A Readable and Computable Formalization of the Streamlet Consensus Protocol

Mauro Jaskelioff
Input Output, Rosario, Argentina

Orestis Melkonian
Input Output, London, UK

James Chapman
Input Output, London, UK

Abstract

Consensus protocols are the fundamental building block of blockchain technology. Hence, correctness of the consensus protocol is essential for the construction of a reliable system. In the past few years, we saw the introduction of a myriad of new protocols of the BFT family of consensus protocols. The Streamlet protocol is one of these new protocols, which while not the fastest, it is certainly the simplest one.

In order to have strong guarantees for the protocol and its implementations we want to obtain formalizations that are readable enough to be used to communicate between formalizers and implementors, that have a mechanized proof of correctness and that can support the testing of implementations.

We present a readable and computable formalization of the Streamlet protocol in Agda, provide a mechanization of its proof of consistency, and show how one may use the formalization for testing implementations of it.

2012 ACM Subject Classification Theory of computation → Type theory; Theory of computation → Logic and verification; Theory of computation → Program specifications

Keywords and phrases blockchain, Streamlet, consensus, formal verification, Agda

Digital Object Identifier 10.4230/OASIcs.FMBC.2025.7

Supplementary Material *Software (Zenodo archive)*: `https://doi.org/10.5281/zenodo.15101644` [20]
Software (Source Code): `https://github.com/input-output-hk/formal-streamlet` [19]
archived at `swh:1:dir:70b9f1e274a05bad6f0e9fd5fe4e0f70033f503f`

1 Introduction

Consensus protocols are the fundamental building block of blockchain technology. Any mistake in their design or implementation could result in huge losses. Therefore, it is imperative to provide as strong guarantees as possible to ensure their correctness.

Consensus protocols can be *permissioned* or *permissionless*. Nakamoto-style consensus protocols are permissionless (all participants can be part of the decision process), while classical protocols like BFT [18] are permissioned (a few designated ones make the decision). With the advent of proof-of-stake blockchains, permissioned protocols can be adapted to work in a blockchain setting: a committee is formed based on the stake of all participants, which makes all decisions until a new committee is designated.

Consensus protocols have been around for a long time [18, 17]. In the past few years, we saw the introduction of a myriad of new consensus protocols of the BFT family [8, 28, 13, 6, 2, 11]. Given that many published consensus algorithms have been shown to be incorrect [3, 25], before adopting one of these new protocols, we would like to have strong guarantees that the protocol is correct and that we can thoroughly test its implementations. Therefore, we are interested in their formalization and mechanization of proof of correctness, as well as extracting computational content from the formalization in order to test implementations. Because the formalization is not only meant to be read by formal methods engineers, but also taken as a ground truth for implementors, another goal is for the formalization to be as readable as possible.

We conduct our work in a mechanized fashion using the Agda proof assistant [21]. Agda has a very flexible syntax that allows us to write readable specifications. Additionally, because Agda is based on constructive type theory, it is possible to use the specification to *compute* and obtain a testing mechanism.

In this work, we formalize the STREAMLET consensus protocol [7]; while not the fastest out of these new BFT-style protocols, it is certainly the simplest one. Its simplicity makes it the ideal candidate to start investigating approaches to our goal of obtaining a **readable** formalization (§2 and §3), a **mechanized proof** of correctness (§4), and a means for **testing** implementations (§5).

The formalization is public [20] and we provide hyperlinks (🐓) throughout the paper:

`https://input-output-hk.github.io/formal-streamlet/`

2 A general formal model for consensus protocols

Consensus protocols are protocols for distributed system, and therefore consist of several nodes, each with their local state, sending messages across a network. The protocol itself is described by the behavior of these nodes, but we need to model the complete system in order to state global properties, such as Consistency (§4) as global properties involve relations among the state of *different* nodes. Therefore in this section we present a rather general formalization of a complete system, with the specific behavior of the protocol abstracted away in a relation (the local-step relation) which describes how each node behaves.

The formalization is done in terms of a global-step relation whose only concerns are:

- describing how a local-step changes the local state of each node;
- describing what a dishonest node can do;
- modeling the network;
- modeling the passage of time.

These last two depend on the network model one uses, which might be asynchronous, synchronous, or partially-synchronous [12]. Streamlet, as most BFT protocols, rely on a partially-synchronous network. However this is not relevant for safety, which is our concern here, so we do not model message delays.

Following the Streamlet paper, we assume synchronized clocks and therefore only consider a discrete notion of time divided into *epochs*: $\mathsf{Epoch} = \mathbb{N}$.

The adversarial behavior is needed, since in the consensus protocols we are considering it is assumed that certain nodes might be dishonest and will actively try to disrupt the expected behavior of reaching consensus.

