Towards Formal Verification of HotStuff-based Byzantine Fault Tolerant Consensus in Agda

Presented By: Jake Ginesin

Background 1: What is Agda?

Agda is...

- Dependent type-based (like Coq, Lean)
- Similar to Haskell. Type theory ppl go crazy over it.
- A programming language *and* proof assistant
- ... has no (built in) tactics[1]

1: Thinking about tactics in Agda

Background 2: What is Hotstuff?

HotStuff: Leader-based Log Replication with more:

- Assumes *partially synchronous* communication
- Concerned with authenticator complexity
- Makes use of crypto, quorum certificates
- Epoch changes (i.e. voting to change the voters)

Q1: thoughts on Epoch Changes? Seems sus to me.
Q2: is partially synchronous communication realistic?

Goals of this paper

- Formal Assumptions
- Formal (Abstracted) Model
- Machine-Checked Proofs
- Towards a verified Haskell implementation (hence Agda)[1]

Q3: Why Haskell over other langs? Is the better interface with the formally verified code even worth?

1: Seems the main implementation uses C++.

Verifying a *subset* of HotStuff

This paper seeks reason about:

- Peer agreement on blocks
- Voting, proposals, quorum certificates
- Epoch changes

Given the assumptions:

- Honest peers can’t make contradicting additions (informal?)
- Honest peers only vote once per round
- Honest peers don’t vote ahead

Q4: was the first assumption formalized?

Model Parameters

```
module LibraBFT.Abstract.Properties
  (UID     : Set)
  (_≟UID_ : (u₀ u₁ : UID) → Dec (u₀ ≡ u₁))
  (NodeId : Set)
  (ℰ      : EpochConfig UID NodeId)
  (𝒱      : VoteEvidence UID NodeId ℰ)
  where
```

- Authors provide an (Agda) implementation, which instantiates this object
- “can be instantiated in order to relate a particular implementation to the abstract machinery”

Q5: Does this affect the proofs? Could I instantiate a config that violates the proofs?
Q6: Are the proofs written such that they cover all cases of the model parameters?

An Example Assumption

$$bft\text{-}assumption : \forall \{xs\ ys\} \rightarrow IsQuorum\ xs \rightarrow IsQuorum\ ys \rightarrow \exists[a](a \in xs \times a \in ys \times MetaHonestPK\ (getPubKey\ a))$$

- ***IsQuorum***, ***MetaHonestPK*** provided by EpochConfig

Data Structures: Record Chains

- Simple Block and Vote objects

```
record Block : Set where
  constructor mkBlock
  field bRound   : Round
        bId      : UID
        bPrevQC  : Maybe UID
```

```
record Vote : Set where
  constructor mkVote
  field vRound     : Round
        vMember    : Member
        vBlockUID  : UID
```

Q7: What is *maybe*? It seems it is a type? (Liquid Haskell?)

Data Structures: Record Chains

- Quorum Certificate Object

```
record QC : Set where
  constructor mkQC
  field qRound       : Round
        qCertBlockId : UID
        qVotes       : List Vote
        qVotes-C1    : IsQuorum (List-map vMember qVotes)
        qVotes-C2    : All (λ v → vBlockUID v ≡ qCertBlockId) qVotes
        qVotes-C3    : All (λ v → vRound v    ≡ qRound)       qVotes
        qVotes-C4    : All V qVotes
```

Q7: How are Quorum Certificates produced cryptographically in practice?

Data Structures: Record Chains

- Records are either a genesis block, a normal block, or a quorum signature
- Record types extend each other in different ways

```
data Record : Set where
  I :                Record
  B : Block  →       Record
  Q : QC     →       Record
```

```
data _←_ : Record → Record → Set where
  I←B : ∀ {b} → 0 < getRound b → bPrevQC b ≡ nothing
        → I ← (B b)
  Q←B : ∀ {q b} → getRound q < getRound b
        → just (qCertBlockId q) ≡ bPrevQC b
        → Q q ← B b
  B←Q : ∀ {b q} → getRound q ≡ getRound b → bId b ≡ qCertBlockId q
        → B b ← Q q
```

