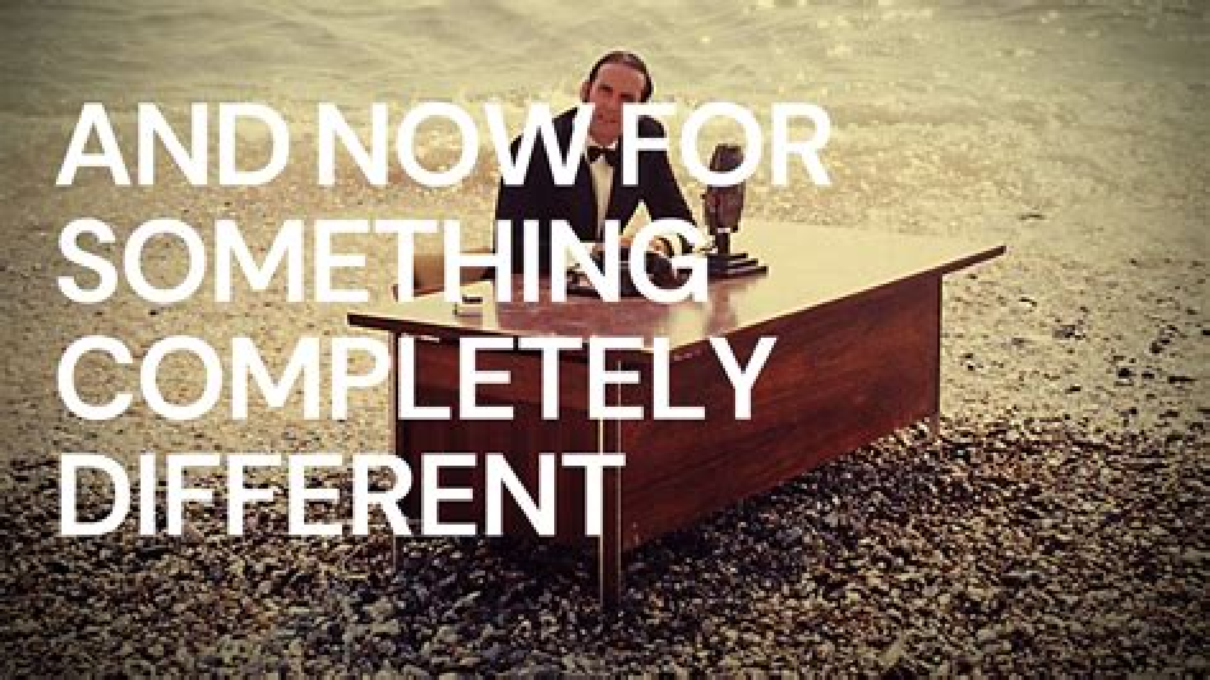

AND NOW FOR
SOMETHING
COMPLETELY
DIFFERENT

A readable and computable formalization of the Streamlet consensus protocol

Mauro Jaskelioff, Orestis Melkonian, James Chapman

4 May 2025, FMBC @ Hamilton, Canada

Motivation

- **Consensus** is an integral piece of blockchain technology
- We want *formally verified* implementations of these protocols

Approach

1. Formally present a readable specification of the protocol
2. Provide mechanized proofs about the protocol's properties (e.g. safety)
3. Make sure the specification is also computable
   - so that we can extract executable code out of the formalization
4. Formally verifying a full implementation is too unrealistic, but...
   - ...we can test that an implementation conforms to the formal model

Approach

1. Formally present a readable specification of the protocol
2. Provide mechanized proofs about the protocol's properties (e.g. safety)
3. Make sure the specification is also computable
    - so that we can extract executable code out of the formalization
4. Formally verifying a full implementation is too unrealistic, but...
    - ...we can test that an implementation conforms to the formal model

TOOL OF CHOICE: the **Agda** proof assistant

Global System

Infrastructure, common to all BFT-style consensus protocols

- cryptographic and consensus assumptions
- messaging between nodes/replicas
- modelling time

Global System: assumptions (crypto)

$\texttt{Epoch} = \mathbb{N}$

Global System: assumptions (crypto)

```
Hash : Type
```

```
record Hashable (A : Type) : Type where
  field _♯    : A → Hash
        ♯-inj : Injective≡ _♯
```