2.1 Assumptions [🐓 `Assumptions`]

First, we postulate the necessary cryptography, as it is completely orthogonal to our concerns:

- A type of Hashes and an ideal hash function $\#$ with no collisions; we assume we can compute hashes on base types and type formers, such as natural numbers ($\mathbb{N}$), products ($\times$), sums ($\uplus$), and lists (List).
- A type of Keys and Signatures, as well as a way to sign any data and verify signatures.

The assumptions pertaining to the specific setup of the consensus protocol are:

- A fixed number of participants (nodes : ℕ) each assigned a unique identifier Pid.
- A (decidable) predicate Honest : Pid → Type that distinguishes between honest and dishonest nodes. We will later use the notion of a vector that stores information for each *honest* node (HonestVec), since we do not want to keep local state for dishonest participants. For a given vector xs, xs @ p retrieves the state of honest participant p, while xs @ p := y locally updates the state of p.
- Crucially, all honest participants (honestPids) should form a 2/3-majority. Hence we assume honest-majority : 3 * length honestPids > 2 * nodes.
- Each epoch has a designated leader given by epochLeader : Epoch → Pid (the leader is chosen at random via a hash function, but there is no need to model that here).
- Last, transactions (Transaction : Type) that comprise a block are kept entirely abstract.

2.2 Global state [`Global.State`]

The global-step relates one global state to the next. A global state consists of a collection of local states (§3.2), one for each *honest* node.

```
StateMap = HonestVec LocalState
```

Other than that, it records the current epoch, in-transit messages, and the whole history of previous messages.

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

Dishonest nodes do no get a local state as we cannot assume anything about their state. Recording the history of messages is not needed to specify the behavior of honest nodes, but it has proven to be an invaluable tool for proving properties about it (§4). Furthermore, keeping the history is essential if we want the give adversaries the power to reuse and re-transmit signed messages sent in the past by honest participants. The network buffer is described in terms of an *envelope*: a pair of a message and its recipient. The initial global state starts at epoch 1 with no messages and initial local states.

2.3 The global-step relation [`Global.Step`]

The global-step is a relation between global states. It has a constructor for each of the concerns described in the previous section:

```
data _⟶_ (s : GlobalState) : GlobalState → Type where
  LocalStep : ⦃ _ : Honest p ⦄ →                  Deliver :
    (p ▷ s .e-now ⊢ s @ p —[ m? ]→ ls′)             (env∈ : env ∈ s .networkBuffer) →
    ─────────────────────────────────────           ─────────────────────────────────────
    s ⟶ broadcast p m? (s @ p := ls′)               s ⟶ deliverMsg s env∈

  DishonestStep :                                  AdvanceEpoch :
    • ¬ Honest p      • NoSignatureForging m s      ─────────────────────
    ─────────────────────────────────────────       s ⟶ advanceEpoch s
    s ⟶ broadcast p (just m) s
```

LocalStep delegates control to the local-step relation (§3.4) if an honest participant p makes a local step from the current global state s, optionally producing a message $m?$ and resulting in a new local state ls', then the whole system transitions to a new global state obtained by broadcasting the message $m?$ and updating the local state of p to ls'. In case there is a message, broadcast will add it to history, as well as place envelopes addressed to each other node into the networkBuffer.

The DishonestStep rule applies to dishonest nodes, who can broadcast any message as long as they do not forge signatures, *i.e.* messages signed by honest participants have to be replayed from history, while messages signed by dishonest participants have no restrictions:

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

The Deliver step takes any in-transit envelope and delivers it to its recipient. The new state after this step is obtained by removing the envelope from the network buffer and modifying the recipient's local state using the deliverMsg function. By design, this rule does not follow a queue order, allowing for messages to be delivered in an arbitrary order.

The AdvanceEpoch global step increments the current epoch (advanceEpoch) and notifies nodes to update their local state (epochChange, §3.2).

3 A formal model of the Streamlet consensus protocol

Onto our main object of study: the STREAMLET consensus protocol [7], aimed to provide an idealistic model for a recent class of protocols [28, 2, 13] that are geared towards the setting of proof-of-stake blockchains and follow a "streamlined" approach that does not require a distinction between happy path and fallback mode [16, 5], or single-shot consensus [18].

Since the generic scaffolding described in Section 2 applies to the case of STREAMLET, we only need to concern ourselves with the local behavior of a node.

Informal description. STREAMLET follows a very simple *propose-vote* paradigm: each epoch, a leader is elected and made responsible for *proposing* a new block, while honest nodes *vote* for these proposals. Once a block gets a majority of votes it becomes *notarized*, and any three adjacent notarized blocks *finalize* the chain up to the second block. Given that each node is only partially aware of the votes in the whole network, they each have their own perspective on which blocks are notarized and which chains they consider final.

