A readable and computable formalization of the Jolteon consensus protocol

Orestis Melkonian, Mauro Jaskelioff, James Chapman

12 June 2025, TYPES @ Glasgow

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

Mechanizing safety: closures as traces

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

  _■ : ∀ x →
      ───────
      x ↞— x

  _⟨_⟩←—_ : ∀ z →
      • z ←— y   • y ↞— x
      ────────────────────
      z ↞— x
```

```
Reachable : GlobalState → Type
Reachable s = s ↞— s₀
```

Mechanizing safety: statement

```
safety : ∀ {s} → Reachable s →
  • b  ∈ (s @ p)  .finalChain
  • b′ ∈ (s @ p′) .finalChain
    ────────────────────────────────
    (b ←—∗ b′) ⊎ (b′ ←—∗ b)
```

Mechanizing safety: lemma 3

```
lemma3 : ∀ {s} → Reachable s →
  • GloballyCertified s b′
  • b •round ≤ b′ •round
  • GloballyDirectCommitted s b
  ────────────────────────────────
  b ←—∗ b′
```

Mechanizing safety: quorum intersection

```
uniqueCertification : ∀ {s} → Reachable s →
  • GloballyCertified  s b
  • 1/3-HonestMajority s b′
  • b •round ≡ b′ •round
  ────────────────────────
  b ≡ b′
```

Mechanizing safety: history is complete

```
history-complete : ∀ {s} → Reachable s →
  (s @ p) .db ⊆ s .history
```

Mechanizing safety: history is complete

```
history-complete (_ , refl , (_ ■)) m∈ rewrite pLookup-replicate p initLocalState = contradict m∈
history-complete (_ , s-init , _ ⟨ st ∣ s ⟩← tr) m∈
  using Rs ← (_ , s-init , tr)
  using sm ← s .stateMap
  with  IH ← history-complete Rs
  with  IH-inbox ← inbox⊆history {p = p} Rs
  with  st
... | WaitUntil _ _ _ = IH m∈
... | Deliver {tpm} _ rewrite receiveMsg-db {s = sm} (honestTPMessage tpm) = IH m∈
... | DishonestLocalStep _ _ = there $ IH m∈
... | LocalStep {p = p′} {ls′ = ls′} st
  with  p ≟ p′
... | no p≢     rewrite pLookup∘updateAt′ p p′ {const ls′} (p≢ ∘ ↑-injective) sm = ∈-++⁺ʳ _ (IH m∈)
... | yes refl rewrite pLookup∘updateAt p ⦃ hp ⦄ {const ls′} sm
  with  st
... | InitNoTC _ _ = IH m∈
... | InitTC _ _    = there $ IH m∈
        ⋮
... | RegisterProposal m∈inbox _ _ _ _ = go
  where go : _; go with ⟫ m∈
        ... | ⟫ here refl = IH-inbox m∈inbox
        ... | ⟫ there m∈  = IH m∈
```


OK COMPUTER
RADIOHEAD
Lost Child.
Lost Child

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
```

Decidability proofs as decision procedures

```
instance
  Dec-certified-∈ : ∀ {b ms} → (b -certified-∈- ms) ⁇
  Dec-certified-∈ {b} {ms} .dec
    with ¿ Any (λ qc → (qc ∙blockId ≡ b ∙blockId) × (qc ∙round ≡ b ∙round)) (allQCs ms)
  ... | yes q = let (qc , qc∈all , (eqᵢ , eqᵣ)) = L.Mem.find q in
    yes $ certified (allQCs-sound ms qc∈all) eqᵢ eqᵣ
  ... | no ¬q = no λ where
    (certified {qc} qc∈ refl refl) →
      ¬q $ L.Any.map (λ x → cong proj₁ (sym x) , cong proj₂ (sym x))
                     (L.Any.map⁻ $ allQCs-complete ms qc∈)
```

Decidability proofs as decision procedures

```
_:RegisterProposal? : let m = _; b = sb .datum in
  {_ : auto: m ∈ ls .inbox}
  {_ : auto: ls .phase ≡ Receiving}
  {_ : auto: ¬ timedOut ls t}
  {_ : auto: sb .node ≡ roundLeader (b •round)}
  {_ : auto: ValidProposal (ls .db) b}
  → s ⟶ _
_:RegisterProposal? {_}{x}{y}{z}{w}{q} = LocalStep $'
  RegisterProposal (toWitness x) (toWitness y) (toWitness z)
                   (toWitness w) (toWitness q)
```

Example correct-by-construction traces

```
begin
 record
 { currentTime = 10
 ; history     = [ v₂ 𝕃 ; v₂ 𝔸 ; p₂ ; v₁ 𝔸 ; v₁ 𝕃 ; p₁ ]
 ; networkBuffer = [ 10 , 𝕃 , v₂ 𝔸 ; 10 , 𝕃 , v₂ 𝕃 ]
 ; stateMap    =
 [ {- 𝕃 -} ⦅ 2 , 2 , qc₁ , nothing , Receiving , _ , [] , [] , just 20 , false , false ⦆
 ; {- 𝔸 -} ⦅ 2 , 2 , qc₁ , nothing , EnteringRound , _ , [] , [] , nothing , false , true ⦆
 ; {- 𝔹 -} ⦅ 0 , 1 , qc₀ , nothing , Voting , _ , _ , [] , just τ , false , false ⦆ ] }
—→⟨ 𝔹 :VoteBlock? b₁ ⟩
 record
 { currentTime = 10
 ; history     = v₁ 𝔹 :: _
 ; networkBuffer = [ 10 , 𝕃 , v₂ 𝔸 ; 10 , 𝕃 , v₂ 𝕃 ; 10 , 𝕃 , v₁ 𝔹 ]
 ; stateMap    =
 [ ⦅ 2 , 2 , qc₁ , nothing , Receiving , _ , [] , [] , just 20 , false , false ⦆
 ; ⦅ 2 , 2 , qc₁ , nothing , EnteringRound , _ , [] , [] , nothing , false , true ⦆
 ; ⦅ 1 , 1 , qc₀ , nothing , Receiving , _ , _ , [] , just τ , false , false ⦆ ] }
—→⟨ 𝔹 :RegisterProposal? ⟩
 record
 { currentTime = 10
 ; history     = _
 ; networkBuffer = _
```