Global System: assumptions (crypto)

```
record HashAssumptions : Type₁ where
  field instance
    -- type formers
    Hashable-×     : ⦃ Hashable A ⦄ → ⦃ Hashable B ⦄ → Hashable (A × B)
    Hashable-⊎     : ⦃ Hashable A ⦄ → ⦃ Hashable B ⦄ → Hashable (A ⊎ B)
    Hashable-List  : ⦃ Hashable A ⦄ → Hashable (List A)
    Hashable-Maybe : ⦃ Hashable A ⦄ → Hashable (Maybe A)

    -- base types
    Hashable-⊤      : Hashable ⊤
    Hashable-ℕ      : Hashable ℕ
    Hashable-String : Hashable String
```

Global System: assumptions (crypto)

```
PrivateKey PublicKey Signature : Type
```

```
record SignatureAssumptions : Type₁ where
  field
    verify-signature : PublicKey → Signature → Hash → Bool
    sign'            : ⦃ Hashable A ⦄ → PrivateKey → A → Signature
```

Global System: assumptions (BFT)

```
record Assumptions : Type₁ where
  field
    nodes  : ℕ
    nodes⁺ : nodes > 0

  Pid : Type
  Pid = Fin nodes
      ⋮
```

Global System: assumptions (BFT)

```
field
  Honest : Pid → Type
  instance
    Dec-Honest : Honest ⁇¹
  Honest-irr : Irrelevant¹ Honest
  honest-majority : 3 * length honestPids > 2 * nodes
```

Global System: assumptions (BFT)

```
field
  epochLeader : Epoch → Pid
```

Global System: assumptions (BFT)

```
field
  Transaction : Type
  instance
    DecEq-Tx    : DecEq Transaction
    Hashable-Tx : Hashable Transaction
```

Global System: state transition as an inductive relation

```
data _→_ : GlobalState → GlobalState → Type
```

Global System: state

```
record GlobalState : Type where
  field e-now     : Epoch             networkBuffer : List Envelope
        stateMap : StateMap          history       : List Message
```

```
StateMap = HonestVec LocalState
```

Global System: step

```
data _→_ (s : GlobalState) : GlobalState → Type where

  Deliver :
    (env∈ : env ∈ s .networkBuffer) →
    ────────────────────────────────
    s → deliverMsg s env∈

  AdvanceEpoch :
    ────────────────────
    s → advanceEpoch s

  LocalStep : ⦃ _ : Honest p ⦄ →
    (p ▷ s .e-now ⊢ s @ p —[ m? ]→ ls′)
    ──────────────────────────────────
    s → broadcast p m? (s @ p ≔ ls′)

  DishonestStep :
    • ¬ Honest p   • NoSignatureForging m s
    ────────────────────────────────────
    s → broadcast p (just m) s
```

Global System: disallowing forged signatures

```
NoSignatureForging : Message → GlobalState → Type
NoSignatureForging m s = Honest (m •pid) → m ∈ s .history
```

Local View of a Replica

More specific to the case of Streamlet; the essence of the protocol:

- local notion of seen **blockchains**
- local **state** of each replica
- **notarization** and **finalization**
- local **protocol behaviour** for each replica

Local View: state transition as an inductive relation

```
data _▷_⊢_–[_]→_ (p : Pid) (e : Epoch) (ls : LocalState) : Maybe Message → LocalState → Type
```

Local View: blockchains

```
Chain = List Block
```

```
record Block : Type where
  constructor ⟨_,_,_⟩
  field parentHash : Hash
        epoch      : Epoch
        payload    : List Transaction
```

Local View: blockchains

```
record _-connects-to-_ (b : Block) (ch : Chain) : Type where
  field hashesMatch   : b .parentHash ≡ ch ♯
        epochAdvances : b .epoch      > ch •epoch
```

Local View: blockchains

```
data ValidChain : Chain → Type where

  [] :                               _::_⊣_ : ∀ b →
                                       • ValidChain ch   • b -connects-to- ch
       ──────────────                ────────────────────────────────────────
       ValidChain []                   ValidChain (b :: ch)
```

Local View: state

```
record LocalState : Type where
  field phase : Phase           inbox : List Message
        db    : List Message    final : Chain
```

```
data Message : Type where
  Propose : SignedBlock → Message
  Vote    : SignedBlock → Message
```

Local View: notarization

```
votes : List Message → Block → List Message
votes ms b = filter (λ m → b ≟ m •block) ms
```

```
NotarizedBlock : List Message → Block → Type
NotarizedBlock ms b = IsMajority (votes ms b)
```

```
NotarizedChain : List Message → Chain → Type
NotarizedChain ms ch = All (NotarizedBlock ms) ch
```