Picking up our mechanization from Section 2, completing the protocol definition amounts to providing a specification of the local step used in the LocalStep rule of the global-step relation. But first, we have to define how blockchains are formed, the state information kept locally by honest nodes, as well as the precise definitions of notarization and finalization.

3.1 Blockchains [Local.Chain]

A *blockchain* consists of a sequence of blocks, where each *block* points to the hash of the block it extends, records its epoch, and carries a payload of transactions.

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

Participants will typically communicate blocks alongside their signature (SignedBlock).

Not all chains are valid though: for any block extending the previous chain, their hashes should match and epochs have to be strictly increasing.[1]

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

We express this inductively: starting from the empty blockchain, we extend it block-by-block, making sure the validity requirements are met.

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

3.2 Local state [Local.State]

Each node keeps track of a local view consisting of the following:

- its current *phase*, either Ready or Voted;
- an *inbox* of messages received from the network, but still not processed;
- a *database* of processed messages, as well as ones sent by this node;
- the (longest) blockchain this node considers *final*.

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

Initially, each node's state is empty and its phase set to Ready. Once a node proposes/votes a proposal, it sets its phase to Voted, which is reset to Ready at each epochChange.

The node's inbox is populated with messages externally via the global step's deliverMsg (§2). A *message* is either a proposal or a vote of a SignedBlock:

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

It is possible that messages appear out-of-order in the database, therefore we need to define when a node "has seen" a (valid) blockchain in their list of messages.

```
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
```

[1] A chain's epoch (accessed via function •epoch) is either the epoch of its most-recent block, or 0 for the empty "genesis" chain.

3.3 Finalization

Given a list of messages ms, we can now precisely specify when a block b is **notarized**: exactly when the nodes who have voted for this block form a majority (*i.e.* at least 2/3 of total participants).

```
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)
```

A blockchain is notarized when all of its constituent blocks are, while a block b_3 finalizes its prefix chain whenever three blocks (b_1, b_2, b_3 in chronological order) with consecutive epoch numbers have been notarized.

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

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
```

We will often care about a blockchain both occurring in a list of messages *and* being notarized, as well as being the longest one.

```
_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)
```

3.4 The local-step relation [Local.Step]

We are finally ready to formally specify the behavior of an *honest* node, as an inductively defined relation between said node p, the current epoch e, the starting state ls, possibly a message m, and the resulting state ls':

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

The participant, epoch, and starting state are promoted to *parameters*[2] as they remain constant across the possible actions of the node, while the rest of the relation's arguments are kept as *indices*[3] since they might vary across constructors of this datatype.

[2] https://agda.readthedocs.io/en/v2.7.0.1/language/data-types.html#parametrized-datatypes
[3] https://agda.readthedocs.io/en/v2.7.0.1/language/data-types.html#indexed-datatypes

The first rule models the proposals made by the epoch leader:

```
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 }
```

At the Ready phase, the leader can vote for a (valid) block extending the longest notarized chain in their view. The phase is updated to Voted to avoid double proposals, and the leader signed the proposed block and broadcasts it to the other nodes in a Propose message.

Other nodes instead follow the second rule, where they vote for proposals by the leader:

```
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∈ }
```

The hypotheses ensure that they vote for the *first* proposal they have seen, as long as it has not been registered in their database and is a valid extension to the longest blockchain in their view. The node also signs the voted block and broadcasts it via a Vote message. Again, the phase is updated accordingly to avoid duplicate votes.

While the previous two rules modeled the propose-vote paradigm employed by STREAMLET, the next rule facilitates the message exchange between nodes by providing the counterpart to the Deliver global step that populates inboxes:

```
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∈ }
```

Concretely, the node moves Vote messages from their inbox to their local database, as long as this vote has not been registered before (to avoid duplicates).

Finally, a node can finalize a valid chain they have seen thus far, as long as the finalization conditions of Section 3.3 are obeyed:

```
FinalizeBlock : ∀ ch b →
  • ValidChain (b ∷ ch)       • FinalizedChain (ls .db) ch b
  ─────────────────────────────────────────────────────────────
  p ▷ e ⊢ ls —[ nothing ]→ record ls { final = ch }
```

Et voila! Putting together these local node actions with the global step of Section 2, we now have a fully mechanized, readable, and complete specification of STREAMLET.

4 Mechanizing Streamlet's consistency proof [`Properties`]

A consensus protocol is safe if it maintains *consistency*. Consistency means that two honest nodes cannot have divergent chains: their corresponding finalized chains must always be a prefix of, or equal to the other. It is perfectly fine for a node to lag behind, in which case its final chain would be a prefix of another.

4.1 Formalizing consistency [`Consistency`]

We formalize the consistency property (*c.f.* [7, Theorem 3]) as a a predicate on GlobalStates.

