An approach to translating Haskell programs to Agda and reasoning about them

Harold Carr[1], Christa Jenkins[0000−0002−5434−5018][2], Mark Moir[3], Victor Cacciari Miraldo[4], and Lisandra Silva[5]

[1] Oracle Labs, USA, harold.carr@oracle.com
[2] University of Iowa, USA, cwjenkins@uiowa.edu
[3] Oracle Labs, New Zealand, mark.moir@oracle.com
[4] Tweag, The Netherlands, victor.miraldo@tweag.io
[5] Runtime Verification, USA, lisandra.silva@runtimeverification.com

Abstract. We are using the Agda programming language and proof assistant to formally verify the correctness of a Byzantine Fault Tolerant consensus implementation based on HotStuff / LibraBFT. The Agda implementation is a translation of our Haskell implementation based on LibraBFT. This paper focuses on one aspect of this work.
We have developed a library that enables the translated Agda implementation to closely mirror the Haskell code on which it is based. This makes it easier and more efficient to review the translation for accuracy, and to maintain the translated Agda code when the Haskell code changes, thereby reducing the risk of translation errors. We also explain how we capture the semantics of the syntactic features provided by our library, thus enabling formal reasoning about programs that use them; details of how we reason about the resulting Agda implementation will be presented in a future paper.
The library that we present is independent of our particular verification project, and is available, open-source, for others to use and extend.

Keywords: formal verification, Agda, Haskell, weakest precondition, Dijkstra monad

1 Introduction

Due to attractive properties relative to previous Byzantine Fault Tolerant (BFT) consensus protocols, implementations based on HotStuff [34] are being developed and adopted. For example, the Diem Foundation (formerly Libra Association) was until recently developing LibraBFT based on HotStuff [5, 32]. (LibraBFT was renamed to DiemBFT before being discontinued; other variants are emerging.)

It is notoriously difficult to build distributed systems that are *correct*, especially if byzantine faults [21] may occur, that is, some participants may *actively* and *maliciously* deviate from the protocol. Many published consensus algorithms—including some with manual correctness proofs—have been shown

to be incorrect [9, 33], meaning that two honest participants can be convinced to accept conflicting decisions, even if all assumptions are satisfied. Therefore, precise, machine-checked formal verification is essential, particularly for new protocols that are being adopted in practice.

In this paper, we describe some aspects of our ongoing work towards verifying correctness of an Agda port of our Haskell implementation of BFT consensus, which is based on LibraBFT. Our initial focus is on proving safety properties for a single "epoch", during which participating peers and protocol parameters do not change. We have reduced the required proof obligations to showing that the code executed by *honest* (non-byzantine) peers satisfies some precise assumptions [11, 12].

Translating Haskell code to equivalent Agda is often quite straightforward—requiring only minor syntactic changes—because Agda's syntax is based on Haskell's. However, Agda does not directly support all Haskell syntax and libraries. As a result, early versions of our Agda translation differed significantly from the Haskell code being modeled, making review and maintenance more difficult, and increasing the risk of inaccuracy.

This paper focuses on a library that we have developed in Agda to a) support various Haskell features that Agda lacks, thus enabling our Agda translation to closely track the Haskell code, and b) capture their semantics, enabling formal verification of properties about the code. As a result, we have been able to port our entire Haskell implementation to Agda code that, in the vast majority of cases, mirrors the Haskell code so closely that side-by-side review requires virtually no mental overhead in our experience.

Haskell features that our library supports for use in Agda include:

- comparison and conditionals based on decidable equality, which enables providing proofs with *evidence* of the relationship between two compared values for the particular case being proved;
- lenses for (nested) record field access and update
- monads for composing programs for various contexts
- monad instances, including for *Either*, *List* and *RWS* (Reader, Writer, State)

Our library also includes straightforward Agda implementations of various Haskell library functions that are not provided by Agda's standard library; see the *Haskell.Prelude* module in the accompanying open-source repository [7].