Data Structures: Record Chains

- RecordChainFrom recursively gathers previous records into a chain

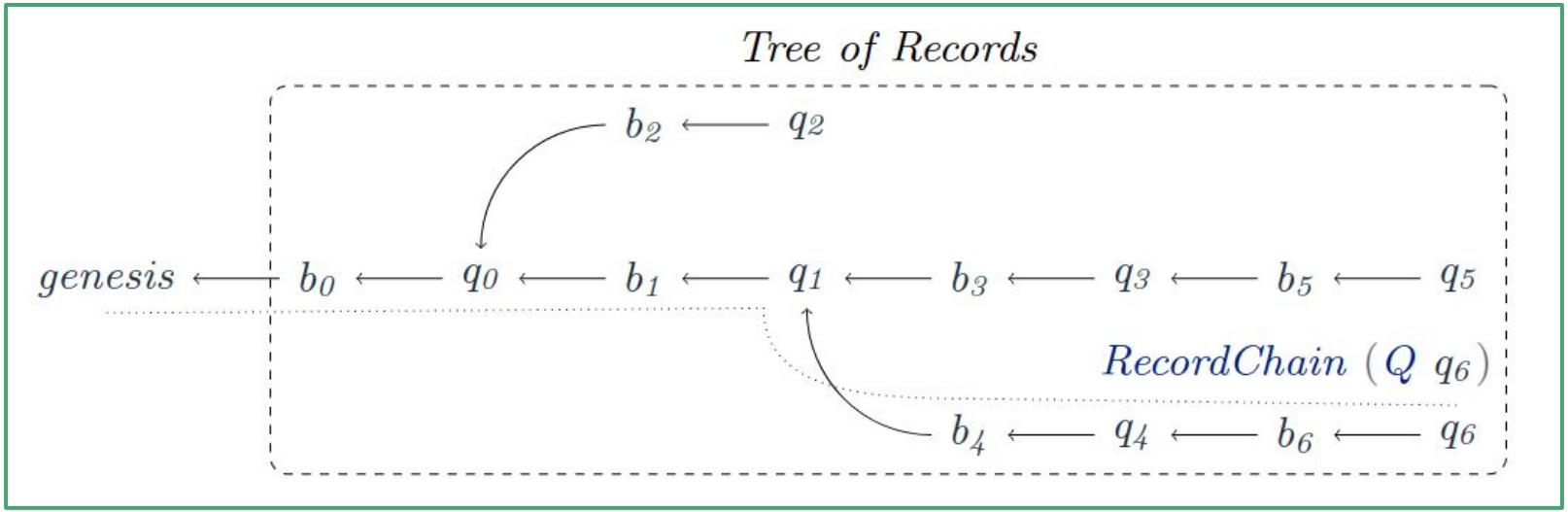
```
data RecordChainFrom (o : Record) : Record → Set where
  empty : RecordChainFrom o o
  step  : ∀ {r r′} → RecordChainFrom o r
        → r ← r′
        → RecordChainFrom o r′

RecordChain : Record → Set
RecordChain = RecordChainFrom I
```

Data Structures: Record Chains

- K-chain is a k-long list of records!

```
data 𝕂-chain (R : ℕ → Record → Record → Set)
  : (k : ℕ) {o r : Record} → RecordChainFrom o r → Set where
  0-chain : ∀ {o r} {rc : RecordChainFrom o r} → 𝕂-chain R 0 rc
  s-chain : ∀ {k o r} {rc : RecordChainFrom o r} {b : Block} {q : QC}
          → (r←b : r ← B b) → (prf : R k r (B b))
          → (b←q : B b ← Q q) → 𝕂-chain R k rc
          → 𝕂-chain R (suc k) (step (step rc r←b) b←q)
```

Data Structures: Record Chains



- Picture is meant to show relationship between hashes & blocks
- We generally won't have this branching behavior

Formalized Assumptions

- All honest members only vote once per round

$$
\begin{aligned}
&VotesOnlyOnceRule\ :\ Set\ \ell\\
&VotesOnlyOnceRule\ =\ (a\ :\ Member)\ \rightarrow\ MetaHonestMember\ a\\
&\qquad\rightarrow\ \forall\ \{q\ q'\}\ \rightarrow\ InSys\ (Q\ q)\ \rightarrow\ InSys\ (Q\ q')\\
&\qquad\rightarrow\ (v\ :\ a\ \in QC\ \ q)\ (v'\ :\ a\ \in QC\ \ q')\\
&\qquad\rightarrow\ vRound\ (\in QC\text{-}Vote\ q\ v)\ \equiv\ vRound\ (\in QC\text{-}Vote\ q'\ v')\\
&\qquad\rightarrow\ \in QC\text{-}Vote\ q\ v\ \equiv\ \in QC\text{-}Vote\ q'\ v'
\end{aligned}
$$

Formalized Assumptions

- Honest peers don’t vote ahead

$$
\begin{aligned}
&\mathit{PreferredRoundRule} \;:\; \mathit{Set}\ \ell \\
&\mathit{PreferredRoundRule} \\
&\quad = \forall\ a\ \{q\ q'\} \to \mathit{MetaHonestMember}\ a \to \mathit{InSys}\ (Q\ q) \to \mathit{InSys}\ (Q\ q') \\
&\quad \to \{rc : \mathit{RecordChain}\ (Q\ q)\}\ \{n : \mathbb{N}\} \to (c_3 : \mathbb{K}\text{-}\mathit{chain}\ \mathit{Contig}\ (3 + n)\ rc) \\
&\quad \to (v : a\ \in\!QC\ \ q)\ (rc' : \mathit{RecordChain}\ (Q\ q'))\ (v' : a\ \in\!QC\ \ q') \\
&\quad \to \mathit{vRound}\ (\in\!QC\text{-}\mathit{Vote}\ q\ v) < \mathit{vRound}\ (\in\!QC\text{-}\mathit{Vote}\ q'\ v') \\
&\quad \to \mathit{Either}\ \mathit{NonInjective}\text{-}{\equiv} \\
&\qquad\quad (\mathit{getRound}\ (\mathit{kchainBlock}\ (\mathit{suc}\ (\mathit{suc}\ \mathit{zero}))\ c_3) \leqslant \mathit{prevRound}\ rc')
\end{aligned}
$$