Local View: notarization

```
data _chain-∈_ : Chain → List Message → Type where

  [] :                          _∷_⊣_ :
                                  • Any (λ m → b ≡ m ∙block) ms
                                  • ch chain-∈ ms• b -connects-to- ch
      ───────────────             ─────────────────────────────────
      [] chain-∈ ms               (b ∷ ch) chain-∈ ms
```

```
_notarized-chain-∈_ _longest-notarized-chain-∈_ : Chain → List Message → Type
ch notarized-chain-∈ ms =
  ch chain-∈ ms × NotarizedChain ms ch
ch longest-notarized-chain-∈ ms =
  ch notarized-chain-∈ ms ×
  (∀ {ch′} → ch′ notarized-chain-∈ ms → length ch′ ≤ length ch)
```

Local View: finalization

```
data FinalizedChain (ms : List Message) : Chain → Block → Type where
  Finalize :
    • NotarizedChain ms (b₃ ∷ b₂ ∷ b₁ ∷ ch)
    • b₃ .epoch ≡ suc (b₂ .epoch)
    • b₂ .epoch ≡ suc (b₁ .epoch)
    ──────────────────────────────────────
    FinalizedChain ms (b₂ ∷ b₁ ∷ ch) b₃
```

Local View: steps

```
data _▷_⊢_—[_]→_ p e ls where

  ProposeBlock :
    let L = epochLeader e
        b = ⟨ ch ♯ , e , txs ⟩
        m = Propose (sign p b)
    in

    • ls .phase ≡ Ready          • ch longest-notarized-chain-∈ ls .db
    • p ≡ L                      • ValidChain (b ∷ ch)
    ──────────────────────────────────────────────────────────────────
    p ▷ e ⊢ ls —[ just m ]→ record ls { phase = Voted; db = m ∷ ls .db }
```

Local View: steps

```
VoteBlock :
  let L   = epochLeader e
      b   = ⟨ ch ♯ , e , txs ⟩
      sbᴸ = sign L b
      mᴸ  = Propose sbᴸ ; m = Vote (sign p b)
  in
  ∀ (m∈ : mᴸ ∈¹ ls .inbox) →                  • p ≢ L
  • sbᴸ ∉ map _•signedBlock (ls .db)          • ch longest-notarized-chain-∈ ls .db
  • ls .phase ≡ Ready                         • ValidChain (b ∷ ch)
  ─────────────────────────────────────────────────────────────────────────────
  p ▷ e ⊢ ls —[ just m ]→ record ls { phase = Voted; db = m ∷ mᴸ ∷ ls .db; inbox = ls .inbox ─¹ m∈ }
```

Local View: steps

```
RegisterVote : let m = Vote sb in
  ∀ (m∈ : m ∈ ls .inbox) →
  • sb ∉ map _•signedBlock (ls .db)
  ────────────────────────────────────────────────────────────────
  p ▷ e ⊢ ls —[ nothing ]→ record ls { db = m :: ls .db; inbox = ls .inbox ─ m∈ }
```

Local View: steps

$$\texttt{FinalizeBlock} : \forall\, ch\; b \to$$

$$\frac{\bullet\ \texttt{ValidChain}\ (b :: ch) \quad \bullet\ \texttt{FinalizedChain}\ (ls\ .\texttt{db})\ ch\ b}{p \triangleright e \vdash ls \ \texttt{-[}\ \texttt{nothing}\ \texttt{]}\!\!\to \texttt{record}\ ls\ \{\ \texttt{final} = ch\ \}}$$

Mechanizing consistency: statement

```
Consistency : StateProperty
Consistency s = ∀ {p p′ b ch ch′} ⦃ _ : Honest p ⦄ ⦃ _ : Honest p′ ⦄ →
  let ms = (s @ p) .db ; ms′ = (s @ p′) .db in

  • (b ∷ ch) chain-∈ ms   • ch′ notarized-chain-∈ ms′
  • FinalizedChain ms ch b• length ch ≤ length ch′
  ────────────────────────────────────────────────
  ch ≼ ch′
```

Mechanizing consistency: closures as traces

```
data _↞_ : GlobalState → GlobalState → Type where
```