```
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′
```

Here ms and ms' are the respective message databases of two honest nodes p and p'. Node p has finalized a chain ch and p' has seen a notarized chain ch' which is longer than ch. Consistency assures us that the finalized chain must be a prefix of or equal to ch'.

Further, we could prove how FinalizedChains correspond to the final fields of each node's state, but this is immediately derivable by inspecting the FinalizeBlock rule which makes sure only FinalizedChains are committed locally.

Also note that the Consistency property is slightly stronger than the informal description of consistency above, as we do not require the longer chain ch' to be part of a final chain; only notarization is required.

How do we prove consistency? We establish that the StateProperty is an *invariant*. That is, we prove that it holds for all states which are reachable from the initial one.

4.2 Proof infrastructure [`Global.Traces`]

We consider **traces** of the (global) step relation, defined as its *reflexive-transitive closure*.

```
data _*←_ : GlobalState → GlobalState → Type where
  _ ∎ : ∀ x →                  _⟨_⟩←—_ : ∀ z →
                                  • z ←— y   • y *← x
      ──────                   ────────────────────
      x *← x                      z *← x
```

A state property is a predicate on global states: StateProperty = GlobalState → Type. In general, we are only interested in global states that are reachable from the initial global state s_0, so one of the most useful state properties is reachability: Reachable $s = s \ast\!\leftarrow s_0$. A StateProperty is an invariant if it holds for every reachable global state.

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

4.3 Example proof [`Invariants.History`]

Let us consider the HistorySound property to illustrate how we can use the tools we just introduced. It states that all messages in history are actually sent by their sender, and therefore are in the sender's database of messages.

```
HistorySound : StateProperty
HistorySound s = ∀ {p m} ⦃ _ : Honest p ⦄ →
  • p ≡ m •pid        • m ∈ s .history
  ─────────────────────────────────────
  m ∈ (s @ p) .db
```

We prove that HistorySound is an Invariant, by induction on the reachability of the current state. The base case is trivially met, as history is empty in the initial state. In the case where a step $s{\to}$ is taken (transitioning from s to s'), we name IH the inductive hypothesis and do a case analysis on what kind of the step $s{\to}$ is.

```
historySound : Invariant HistorySound
historySound (s′ ⟨ s→ ∣ s ⟩←— Rs) {p}{m} p≡ m∈
  with IH ← historySound Rs {p}{m} p≡
  with s→
```

In the case of a step taken by a dishonest participant, we can use the inductive hypothesis, since the message necessarily has to be in history (m∈).[4]

```
  | DishonestStep _ replay
   with ≫ m∈
... | ≫ here refl rewrite p≡ = IH (replay it)
... | ≫ there m∈             = IH m∈
```

The most interesting case is when the step is a LocalStep. As is quite often, we need to consider whether the step is by the node p in question or by another node. In the former case, we rewrite with equality lookup✓ : $(s \mathbin{@} p := ls') \mathbin{@} p \equiv ls'$ to simplify the goal and continue reasoning about p's updated state ls'. In the latter case where p' is different than p, we instead rewrite with lookup× : $(s \mathbin{@} p' := ls') \mathbin{@} p \equiv s \mathbin{@} p$ and appeal to the induction hypothesis.

The proof of the LocalStep case proceeds by analyzing the four different cases for local step $ls{\to}$:

```
  | 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∈
```

We only show the case of ProposeBlock, as the other three cases are analogous. We also omit the cases of the global steps Deliver and AdvanceEpoch as they are trivial invocations of the inductive hypothesis, much like the case of DishonestStep.

[4] The use of singleton types (≫) is a technical artifact; it circumvents Agda's limitation to perform with-matching on a telescope variable.

4.4 Proving consistency [Consistency]

The proof of consistency, although it follows the informal paper proof [7], required some changes to the proof structure. The (strengthened) consensus property considers the case of a finalized chain ch and a notarized one ch', where the length of ch is less than or equal to the length of ch'. Because the longer chain can be shortened to the length of the shorter one, we can simplify consensus to the case where the two chains are of *equal length* (in the formalization, property ConsistencyEqualLen). Asking ch' to only be notarized is key in this reasoning, as any prefix of a notarized chain is notarized, while this is not the case for finalized chains, which require three consecutive epochs.

Having made this modification, we can follow the paper proof, which is based on two results: the ConsistencyLemma [7, Lemma 14] and UniqueNotarization [7, Lemma 10].

Unique notarization states that there can only be a unique notarization per epoch in honest view.