Having ported our Haskell implementation to Agda using our library, we are working on verifying that it ensures the properties that we have already proved are sufficient to establish correctness of the implementation [11, 12]. To enable reasoning about the behavior of code in the *Either* and *RWS* monads, our library provides a predicate transformer semantics for such code based on Dijkstra's weakest precondition calculus [4, 23, 28, 30]. Details of how our library supports this and how we use it are beyond the scope of this paper, but we do discuss how we assign semantics to monadic programs for *Either* and *RWS*.

Section 2 presents several examples that illustrate some of the syntactic features we have introduced, and briefly discusses some of the interesting aspects of implementing them. References are provided to enable the reader to locate

the details in our open-source development. In Section 3, we describe further extensions that establish the foundation for the machinery that we use to reason about monadic programs. Section 4 dicusses related work, some discussion is included in Section 5, and we conclude in Section 6.

This paper assumes that the reader is familiar with Haskell and has access to our open-source repository [7]; module *X.Y* can be found in `src/X/Y.agda`.

2 Making Agda look (even more) like Haskell

The following Haskell function is part of the implementation that we are verifying. We do not present type definitions or explain the purpose of the code, as we are interested only in syntax here.

```
verify :: BlockRetrievalResponse -> ... -> Either ErrLog ()
verify self ... =
  if  | self^.brpStatus /= BRSSucceeded -> Left ...
      | ...
      | otherwise -> verifyBlocks (self^.brpBlocks)
```

The corresponding function in Agda is:

```
verify : BlockRetrievalResponse → ... → Either ErrLog Unit
verify self ... =
  grd‖ self ^· brpStatus /= BRSSucceeded := Left ...
     ‖ ...
     ‖ otherwise := verifyBlocks (self ^· brpBlocks)
```

This example demonstrates several of the syntactic features that we added to our library to enable the Agda version to closely mirror the Haskell code.

Guarded conditionals Agda does not support Haskell guard syntax. Therefore, we defined equivalent syntax (with changes such as replacing "if |" with "grd‖" and "|" with "‖" to avoid conflicts with core Agda syntax); see module *Haskell.Prelude*.

Equality and comparison: To support the `==` and `/=` operators from Haskell's `Data.Eq` typeclass, we define an *Eq* record that provides the same operators in Agda (*Haskell.Modules.Eq*). However, to construct proofs in Agda, we need *evidence* that two values are or are not equal. Therefore, our *Eq* record actually contains only one field, $\_\stackrel{?}{=}\_$, which provides a method for deciding equality for the relevant type; `==` and `/=` are defined using it. Our library also contains an implementation of Haskell's *compare*, implemented via Agda's `<-cmp`, which provides *evidence* of the relationship between two values; see *Haskell.Prelude*.

Lenses: The Haskell code uses the Lens libray [18] for (nested) record field access and update. To enable the same in Agda, we developed an *Optics* library, which uses reflection to derive van Laarhoven lenses [20] for simple, non-dependent records. Because we are interested in translating code from Haskell, the records that we use are all non-dependent, and thus have van Laarhoven lenses.

Monads: Much of our Haskell code is monadic. For example, we use the *Either* monad [24] for sequencing and error handling.

Like Haskell, Agda supports monadic do-notation, and the Agda standard library comes with a definition of a monad as a record which, in combination with instance arguments, can be used to simulate Haskell's `Monad` typeclass. However, the Agda standard library defines `Monad` as having type $Set\ \ell \rightarrow Set\ \ell$ where ℓ is an arbitrary universe level [1]. In contrast, the types that we use to represent program ASTs (for example, the *EitherD* definition shown later) have type $Set \rightarrow Set1$ because some constructors quantify over *Set*s. We have therefore defined our own *Monad* record with *return* and `>>=` fields. We similarly define records for *Applicative* and *Functor*, enabling `_<*>_` and `_<$>_` operators, respectively, and functions from *Monad* to *Applicative* and from *Applicative* to *Functor*. These operators are made available in various contexts by defining Agda instances of the relevant monad (e.g., *Either* in the example above). An important side effect is providing a definition for `>>=`, which is how the semantics of $\leftarrow$ in a *do* block is defined in Agda. The next example illustrates all of this functionality in the context of the *RWS* monad [16], which combines Reader, Writer and State monads.

```
processProposalM : Block → RWS Unit Output RoundManager Unit
processProposalM proposal = do
  ...
  vp ← ProposerElection.isValidProposalM proposal
  grd‖ isNothing (proposal ^. bAuthor) := logErr ...
     ‖ ...
     ‖ otherwise:=
    (executeAndVoteM proposal >>= λ where
      (Left e) → logErr e
      (Right vote) → do
        RoundState.recordVote vote
        si ← BlockStore.syncInfoM
        recipient ← ProposerElection.getValidProposer
          <$> use lProposerElection
          <*> pure (proposal ^. bRound + 1)
        tell $ (SendVote (VoteMsg vote si) recipient)) :: []
```