`_∎ : ∀ x →`

$$\frac{}{x \twoheadleftarrow x}$$

`_⟨_⟩←_ : ∀ z →`

$$\frac{\bullet\ z \leftarrow y \qquad \bullet\ y \twoheadleftarrow x}{z \twoheadleftarrow x}$$

Mechanizing consistency: invariants

```
StateProperty = GlobalState → Type
```

```
Reachable : StateProperty
Reachable s = s ↞ s₀
```

```
Invariant : StateProperty → Type
Invariant P = ∀{s} → Reachable s → P s
```

Mechanizing consistency: example invariant

```
HistorySound : StateProperty
HistorySound s = ∀ {p m} ⦃ _ : Honest p ⦄ →

  • p ≡ m ∙pid     • m ∈ s .history
  ────────────────────────────────────
  m ∈ (s @ p) .db
```

Mechanizing consistency: example invariant

```
historySound : Invariant HistorySound
historySound (s′ ⟨ s→ ∣ s ⟩← Rs) {p}{m} p≡ m∈
  with IH ← historySound Rs {p}{m} p≡
  with s→
  | DishonestStep _ replay
    with ⟫ m∈
... | ⟫ here refl rewrite p≡ = IH (replay it)
... | ⟫ there m∈              = IH m∈
  | LocalStep {p = p′}{mm}{ls′} ls→
      with ⟫ ls→
    ... | ⟫ ProposeBlock _ _ _ _
          with ⟫ m∈
    ...   | ⟫ here refl rewrite p≡ | lookup✓ = here refl
    ...   | ⟫ there m∈  with p ≟ p′
    ...                 | yes refl rewrite lookup✓     = there $ IH m∈
    ...                 | no p≢     rewrite lookup✖ p≢ = IH m∈
```

Mechanizing consistency: main lemmas

```
UniqueNotarization : StateProperty
UniqueNotarization s = ∀ {p p′ b b′} ⦃ _ : Honest p ⦄ ⦃ _ : Honest p′ ⦄ →
  let ms = (s @ p) .db ; ms′ = (s @ p′) .db in

  • NotarizedBlock ms  b  • NotarizedBlock ms′ b′ • b .epoch ≡ b′ .epoch
  ─────────────────────────────────────────────────────────────────────
  b ≡ b′
```

Mechanizing consistency: main lemmas

```
ConsistencyLemma : StateProperty
ConsistencyLemma s = ∀ {p p′ b₁ b₂ b ch ch′} ⦃ _ : Honest p ⦄ ⦃ _ : Honest p′ ⦄ →
  let ms = (s @ p) .db ; ms′ = (s @ p′) .db in
  • (b₂ ∷ b₁ ∷ ch) chain-∈ ms              • (b ∷ ch′) notarized-chain-∈ ms′
  • FinalizedChain ms (b₁ ∷ ch) b₂         • length ch′ ≡ length ch
  ────────────────────────────────────────────────────────────────────────────
  b₁ ≡ b
```

Mechanizing consistency: other (important) lemmas

```
IncreasingEpochs : StateProperty
IncreasingEpochs s = ∀ {p p′ p″ b ch b′ ch′} ⦃ _ : Honest p ⦄ ⦃ _ : Honest p′ ⦄ ⦃ _ : Honest p″ ⦄ →
  let ms = (s @ p) .db ; ms′ = (s @ p′) .db in

  • p″ ∈ voteIds ms b        • p″ ∈ voteIds ms′ b′      • length ch < length ch′
  • b -connects-to- ch       • b′ -connects-to- ch′
  ────────────────────────────────────────────────────────────────────────
  b .epoch < b′ .epoch
```

Mechanizing consistency: other (important) lemmas

```
MessageSharing : StateProperty
MessageSharing s = ∀ {p p′ b} ⦃ _ : Honest p ⦄ ⦃ _ : Honest p′ ⦄ →
  let ms = (s @ p) .db ; ms′ = (s @ p′) .db in
  p′ ∈ voteIds ms  b
  ─────────────────────
  p′ ∈ voteIds ms′ b
```


OK COMPUTER
RADIOHEAD
Lost Child.
Lost Child.

