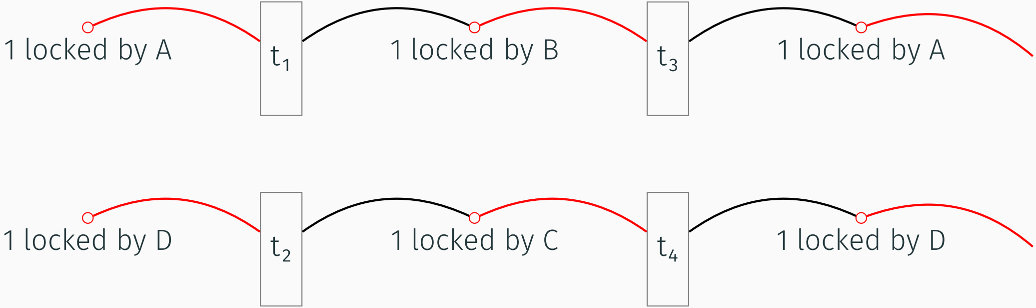
[PAR] :

- $l_1 \parallel l_2 \equiv l$
- $\langle P_1 \rangle\, l_1\, \langle Q_1 \rangle$
- $\langle P_2 \rangle\, l_2\, \langle Q_2 \rangle$

$$\frac{}{\langle P_1 * P_2 \rangle\, l\, \langle Q_1 * Q_2 \rangle}$$

Abstract UTxO: Example transaction graph

```
A B C D : Address
t₁₋₄ = L ∋ ⟦ t₁ , t₂ , t₃ , t₄ ⟧
```



Abstract UTxO: Example derivation (monolithic)

```
_ : ⟨ A ↦ 1 ∗ B ↦ 0 ∗ C ↦ 0 ∗ D ↦ 1 ⟩ t₁₋₄ ⟨ A ↦ 1 ∗ B ↦ 0 ∗ C ↦ 0 ∗ D ↦ 1 ⟩
_ = begin A ↦ 1 ∗ B ↦ 0   ∗ C ↦ 0 ∗ D ↦ 1 ~⟪ ∗↝ ⟩
          (A ↦ 1 ∗ B ↦ 0) ∗ C ↦ 0 ∗ D ↦ 1 ~⟨ t₁ :- [FRAME] (C ↦ 0 ∗ D ↦ 1) (A ↝ B) ⟩
          (A ↦ 0 ∗ B ↦ 1) ∗ C ↦ 0 ∗ D ↦ 1 ~⟪ ∗↔ ⟩
          (C ↦ 0 ∗ D ↦ 1) ∗ A ↦ 0 ∗ B ↦ 1 ~⟨ t₂ :- [FRAME] (A ↦ 0 ∗ B ↦ 1) (C ↜ D) ⟩
          (C ↦ 1 ∗ D ↦ 0) ∗ A ↦ 0 ∗ B ↦ 1 ~⟪ ∗↔ ⟩
          (A ↦ 0 ∗ B ↦ 1) ∗ C ↦ 1 ∗ D ↦ 0 ~⟨ t₃ :- [FRAME] (C ↦ 1 ∗ D ↦ 0) (A ↜ B) ⟩
          (A ↦ 1 ∗ B ↦ 0) ∗ C ↦ 1 ∗ D ↦ 0 ~⟪ ∗↔ ⟩
          (C ↦ 1 ∗ D ↦ 0) ∗ A ↦ 1 ∗ B ↦ 0 ~⟨ t₄ :- [FRAME] (A ↦ 1 ∗ B ↦ 0) (C ↝ D) ⟩
          (C ↦ 0 ∗ D ↦ 1) ∗ A ↦ 1 ∗ B ↦ 0 ~⟪ ∗↔ ⟩
          (A ↦ 1 ∗ B ↦ 0) ∗ C ↦ 0 ∗ D ↦ 1 ~⟪ ↜∗ ⟩
          A ↦ 1 ∗ B ↦ 0   ∗ C ↦ 0 ∗ D ↦ 1 ∎
```