In the above example:

- the Reader monad is not used (so the type that it can read is *Unit*);
- the Writer monad enables values (of type *Output*) to be written using *tell*;
- the State monad enables fetching, replacing and updating state of type *RoundManager* via functions *get*, *put* and *modify* (not shown), respectively, as well as accessing a nested field via a lens (e.g., *lProposerElection* above) with *use*.

The *processProposalM* function does not directly modify the State. However, it calls another *RWS* function *RoundState.recordVote*, which does:

```
recordVoteM : Vote → RWS Unit Output RoundManager Unit
recordVoteM v = rsVoteSent-rm ·= just v
```

Here, $\cdot=$ is syntax for *setL*; it sets a (possibly nested) field of the State via a lens (*rsVoteSent-rm*). It is implemented using *RWS*'s *get* and *put* operations.

3 Support for reasoning about monadic programs

To reason about effects, we equip our monadic code with a predicate transformer semantics based on Dijkstra's weakest precondition calculus [31] (see also *Dijkstra monads* [4, 23, 28, 30]). This enables automatically calculating weakest preconditions for desired postconditions. Figure 1 illustrates using *Either E*, the monad for code that may throw exceptions of type *E*.

The type *EitherD E* enables expressing the AST of code that may throw errors of type *E* (see *Dijkstra.EitherD*, and *Dijkstra.EitherD.Syntax* for its monad instance) in a way that enables connecting it to its semantics for verification, as discussed below. *EitherD* has constructors *EitherD-return* for returning a pure value, *EitherD-bind* for sequencing exceptional code, and *EitherD-bail* for throwing an exception. Additional constructors (not shown) help to structure proofs for conditional code.

The operational semantics of an *EitherD* program is given by *EitherD-run*. Running *EitherD-bind m f* first recursively runs *m*. If the result is *Left x* (an error), then it is returned. If it is *Right y*, then the result of recursively running *f y* is returned. (*RWS* is defined using a similar approach, but is somewhat more complicated than *EitherD*. The free monad for *RWS* has constructors for *return*, *bind*, *gets*, *put*, *ask* and *tell*. *RWS-run* assigns semantics straightforwardly for most of these. Running *RWS-bind m f* recursively runs *m*, calls *f* with the value returned by *m*, which produces another *RWS* program $f\ x_1$. Then, it runs $f\ x_1$ with the State resulting from running *m*, returning a tuple comprising: the resulting value, the state resulting from running $f\ x_1$ and the concatenation of values written by running *m* and then $f\ x_1$.)

