Formalizing Correct-By-Construction Casper in Coq

No Author Given

Runtime Verification

1 Overview

We present a machine-checked formalization of the Correct-By-Construction Casper (CBC Casper) family of consensus protocols in the Coq proof assistant. We leverage Coq's type classes to represent CBC Casper using a hierarchy of type classes that derive increasingly stronger properties. In doing so, we 1) illuminate the assumptions required of each desired property, and 2) reformulate the protocol in terms of more general mathematical objects, namely partial orders with non-local confluence. We highlight two advantages of our approach: from a proof engineering perspective, it gives rise to a clear separation of concerns between theory and implementation; from a protocol engineering perspective, it provides us with a rigorous, foundational understanding which allows us to derive stronger properties about the protocol. We highlight one such property, namely strong non-triviality.

2 Type class hierarchy

The properties desired of CBC Casper broadly fall into two categories: 1) safety and 2) non-triviality. Definition of what these two things are. Some intuition on why these two things are important. We show below our incremental approach to proving these properties, starting from a definition of CBC Casper in terms of a state transition system with a reflexive and transitive reachability relation, also known in rewrite logic literature as a partial order and in modal logic literature as a KT4 Kripke model.

2.1 Partial order

```
Class PartialOrder :=
{ A : Type;
  A_eq_dec : forall (a1 a2 : A), {a1 = a2} + {a1 <> a2};
  A_inhabited : exists (a0 : A), True;
  A_rel : A -> A -> Prop;
  A_rel_refl :> Reflexive A_rel;
  A_rel_trans :> Transitive A_rel;
}.
```

At this level, we are able to derive all of the safety properties desired of CBC Casper, namely:

```
Theorem pair_common_futures '{CBC_protocol_eq}:
forall s1 s2 : pstate,
(equivocation_weight (state_union s1 s2) <= proj1_sig t)%R ->
exists s : pstate, pstate_rel s1 s /\ pstate_rel s2 s.

Theorem n_common_futures '{CBC_protocol_eq} :
forall ls : list pstate,
(equivocation_weight (fold_right state_union state0 (map (fun ps => proj1_sig ps) ls)) <=
exists ps : pstate, Forall (fun ps' => pstate_rel ps' ps) ls.

Theorem pair_consistency_prot '{CBC_protocol_eq} :
forall s1 s2 : pstate,
(equivocation_weight (state_union s1 s2) <= proj1_sig t)%R ->
forall P,
~ (decided P s1 /\ decided (not P) s2).

Theorem n_consistency_prot '{CBC_protocol_eq} :
forall ls : list pstate,
(equivocation_weight (fold_right state_union state0 (map (fun ps => proj1_sig ps) ls)) <=
state_consistency ls.

Theorem n_consistency_consensus '{CBC_protocol_eq} :
forall ls : list pstate,
(equivocation_weight (fold_right state_union state0 (map (fun ps => proj1_sig ps) ls)) <=
consensus_value_consistency ls.
```

These results correspond to Theorems X - Y in [4].

2.2 Partial order with non-local confluence

In order to prove non-triviality properties, we additionally require that our partial order possess certain confluence properties. In fact, we find that non-triviality as defined in [4] directly captures the notion of non-local confluence in state transition systems, defined abstractly as follows:

```
Class PartialOrderNonLCish '{PartialOrder} :=
{ no_local_confluence_ish : exists (a a1 a2 : A),
                            A_rel a a1 /\ A_rel a a2 /\
                            ~ exists (a' : A), A_rel a1 a' /\ A_rel a2 a';
}.
```

2.3 Abstract protocol

To provide a richer, protocol-specific language to describe our desired properties, we give an abstract type class from which we can generalize a partial order, but which contains information specific to consensus protocols, including types for validators, consensus values, states, and an abstract, total estimator function.

```
Class CBC_protocol_eq :=
{
consensus_values : Type;
about_consensus_values : StrictlyComparable consensus_values;
validators : Type;
about_validators : StrictlyComparable validators;
weight : validators -> {r | (r > 0)%R};
t : {r | (r >= 0)%R};
suff_val : exists vs, NoDup vs /\ ((fold_right (fun v r => (proj1_sig (weight v) + r)%R)
state : Type;
about_state : StrictlyComparable state;
state0 : state;
state_eq : state -> state -> Prop;
state_union : state -> state -> state;
state_union_comm : forall s1 s2, state_eq (state_union s1 s2) (state_union s2 s1);
reach : state -> state -> Prop;
reach_refl : forall s, reach s s;
reach_trans : forall s1 s2 s3, reach s1 s2 -> reach s2 s3 -> reach s1 s3;
reach_union : forall s1 s2, reach s1 (state_union s1 s2);
reach_morphism : forall s1 s2 s3, reach s1 s2 -> state_eq s2 s3 -> reach s1 s3;
E : state -> consensus_values -> Prop;
estimator_total : forall s, exists c, E s c;
prot_state : state -> Prop;
about_state0 : prot_state state0;
equivocation_weight : state -> R;
equivocation_weight_compat : forall s1 s2, (equivocation_weight s1 <= equivocation_weight
about_prot_state : forall s1 s2, prot_state s1 -> prot_state s2 ->
(equivocation_weight (state_union s1 s2) <= proj1_sig t)%R -> prot_state (state_union s1
}.
```

The plan here is to 1) say that our types directly correspond to CBC Casper's abstract parameters, and 2) briefly explain why the additional properties about each type are required.

We prove that `CBC_protocol_eq` can derive `PartialOrder`.

3 Full node

Emphasis on explicating:

1. least fixed point-based state definitions
2. strong non-triviality proof: atomic equivocation construction, pivotal validator proof, recursive atomic equivocation construction, overall proof sketch

4 Light node

Emphasis on explicating:

1. why we no longer need least fixed point-based state definitions

2. to the extent that it differs from the full node version, the strong non-triviality proof: atomic equivocation construction, pivotal validator proof, recursive atomic equivocation construction, overall proof sketch

5 Related work and discussion

To include:

1. Compare/contrast with [3]
2. Areas for further refinement of abstraction
3. Possibility of fitting other consensus protocols into our abstract type class hierarchy

6 Conclusion

References

1. Thierry Coquand, Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76(2-3):95–120, 1988.
2. The Coq Proof Assistant. https://coq.inria.fr/.
3. Ryuya Nakamura, Takayuki Jimba, Dominik Harz. Refinement and Verification of CBC Casper. *Cryptology ePrint Archive*, Report 2019/415, 2019.
4. Vlad Zamfir, Nate Rush, Aditya Asgaonkar, Georgios Piliouras. Introducing the "Minimal CBC Casper" Family of Consensus Protocols. 2019.
5. Vlad Zamfir Introducing the "Minimal" CBC Casper Light Client – with some explorations. 2019.