Example correct-by-construction traces

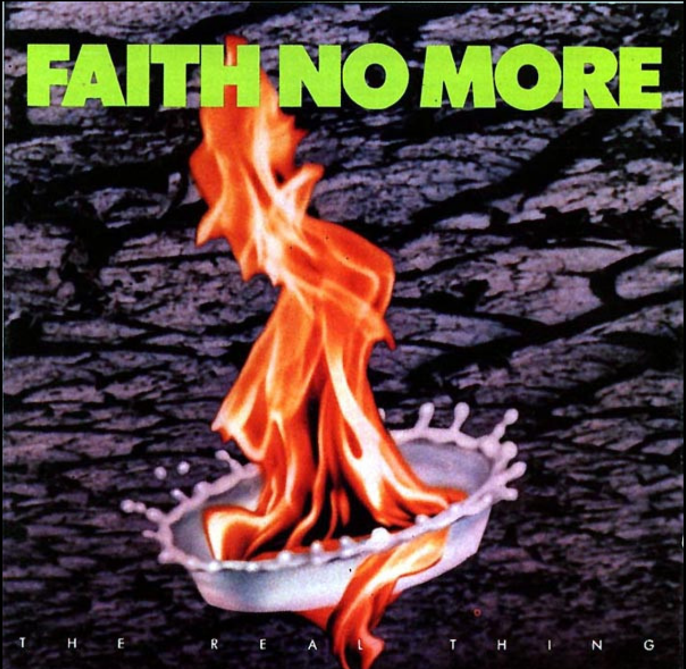
```
  ⋮
—→⟨ 𝕃 :RegisterVote? b₂ ⟩
 record
 { currentTime = 13
 ; history     = _
 ; networkBuffer = []
 ; stateMap    =
 [ ⟪ 2 , 2 , qc₁ , nothing , AdvancingRound , v₂ 𝔸 ∷ _ , v₂ 𝕃 ∷ _ , [] , just 20 , false , false ⟫
 ; ⟪ 2 , 2 , qc₁ , nothing , EnteringRound , [ p₂ ; p₁ ] , [] , [] , nothing , false , true ⟫
 ; ⟪ 2 , 2 , qc₁ , nothing , EnteringRound , [ p₂ ; p₁ ] , [] , [] , nothing , false , true ⟫ ] }
  ⋮
```

Example correct-by-construction traces

```
  ⋮
—→⟨ 𝕃 :RegisterVote? b₂ ⟩
 record
 { currentTime = 13
 ; history     = _
 ; networkBuffer = []
 ; stateMap    =
 [ ⦅ 2 , 2 , qc₁ , nothing , AdvancingRound , v₂ 𝕃 ∷ _ , _ , [] , just 20 , false , false ⦆
 ; ⦅ 2 , 2 , qc₁ , nothing , EnteringRound , _ , [] , [] , nothing , false , true ⦆
 ; ⦅ 2 , 2 , qc₁ , nothing , EnteringRound , _ , [] , [] , nothing , false , true ⦆ ] }
   ⋮
```

Example correct-by-construction traces

```
  ⋮
—→⟨ 𝕃 :Commit? [ b₂ ⨾ b₁ ] ⟩
 record
 { currentTime = 13
 ; history     = _
 ; networkBuffer = []
 ; stateMap    =
 [ ⦅ 2 , 3 , qc₂ , nothing , Voting , _ , _ , [ b₁ ] , nothing , false , true ⦆
 ⨾ ⦅ 2 , 2 , qc₁ , nothing , EnteringRound , _ , [] , [] , nothing , false , true ⦆
 ⨾ ⦅ 2 , 2 , qc₁ , nothing , EnteringRound , _ , [] , [] , nothing , false , true ⦆ ] }
∎
```


FAITH NO MORE
THE REAL THING

Conformance testing: trace verifier

```
data Action : Type where
  InitTC      : Action
  InitNoTC    : Action
  ProposeBlock : List Transaction → Action
      ⋮
  VoteBlock   : Block → Action
  Deliver     : Message → Action
  WaitUntil   : Time → Action
```

Conformance testing: trace verifier

```
ValidTrace : List Action → GlobalState → Type
ValidTrace αs s = ∃ λ s′ → s —[ αs ]↠ s′
```

```
⟦_⟧ : ValidTrace αs s → GlobalState
⟦_⟧ = proj₁

ValidTrace-sound : (vαs : ValidTrace αs s) → s —[ αs ]↠ ⟦ vαs ⟧
ValidTrace-sound = proj₂

ValidTrace-complete : s —[ αs ]↠ s′ → ValidTrace αs s
ValidTrace-complete = -,_
```

```
instance
  Dec-ValidTrace : ∀ {αs s} → ValidTrace αs s ⁇
```

Conclusion

We've demonstrated a formalization of Jolteon, 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.

Conclusion

We've demonstrated a formalization of Jolteon, 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.

WIP

- closing in on a liveness proof
  - significantly less straightforward than safety...
- integrating trace verifier to prototype Rust implementation with nice errors, *etc.*

¿¿¿ Questions ¿¿¿

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

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