EitherD-weakestPre defines, for any *EitherD E A* program *m*, a predicate transformer that produces the weakest precondition required to ensure that a given postcondition holds after executing *m*. The weakest precondition for a postcondition *P : Either E A → Set* (a predicate over *Either E A*) to hold after running *EitherD-return x* is that *P* (*Right x*) holds (because *EitherD-run* (*EitherD-return x*) = *Right x*); a similar situation applies to the *EitherD-bail* case. For *EitherD-bind m f*, the postcondition that is required for *m* is *bindPost f P*, which is the weakest precondition ensuring that *P* holds after executing *f* with the result (if any) of *m*. For the case in which *m* returns *Right y*, intuitively, we would require *EitherD − weakestPre* (*f y*) $'\cdot$. Our definition also provides an *alias c* for *y*, along with evidence that $c \equiv y$. While this is logically equivalent, the aliasing is helpful for keeping the proof state more comprehensible for a human reader, because they can control whether/when the structure of *c* is revealed, and until then see it as a single variable *c*.

```
data EitherD (E : Set) : Set → Set1 where
  EitherD-return : ∀ {A} → A → EitherD E A
  EitherD-bind   : ∀ {A B} → EitherD E A → (A → EitherD E B)
                   → EitherD E B
  EitherD-bail   : ∀ {A} → E → EitherD E A

EitherD-run : EitherD E A → Either E A
EitherD-run (EitherD-return x) = Right x
EitherD-run (EitherD-bind m f)
  with EitherD-run m
... | Left x  = Left x
... | Right y = EitherD-run (f y)
EitherD-run (EitherD-bail x) = Left x

EitherD-weakestPre : (m : EitherD E A) → (P : Either E A → Set) → Set
EitherD-weakestPre (EitherD-return x) P = P (Right x)
EitherD-weakestPre (EitherD-bind m f) P =
  EitherD-weakestPre m (bindPost f P) where
    bindPost f P (Left x)  = P (Left x)
    bindPost f P (Right y) = ∀ c → c ≡ y → EitherD-weakestPre (f c) P
EitherD-weakestPre (EitherD-bail x) P = P (Left x)

EitherD-contract : (m : EitherD E A) → (P : Either E A → Set)
                   → EitherD-weakestPre m P → P (EitherD-run m)
```

Fig. 1. The *EitherD* data type and associated definitions

The two semantics—operational and predicate transformer—are connected by the proof (not shown) of *EitherD-contract*, which states that, in order to show postcondition P holds for the result produced by m, it suffices to prove the weakest precondition of P with respect to m.

4 Related work

We have described a library that we developed to make our Agda code closely mirror the existing Haskell codebase from which we ported it, thus reducing the likelihood of errors. Here, we briefly survey some potential alternative approaches.

Haskell to Agda Our verification tool of choice for our broader project was Agda [1], due in part to experience and expertise within our group and other related projects. However, we could have used a different tool for our verification, such as Coq [6] or Isabelle/HOL [25], using translation tools such as `hs-to-coq` [8, 29] or "Haskabelle" [19], respectively (although Haskabelle seems to be unmaintained and out-of-date). CoverTranslator translates Haskell to Agda, but it is based on work from 2005 [2] and is not compatible with Agda 2 [13].

Agda to Haskell: An alternative high-level approach would be to develop our original code in Agda, verify it, and translate it back to Haskell. Proof assistants generally support "extracting" code to various languages [17, 22], including Agda's "MAlonzo" backend [3]. However, due to its attempt to generate Haskell for any Agda code, code generated by MAlonzo is unreadable and unmaintainable, which hinders debugging, performance analysis and tuning. "Agate" [27] is another Agda-to-Haskell compiler, but it hasn't been updated since 2008.

We have recently learned of `agda2hs` [26], which is work-in-progress towards translating a "Haskell-like" subset of Agda to *human-readable* Haskell. We cannot use it at this time because it does not currently support monadic computation. It includes a library that has similarities to ours, but it does not support guards, lenses, the RWS monad, etc. There is a potential for our work to influence improvements to `agda2hs`'s library and vice versa.

Lenses: Work on non-dependent, dependent and higher lenses [10, 14, 15] in Agda focuses on exploring their properties rather than providing a library that is useful in practice. Our *Optics* library is small, providing only the functionality and generality required for the purposes of our motivating project.

Predicate transformer semantics: We use a predicate transformer semantics based on Dijkstra's weakest precondition calculus as a verification methodology for programs in *Either* and *RWS* monads. This approach can be viewed as a case study of some of the techniques from the "Dijkstra monad" papers [4, 23, 28, 30].

5 Discussion

Although we have developed sufficient support in our library to enable translating a substantial Haskell code base (our implementation of BFT consensus based on HotStuff / LibraBFT) to Agda, we do not claim that it supports all Haskell language features. Furthermore, only a small number of Haskell library functions were needed.