```
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′
```

The core of the consistency proof is the ConsistencyLemma. It states that if some honest node sees a notarized chain with three adjacent blocks b_0 , b_1 , b_2 with consecutive epoch numbers e, $e+1$, and $e+2$, then there cannot be a conflicting block $b \not\equiv b_1$ that also gets notarized in honest view at the same length as b_1.

```
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
```

As it often happens when formalizing a paper proof, many hidden details of the proof must be made apparent. An example of this is the IncreasingEpochs property which is a key to proving ConsistencyLemma, but left implicit in the paper proof. It states that honest nodes cannot vote for a block of a previous epoch, *i.e.* the epochs of blocks being voted is *monotonic*. In other words, honest participants never backtrack on their votes, *i.e.* if an honest participant p'' votes for a block b extending chain ch, but also votes for another block b' now extending a longer chain ch', then it must be the case that the epoch of b' is strictly greater than that of b.

```
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
```

where, voteIds ms b $=$ map _∙pid (votes ms b) computes the voters for block b in ms.

The use of the history field of GlobalState is essential for connecting local state properties across different nodes. For instance, we prove the general invariant of *message sharing*: if we find an honest vote in the database of another honest participant, then it is certainly also stored in the sender's database.

```
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
```

Its proof relies on properties like historySound (presented in Section 4.3) and its inverse historyComplete, which ensures every local database is included in history.

5 Testing

One of the most crucial reasons for conducting our work in *constructive* type theory is to be able to compute with our specification: proof assistants of this sort – Agda included – typically provide facilities to *extract* one's formalization to executable code.

While we have claimed to provide an *executable* specification of STREAMLET, we should clarify that this is only partly true due to the non-deterministic nature of the protocol. That is, the relational specification of Section 3 is *non-deterministic*, thus specifying a whole *set of implementations* that would be valid with respect to such a relation.

Furthermore, the assumptions made in Section 2.1 and left abstract for the rest of the formal development, should now be made concrete by instantiating all assumptions with actual implementations in order for extraction to executable code to make sense.

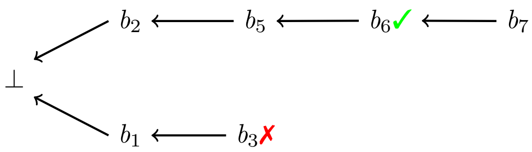
Therefore, we cannot hope to extract a full STREAMLET implementation out of our formal development, but there are still many constituent parts of our formalization that are indeed computable:

- We can prove that all of the logical propositions defined throughout the paper are indeed *decidable*. Proving that a proposition is decidable amounts to providing a *decision procedure* that answers whether the proposition holds or does not together a corresponding proof (§A).
- It is now possible to exhibit example traces of protocol execution without the need to explicitly discharge proof obligations for each rule invocation (§5.1). Traces manifest as proof derivations of the step relation, and all proof obligations are discharged by invoking the decision procedure that corresponds to each hypothesis, a technique known as *proof-by-computation* [27].
- Once extracted, the decision procedures enable us to test an actual implementation for *conformance* with respect to our mechanized semantics. To illustrate this point, we sketch a *trace verifier* that can validate traces randomly generated by an (unverified) implementation (§5.2).

5.1 Example trace [Test.ExampleTrace]

One crucial step to allow for computation and extraction is to provide a concrete instantiation of the assumptions (§2.1), otherwise computation would get stuck on encountering such a postulate. To do so amounts to giving a term of type Assumptions; we use naive hash functions and signature schemes and restrict to a set of three participants $\mathbb{L}$, $\mathbb{A}$, $\mathbb{B}$ where $\mathbb{L}$ is chosen as the leader at every epoch.

We will demonstrate an execution trace corresponding to the running example of the original Streamlet paper [7, Figure 1], where out of two competing chains $b_2 \leftarrow b_5 \leftarrow b_6 \leftarrow b_7$ and $b_2 \leftarrow b_3$ only the top one finalizes its prefix chain up to block b_6:



A block b_i is proposed on epoch i, thus the property of consistency mechanized in Section 4 makes it impossible for any extension of the bottom chain to be considered final anymore (due to the consecutive epochs of b_5, b_6, and b_7). The leaders makes proposals p_i for every block b_i and nodes vote for the same block with v_i, where $\mathbb{A}$ exclusively votes for the top chain and $\mathbb{B}$ for the bottom one. We are finally ready to make use of the proof automation of Appendix A to demonstrate an execution trace where b_6 eventually gets finalized:

```
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 ⟩ ⟩
  _
—→⟨ 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 ] , [] , [] ⦆ ]}
   ⋮
—→⟨ Propose? 𝕃 [ b6 ; b5 ; b2 ] [] ⟩  -- leader proposes b7
   ⋮
—→⟨ Vote? 𝔸 [ b6 ; b5 ; b2 ] [] ⟩      -- b7 becomes notarized
   ⋮
—→⟨ 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 , _ , [] , []                ⦆ ]}
◀
```

For the sake of brevity, we have elided many intermediate steps and states, but it should still be clear that the above demonstrates a provably correct derivation chain of steps, at the end of which node $\mathbb{A}$ has finalized the top chain up to b_6.

5.2 Conformance Testing [TraceVerifier]

The question remains: can we leverage the functions extracted from our STREAMLET mechanization in any other way outside the formalization itself?

We believe there is a strong case to be made for a **conformance testing** approach, where there already exists an implementation that is developed independently and is not formally verified, and we wish to ensure that it *conforms* to the formal specification. The central properties and invariants we have identified in Section 4 can inform the behavior being tested in the actual implementation. In particular, this seems to be an excellent fit to *property-based testing* [9], since the types of our theorems should easily translate to properties embedded in the implementation language.

This however relies on randomly generating traces of execution to feed as input to said tests. One possible way to bridge the gap between our Agda formal model of STREAMLET and its actual implementation is to extract a *trace verifier* that decides whether a trace generated by the implementation indeed respects the semantics of the global-step relation.

We first need to define a simple interface of actions, which will comprise the traces we communicate to external systems:

```
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
```

Actions provide the necessary input to make the rule selection *deterministic*: there is no ambiguity as to which rule applies at any given point. Equivalently, you can think of the action data being the same as the input we had to provide in the proof-automated steps of the example trace in Section 5.1.

Not all sequences of actions are valid though; we define a predicate that precisely characterizes the sequences that correspond to a valid trace: ValidTrace : Actions → Type, which relies on an evaluator ⟦_⟧ that executes a given action and returns the next state. We have omitted their definitions as they are just trivial repetitions of the rules: validity can be read off the rule hypotheses, while the next evaluated state can be read off each rule's conclusion.

We then provide a decision procedure to decide whether a sequence of actions is indeed valid, i.e. a *trace verifier*: instance Dec-ValidTrace : ∀ {*tr*} → ValidTrace *tr* ⁇. Although the correspondence between the trace verifier and the relational semantics of the previous sections is clear from the use of the same logical propositions, there is still no *formal* connection between them. We bridge this gap by proving the trace verifier *sound* and *complete* w.r.t. the global-step relation:

```
ValidTrace-sound :
  (tr : ValidTrace αs) →
  ⟦ tr ⟧ ∗← initGlobalState

ValidTrace-complete :
  (st : s ∗← initGlobalState) →
  ∃ λ (tr : ValidTrace (getLabels st)) →
    ⟦ tr ⟧ ≡ s
```

Soundness amounts to reconstructing a logical trace from a sequence of (valid) actions, while completeness ensures that all logical traces have a corresponding sequence of actions that results in the same state after execution.

6 Related Work

Given the importance of having strong guarantees for consensus protocols, it is no wonder that there are many formalizations of them; we are especially interested in ones that are conducted in an interactive proof assistant [24, 22, 1, 26, 4, 15]. However the objectives of each of these are slightly different, leading to different design choices.

Thomsen and Spitters [26] formalize a Nakamoto-style consensus algorithm (essentially Ouroboros Praos [10]) in Coq, and prove both safety and liveness. Their local state is based on an abstract block tree structure allowing for greater flexibility, while ours is a concrete list of messages received. This work inspired us to include history in the global state.

Carr *et al.* [4] formalize the LibraBFT protocol (which is based on Hotstuff [28]) in Agda and prove safety. Being a formalization of a BFT protocol in Agda, this work is closest to ours, however we had slightly different objectives, resulting in different approaches. Readability was not one of the main concerns so the model favors abstraction, allowing to potentially conclude safety from the properties of the instantiations of the abstract structures. Our model is more concrete and direct, but is more suitable for extracting a testing oracle.

Another line of research is concerned with generic frameworks for building consensus algorithms [14, 29, 23]; these however heavily rely on high-level abstractions, making it harder to relate a formalized protocol to the informal paper description. Here, we opt for a more direct approach.

7 Conclusion

We have presented our formalization of the BFT protocol STREAMLET using the Agda proof assistant. Using a relational approach for the step semantics, we have obtained a readable specification and proven consistency (safety). By implementing decision procedures we have made it possible to easily write verified traces of execution, and shown a path towards conformance testing.

References

1 Musab A. Alturki, Jing Chen, Victor Luchangco, Brandon M. Moore, Karl Palmskog, Lucas Peña, and Grigore Rosu. Towards a verified model of the Algorand consensus protocol in Coq. *CoRR*, abs/1907.05523, 2019. arXiv:1907.05523.

2 Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *CoRR*, abs/1710.09437, 2017. arXiv:1710.09437.

3 Christian Cachin and Marko Vukolic. Blockchain consensus protocols in the wild. *ArXiv*, abs/1707.01873, 2017. arXiv:1707.01873.

4 Harold Carr, Christa Jenkins, Mark Moir, Victor Cacciari Miraldo, and Lisandra Silva. Towards formal verification of HotStuff-based byzantine fault tolerant consensus in Agda. In Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez, editors, *NASA Formal Methods - 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24-27, 2022, Proceedings*, volume 13260 of *Lecture Notes in Computer Science*, pages 616–635. Springer, 2022. doi:10.1007/978-3-031-06773-0_33.

5 Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In Margo I. Seltzer and Paul J. Leach, editors, *Proceedings of the Third USENIX Symposium on Operating Systems*

Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999, pages 173–186. USENIX Association, 1999. URL: https://dl.acm.org/citation.cfm?id=296824.

6 Benjamin Y. Chan and Rafael Pass. Simplex consensus: A simple and fast consensus protocol. In Guy N. Rothblum and Hoeteck Wee, editors, *Theory of Cryptography - 21st International Conference, TCC 2023, Taipei, Taiwan, November 29 - December 2, 2023, Proceedings, Part IV*, volume 14372 of *Lecture Notes in Computer Science*, pages 452–479. Springer, 2023. doi:10.1007/978-3-031-48624-1_17.

7 Benjamin Y. Chan and Elaine Shi. Streamlet: Textbook streamlined blockchains. In *AFT '20: 2nd ACM Conference on Advances in Financial Technologies, New York, NY, USA, October 21-23, 2020*, pages 1–11. ACM, 2020. doi:10.1145/3419614.3423256.

8 T-H. Hubert Chan, Rafael Pass, and Elaine Shi. Pala: A simple partially synchronous blockchain. *IACR Cryptol. ePrint Arch.*, 2018:981, 2018. URL: https://api.semanticscholar.org/CorpusID:53238268.

9 Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of Haskell programs. *Acm sigplan notices*, 46(4):53–64, 2011.

10 Bernardo Machado David, Peter Gazi, Aggelos Kiayias, and Alexander Russell. Ouroboros Praos: An adaptively-secure, semi-synchronous proof-of-stake protocol. *IACR Cryptology ePrint Archive*, 2017:573, 2017. URL: http://eprint.iacr.org/2017/573.

11 Isaac Doidge, Raghavendra Ramesh, Nibesh Shrestha, and Joshua Tobkin. Moonshot: Optimizing chain-based rotating leader bft via optimistic proposals, 2024. doi:10.48550/arXiv.2401.01791.

12 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, April 1988. doi:10.1145/42282.42283.

13 Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. Jolteon and Ditto: Network-adaptive efficient consensus with asynchronous fallback. In Ittay Eyal and Juan A. Garay, editors, *Financial Cryptography and Data Security - 26th International Conference, FC 2022, Grenada, May 2-6, 2022, Revised Selected Papers*, volume 13411 of *Lecture Notes in Computer Science*, pages 296–315. Springer, 2022. doi:10.1007/978-3-031-18283-9_14.

14 Wolf Honoré, Longfei Qiu, Yoonseung Kim, Ji-Yong Shin, Jieung Kim, and Zhong Shao. AdoB: Bridging benign and byzantine consensus with atomic distributed objects. *Proc. ACM Program. Lang.*, 8(OOPSLA1), April 2024. doi:10.1145/3649826.

15 Elliot Jones and Diego Marmsoler. Towards Mechanised Consensus in Isabelle. In Bruno Bernardo and Diego Marmsoler, editors, *5th International Workshop on Formal Methods for Blockchains (FMBC 2024)*, volume 118 of *Open Access Series in Informatics (OASIcs)*, pages 4:1–4:22, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/OASIcs.FMBC.2024.4.

16 Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998. doi:10.1145/279227.279229.

17 Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, 2001.

18 Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982. doi:10.1145/357172.357176.

19 Orestis Melkonian and Mauro Jaskelioff. input-output-hk/formal-streamlet. Software, swhId: swh:1:dir:70b9f1e274a05bad6f0e9fd5fe4e0f70033f503f (visited on 2025-04-14). URL: https://github.com/input-output-hk/formal-streamlet, doi:10.4230/artifacts.23005.

20 Orestis Melkonian and Mauro Jaskelioff. Agda formalization of the Streamlet protocol. https://github.com/input-output-hk/formal-streamlet, March 2025. doi:10.5281/zenodo.15101644.

21 Ulf Norell. Dependently typed programming in Agda. In *International School on Advanced Functional Programming*, pages 230–266. Springer, 2008. doi:10.1007/978-3-642-04652-0_5.

22 George Pîrlea and Ilya Sergey. Mechanising blockchain consensus. In June Andronick and Amy P. Felty, editors, *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*, pages 78–90. ACM, 2018. doi:10.1145/3167086.

23 Longfei Qiu, Yoonseung Kim, Ji-Yong Shin, Jieung Kim, Wolf Honoré, and Zhong Shao. LiDO: Linearizable byzantine distributed objects with refinement-based liveness proofs. *Proc. ACM Program. Lang.*, 8(PLDI), June 2024. doi:10.1145/3656423.

24 Vincent Rahli, Ivana Vukotic, Marcus Völp, and Paulo Jorge Esteves Veríssimo. Velisarios: Byzantine fault-tolerant protocols powered by Coq. In Amal Ahmed, editor, *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, volume 10801 of *Lecture Notes in Computer Science*, pages 619–650. Springer, 2018. doi:10.1007/978-3-319-89884-1_22.

25 Pierre Tholoniat and Vincent Gramoli. Formal verification of blockchain byzantine fault tolerance. In *Handbook on Blockchain*, pages 389–412. Springer, 2022. doi:10.1007/978-3-031-07535-3_12.

26 Søren Eller Thomsen and Bas Spitters. Formalizing Nakamoto-style proof of stake. In *34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021*, pages 1–15. IEEE, 2021. doi:10.1109/CSF51468.2021.00042.

27 Paul Van Der Walt and Wouter Swierstra. Engineering proof by reflection in Agda. In *Symposium on Implementation and Application of Functional Languages*, pages 157–173. Springer, 2012. doi:10.1007/978-3-642-41582-1_10.

28 Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: BFT consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, pages 347–356, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3293611.3331591.

29 Qiyuan Zhao, George Pîrlea, Karolina Grzeszkiewicz, Seth Gilbert, and Ilya Sergey. Compositional verification of composite byzantine protocols. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, pages 34–48, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3658644.3690355.

A Decidability

[Decidability]

For any given proposition P, proving that it is decidable amounts to providing a program of type Dec P that decides whether the proposition holds (yes) or does not (no):