(Not Mentioned) Formalized Assumptions

- No Hash Collisions

```
51  -- We say an InSys predicate has NoCollisions if there are no two different Blocks that satisfy
52  -- InSys and have different ids.
53  NoCollisions : ∀{ℓ} → (Record → Set ℓ) → Set ℓ
54  NoCollisions ∈sys = ∀ {b₀ b₁} → ∈sys (B b₀) → ∈sys (B b₁) → bId b₀ ≡ bId b₁ → b₀ ≡ b₁
55
```

Correctness Prop #1: thmS5

- Ensures one block is in the record chain of the other
- (Implies no branching paths)

```
data CommitRuleFrom {o r : Record}
                    (rc : RecordChainFrom o r) (b : Block) : Set where
  commit-rule : (c₃ : 𝕂-chain Contig 3 rc) → b ≡ kchainBlock 2 c₃
              → CommitRuleFrom rc b
```

```
thmS5 : ∀ {q q'} → {rc : RecordChain (Q q)} → AllInSys rc
      → {rc' : RecordChain (Q q')} → AllInSys rc'
      → {b b' : Block} → CommitRule rc b → CommitRule rc' b'
      → Either NonInjective-≡ (Either ((B b) ∈RC rc') ((B b') ∈RC rc))
```

Correctness Prop #2: lemmaS2

- Ensures there is at most one verified block added per round

$$
\begin{aligned}
lemmaS2 \; : \; & \forall \, \{b_0 \; b_1 \; : \; Block\} \, \{q_0 \; q_1 \; : \; QC\} \; \rightarrow \; InSys \; (Q \; q_0) \; \rightarrow \; InSys \; (Q \; q_1) \\
& \rightarrow \; (p_0 \; : \; B \; b_0 \leftarrow Q \; q_0) \, (p_1 \; : \; B \; b_1 \leftarrow Q \; q_1) \\
& \rightarrow \; getRound \; b_0 \; \equiv \; getRound \; b_1 \\
& \rightarrow \; Either \; NonInjective\text{-}\equiv \; (b_0 \; \equiv \; b_1)
\end{aligned}
$$

Correctness Prop #3: lemmaS3

- Makes PreferredRoundRule apply to quorum

$$
\begin{aligned}
&lemmaS3 : \forall \{r_2\ q'\}\{rc : RecordChain\ r_2\} \rightarrow InSys\ r_2 \\
&\quad \rightarrow (rc' : RecordChain\ (Q\ q')) \rightarrow InSys\ (Q\ q') \\
&\quad \rightarrow (c_3 : kchain\ Contig\ 3\ rc) \rightarrow round\ r_2 < getRound\ q' \\
&\quad \rightarrow Either\ NonInjective\text{-}{\equiv}\ (getRound\ (kchainBlock\ (suc\ (suc\ zero))\ c_3) \\
&\qquad\qquad \leqslant prevRound\ rc')
\end{aligned}
$$

Correctness Prop #4: propS4

- A “non-symmetric variant” of thmS5(?)

$$
\begin{aligned}
propS4 \;:\; & \forall\,\{q\ q'\}\,\{rc \;:\; RecordChain\ (Q\ q)\} \;\to\; AllInSys\ rc \\
& \to\; (rc' \;:\; RecordChain\ (Q\ q')) \;\to\; AllInSys\ rc' \\
& \to\; (c_3 \;:\; \mathbb{K}\text{-}chain\ Contig\ 3\ rc) \\
& \to\; getRound\ (kchainBlock\ (suc\ (suc\ zero))\ c_3) \;\leqslant\; getRound\ q' \\
& \to\; Either\ NonInjective\text{-}{\equiv}\ (B\ (kchainBlock\ (suc\ (suc\ zero))\ c_3)\ {\in}RC\ \ rc')
\end{aligned}
$$

Correctness Prop #5: CommitsDoNotConflict' (thmS5++)

- A variant of thmS5 that ensures parties that don't participate in consensus still don't confirm conflicting commits

$$
\begin{aligned}
&\mathit{CommitsDoNotConflict'} : \forall \{o\ o'\ q\ q'\} \\
&\quad \to \{rcf : \mathit{RecordChainFrom}\ o\ (Q\ q)\} \to \mathit{AllInSys}\ rcf \\
&\quad \to \{rc' : \mathit{RecordChainFrom}\ o'\ (Q\ q')\} \to \mathit{AllInSys}\ rc' \\
&\quad \to \{b\ b' : \mathit{Block}\} \to \mathit{CommitRuleFrom}\ rcf\ b \to \mathit{CommitRuleFrom}\ rcf'\ b' \\
&\quad \to \mathit{Either}\ \Sigma\ (\mathit{RecordChain}\ (Q\ q'))\ ((B\ b) \in RC\_) \\
&\qquad\qquad \Sigma\ (\mathit{RecordChain}\ (Q\ q))\ ((B\ b') \in RC\_)
\end{aligned}
$$

Any Concluding Thoughts?