Testing: decidability proofs as decision procedures

```
data Dec (P : Type) : Type where
  yes : P   → Dec P
  no  : ¬ P → Dec P
```

```
record _⁇ (P : Type) : Type where
  field dec : Dec P

¿_¿ : ∀ P → ⦃ P ⁇ ⦄ → Dec P
¿ _ ¿ = dec
```

```
instance
  Dec-⊥ : ⊥ ⁇
  Dec-⊥ .dec = no λ()

  Dec-⊤ : ⊤ ⁇
  Dec-⊤ .dec = yes tt
```

```
module _ ⦃ _ : A ⁇ ⦄ ⦃ _ : B ⁇ ⦄ where instance
  Dec-→ : (A → B) ⁇
  Dec-→ .dec with ¿ A ¿ | ¿ B ¿
  ... | no ¬a | _     = yes λ a → contradict (¬a a)
  ... | yes a | yes b = yes λ _ → b
  ... | yes a | no ¬b = no λ f → ¬b (f a)

  Dec-× : (A × B) ⁇
  Dec-× .dec with ¿ A ¿ | ¿ B ¿
  ... | yes a | yes b = yes (a , b)
  ... | no ¬a | _     = no λ (a , _) → ¬a a
  ... | _     | no ¬b = no λ (_ , b) → ¬b b
```

Testing: decidability proofs as decision procedures

```
instance
  Dec-Finalized : ∀ {ms ch b} → FinalizedChain ms ch b ⁇
  Dec-Finalized {ch = ch} .dec
    with ch
  ... | []       = no λ ()
  ... | _ ∷ [] = no λ ()
  ... | _ ∷ _ ∷ _
    with dec | dec | dec
  ... | yes p | yes q | yes r = yes (Finalize p q r)
  ... | no ¬p | _     | _     = no λ where (Finalize p _ _) → ¬p p
  ... | _     | no ¬q | _     = no λ where (Finalize _ q _) → ¬q q
  ... | _     | _     | no ¬r = no λ where (Finalize _ _ r) → ¬r r
```

Testing: decidability proofs as decision procedures

```
Propose? : ∀ ch txs → let
    ...
    ls′ = proposeBlock ls m in
    ⦃ _ : p ≡ L ⦄
    {_ : auto: ls .phase ≡ Ready }
    {_ : auto: ch longest-notarized-chain-∈ ls .db }
    {_ : auto: ValidChain (b ∷ ch) } →
    ────────────────────────────────────────
    s → broadcast L (just m) (updateLocal p ls′ s)
```

Testing: example correct-by-construction traces

$$
\begin{tikzcd}
 & b_2 \arrow[dl] & b_5 \arrow[l] & b_6 ✓ \arrow[l] & b_7 \arrow[l] \\
\bot & & & & \\
 & b_1 \arrow[ul] & b_3 ✗ \arrow[l] & &
\end{tikzcd}
$$

Testing: example correct-by-construction traces

```
begin
  initGlobalState
→⟨ Propose? 𝕃 [] [] ⟩ -- leader proposes b₁
  record { e-now         = 1
         ; history       = [ p₁ ]
         ; networkBuffer = [ [ 𝔸 ∣ p₁ ⟩ ⨾ [ 𝔹 ∣ p₁ ⟩ ]
         ; stateMap      = [ {- 𝕃 -} ⦅ Voted , [ p₁ ] , [] , [] ⦆
                           ⨾ {- 𝔸 -} ⦅ Ready , []       , [] , [] ⦆
                           ⨾ {- 𝔹 -} ⦅ Ready , []       , [] , [] ⦆ ]}
→⟨ Deliver? [ 𝔹 ∣ p₁ ⟩ ⟩
  _
```