Abstract UTxO: Example derivation (modular)

```
_ : ⟨ A ↦ 1 ∗ B ↦ 0 ∗ C ↦ 0 ∗ D ↦ 1 ⟩ t₁₋₄ ⟨ A ↦ 1 ∗ B ↦ 0 ∗ C ↦ 0 ∗ D ↦ 1 ⟩
_ = begin A ↦ 1 ∗ B ↦ 0 ∗ C ↦ 0 ∗ D ↦ 1   ~⟪ ∗↝ ⟩
          (A ↦ 1 ∗ B ↦ 0) ∗ C ↦ 0 ∗ D ↦ 1 ~⟨ t₁₋₄ :- [PAR] auto H₁ H₂ ⟩++
          (A ↦ 1 ∗ B ↦ 0) ∗ C ↦ 0 ∗ D ↦ 1 ~⟪ ↜∗ ⟩
          A ↦ 1 ∗ B ↦ 0 ∗ C ↦ 0 ∗ D ↦ 1   ∎
  where
    H₁ : ℝ⟨ A ↦ 1 ∗ B ↦ 0 ⟩ t₁ ∷ t₃ ∷ [] ⟨ A ↦ 1 ∗ B ↦ 0 ⟩
    H₁ = A ↦ 1 ∗ B ↦ 0 ~⟨ t₁ :- A ↝ B ⟩
         A ↦ 0 ∗ B ↦ 1 ~⟨ t₃ :- A ↜ B ⟩
         A ↦ 1 ∗ B ↦ 0 ∎

    H₂ : ℝ⟨ C ↦ 0 ∗ D ↦ 1 ⟩ t₂ ∷ t₄ ∷ [] ⟨ C ↦ 0 ∗ D ↦ 1 ⟩
    H₂ = C ↦ 0 ∗ D ↦ 1 ~⟨ t₂ :- C ↜ D ⟩
         C ↦ 1 ∗ D ↦ 0 ~⟨ t₄ :- C ↝ D ⟩
         C ↦ 0 ∗ D ↦ 1 ∎
```

Sound Abstraction: States and Validity

```
absS : ℂ.S → 𝔸.S

absVT : ℂ.IsValidTx t s → ∃ λ t̂ → 𝔸.IsValidTx t̂ (absS s)

absVL : ℂ.ValidLedger s l → ∃ λ l̂ → 𝔸.ValidLedger (absS s) l̂
```

Sound Abstraction: Denotations Coincide

```
denot-abs-t : ∀ (vt : ℂ.IsValidTx t s) →
  𝔸.⟦ absT vt ⟧ (absS s) ≡ (absS <$> ℂ.⟦ t ⟧ s)
denot-abs : ∀ (vl : ℂ.ValidLedger s l) →
  𝔸.⟦ absL vl ⟧ (absS s) ≡ (absS <$> ℂ.⟦ l ⟧ s)
```

$$\begin{tikzcd} s \arrow[r, "\mathbb{C}.[\![\_]\!]"] \arrow[d, "absS"'] & s' \arrow[d, "absS"] \\ \hat{s} \arrow[r, "\mathbb{A}.[\![\_]\!] \circ abs"'] & \hat{s}' \end{tikzcd}$$

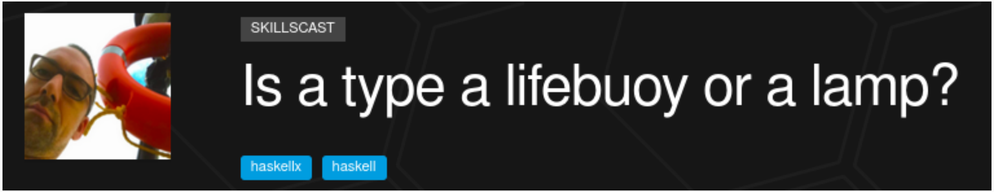
Sound Abstraction

```
soundness :
  ∀ (vl : ℂ.ValidLedger s l) →
  𝔸⟨ P ⟩ absL vl ⟨ Q ⟩
  ────────────────────────────
  ℂ⟨ P ∘ absS ⟩ l ⟨ Q ∘ absS ⟩
```

Future Work

- Deeper compositionality (i.e. monoidally exploit the values in the bag)
  - $\rightarrow$ will require further abstraction of split/merge transactions
- Go beyond the monetary values (states, transaction data)
  - $\rightarrow$ leads to more practical verification of smart contracts
- Generalise to multiple separation views, aka zooming levels
- Generically grow such separation logics, i.e. "Separation Logics à la carte"

Agda as a design guide, rather than merely a verification tool of existing systems.



Questions? Suggestions?