```
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
```

We collect all decidable propositions in a typeclass (⁇), and use the notation ¿ P ¿ to acquire the corresponding decision procedure by *instance search*.[5]

[5] https://agda.readthedocs.io/en/v2.7.0/language/instance-arguments.html

Decidability of basic types and type formers is already defined in the standard library:

```
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

  Dec-⊎ : (A ⊎ B) ⁇
  Dec-⊎ .dec with ¿ A ¿ | ¿ B ¿
  ... | yes a | _     = yes (inj₁ a)
  ... | _     | yes b = yes (inj₂ b)
  ... | no ¬a | no ¬b = no λ where (inj₁ a) → ¬a a; (inj₂ b) → ¬b b
```

Since these would take care of the most trivial combinations of other properties, we are only tasked with proving decidability of only the *interesting* propositions that we introduced in this paper that cannot be trivially solved by instance search.

Let us illustrate with the example of deciding whether a chain has been finalized:

```
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
```

We first check whether the chain in question does not even have three blocks, in which case we immediately decide the proposition does not hold. We then decide whether the finalization conditions of Section 3.3 hold and respond accordingly. Notice that we do not even have to state the propositions we are deciding in the process; the type system takes care of this for us!

Once all propositions that are explicitly or implicitly used in rule hypotheses have been proven decidable, we can provide an alternative version of the rules where the user no longer needs to provide explicit proofs in the case of *closed* examples (*i.e.* ones without any free variables). Instead, the corresponding decision procedures automatically discharge the proof obligations, otherwise we would get a typechecking error that the proposition under question is not true. Concretely, we prefix a proposition P with auto: to invoke its decision procedure; since computation will not block on any variables, we will eventual compute either a yes and replace the obligation with the trivial unit type ($\top$), or trigger an error by returning the absurd empty type ($\bot$) which can never be discharged.

```
auto:_ : (P : Type) → ⦃ P ⁇ ⦄ → Type
auto: P with ¿ P ¿
... | yes _ = ⊤
... | no  _ = ⊥
```

As an example, the ProposeBlock rule would remain mostly unchanged, except that all logical hypotheses are annotated as *implicit arguments*[6] and prefixed with auto: to trigger the aforementioned proof-by-computation.

```
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)
```

[6] `https://agda.readthedocs.io/en/v2.7.0/language/implicit-arguments.html`