Testing: example correct-by-construction traces

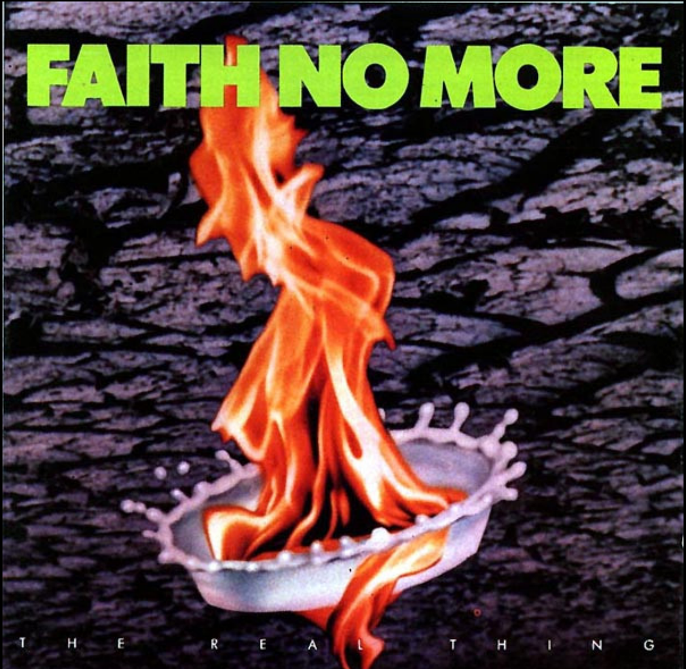
```
  ⋮
→⟨ Vote? 𝔹 [] [] ⟩ -- b₁ becomes notarized
 record { e-now         = 1
        ; history       = [ v₁ ⨾ p₁ ]
        ; networkBuffer = [ [ 𝔸 ∣ p₁ ⟩ ⨾ [ 𝕃 ∣ v₁ ⟩ ⨾ [ 𝔸 ∣ v₁ ⟩ ]
        ; stateMap      = [ ⦅ Voted , [ p₁ ]      , [] , [] ⦆
                          ⨾ ⦅ Ready , []          , [] , [] ⦆
                          ⨾ ⦅ Voted , [ v₁ ⨾ p₁ ] , [] , [] ⦆ ]}
   ⋮
```

Testing: example correct-by-construction traces

```
   ⋮
→⟨ Propose? 𝕃 [ b₆ ⨾ b₅ ⨾ b₂ ] [] ⟩ -- leader proposes b₇
    ⋮
→⟨ Vote? 𝔸 [ b₆ ⨾ b₅ ⨾ b₂ ] [] ⟩ -- b₇ becomes notarized
    ⋮
```

Testing: example correct-by-construction traces

```
  ⋮
→⟨ Finalize? 𝔸 [ b₆ ⨾ b₅ ⨾ b₂ ] b₇ ⟩ -- b₆ becomes finalized
 record { e-now         = 7
        ; history       = [ v₇ ⨾ p₇ ⨾ v₆ ⨾ p₆ ⨾ v₅ ⨾ p₅ ⨾ v₃ ⨾ p₃ ⨾ v₂ ⨾ p₂ ⨾ v₁ ⨾ p₁ ]
        ; networkBuffer = _
        ; stateMap      = [ ⦅ Voted , _ , [] , []                 ⦆
                          ⨾ ⦅ Voted , _ , [] , [ b₆ ⨾ b₅ ⨾ b₂ ] ⦆
                          ⨾ ⦅ Ready , _ , [] , []                 ⦆ ]}
∎
```


FAITH NO MORE
T H E R E A L T H I N G

Conformance testing: trace verification

```
data Action : Type where
  Propose       : Pid → Chain → List Transaction → Action
  Vote          : Pid → Chain → List Transaction → Action
  RegisterVote  : Pid → ℕ → Action
  FinalizeBlock : Pid → Chain → Block → Action
  DishonestStep : Pid → Message → Action
  Deliver       : ℕ → Action
  AdvanceEpoch  : Action

Actions = List Action
```

Conformance testing: trace verification

```
getLabels : (s —↠ s') → Actions
```

```
ValidTrace : Actions → GlobalState → Type
ValidTrace αs s = ∃ λ s' → ∃ λ (st : s —↠ s') → getLabels st ≡ αs
```

```
instance
  Dec-ValidTrace : ValidTrace ⁇²
```

Conformance testing: trace verification

```
⟦_⟧* : ValidTrace αs s → GlobalState
⟦ s′ , _ ⟧* = s′

ValidTrace-sound : (vas : ValidTrace αs s) → s —↠ ⟦ vas ⟧*
ValidTrace-sound (_ , s↠ , refl) = s↠

ValidTrace-complete : (st : s —↠ s′) → ValidTrace (getLabels st) s
ValidTrace-complete s↠ = -, s↠ , refl
```

Future Work

- Also prove the crucial property of liveness
  - $\rightarrow$ should be possible using the same methodology as consistency
- Apply our methodology to more complex realistic protocols, e.g. Jolteon
  - $\rightarrow$ in fact we have already done so (proved safety for Jolteon) and thus we're confident that our approach scales well

Conclusion

We've demonstrated a formalization of Streamlet, which is:

- mechanized in Agda to make sure there are no mistakes;
- presented in a readable fashion;
- also computable to leverage the formal model for conformance testing.

Questions?

https://github.com/input-output-hk/formal-streamlet