Because we have manually implemented these language features and library functions, it would be possible for our Agda implementations to not faithfully capture the semantics of the original Haskell, in which case a correctness proof about the translated Agda version might fail to hold for the original Haskell code. Nonetheless, the small number of such language features and library functions we implemented to enable translating our motivating use case are in most cases based directly on the original Haskell functions and are small enough that we can be reasonably confident in them by inspection.

Our translation effort also resulted in some changes to the original Haskell code. In some cases this occurred simply because side-by-side code reviews of the Agda identified improvements that could be made to the Haskell code. Furthermore, due to our efforts to keep our Haskell code similar to the original Rust code on which it is based, many functions in the Haskell code would exit in response to unexpected conditions that are expressed in the original Rust code

via assertions. We initially considered modifying our system model to capture this behavior, so that we could express and prove properties showing that the unexpected conditions cannot occur. This approach was more disruptive than it was worth. We therefore refactored the Haskell implementation so that, after successful initialization, no code would ever exit. We did this by changing functions that contained such assertions so that they would return either an error or the original type, with errors propagated up the stack. This improved the quality of the implementation by ensuring not only that the desired safety properties hold, but also that the implementation would not exit unexpectedly.

As noted in Section 3, *EitherD* has additional constructors to capture conditional code. This enables proof obligations to be automatically generated for various cases of a conditional. The same is true for *RWS*. The details are beyond the scope of this paper, but will be addressed in a forthcoming paper, in which we generalize from these two examples (*EitherD* and *RWS*), showing that we can deliver the same benefits for a large class of ASTs of monadic programs by systematically extending them with constructors for conditionals.

6 Concluding remarks

We have described a library that we developed to support translating Haskell code to Agda that mirrors it closely and provides the necessary support for proving properties about it. This support includes various syntactic features, as well as predicate transformer semantics for *Either* and *RWS* (Reader, Writer, State) monads, which enable automatically determining the weakest precondition required for a given postcondition.

Our library is available in open source [7], along with the Agda implementation of Byzantine Fault Tolerant consensus based on HotStuff/LibraBFT that motivated this work. Nonetheless, our library is independent of this motivating project, and could be used and/or extended for a variety of projects.

Bibliography

[1] Agda 2.6.1.1 documentation (May 2021), http://agda.readthedocs.io/en/v2.6.1.1

[2] Abel, A., Benke, M., Bove, A., Hughes, J., Norell, U.: Verifying Haskell programs using constructive type theory. In: Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell. p. 62–73. Haskell '05, Association for Computing Machinery, New York, NY, USA (2005), http://doi.org/10.1145/1088348.1088355

[3] Agda team: GHC backend (2021), http://agda.readthedocs.io/en/v2.6.2.1/tools/compilers.html#ghc-backend

[4] Ahman, D., Hritcu, C., Maillard, K., Martínez, G., Plotkin, G.D., Protzenko, J., Rastogi, A., Swamy, N.: Dijkstra monads for free. In: Castagna, G., Gordon, A.D. (eds.) Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017. pp. 515–529. ACM (2017), http://doi.org/10.1145/3009837.3009878

[5] Baudet, M., Ching, A., Chursin, A., Danezis, G., Garillot, F., Li, Z., Malkhi, D., Naor, O., Perelman, D., Sonnino, A.: State machine replication in the libra blockchain (2019), http://developers.diem.com/papers/diem-consensus-state-machine-replication-in-the-diem-blockchain/2019-06-28.pdf

[6] Bertot, Y., Castran, P.: Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions. Springer Publishing Company, Incorporated, 1st edn. (2010)

[7] BFT consensus in Agda (December 2021), http://github.com/oracle/bft-consensus-agda/releases/tag/nasafm2022

[8] Breitner, J., Spector-Zabusky, A., Li, Y., Rizkallah, C., Wiegley, J., Weirich, S.: Ready, set, verify! Applying hs-to-coq to real-world Haskell code (experience report). Proc. ACM Program. Lang. **2**(ICFP), 89:1–89:16 (2018), http://doi.org/10.1145/3236784

[9] Cachin, C., Vukolic, M.: Blockchain consensus protocols in the wild. CoRR **abs/1707.01873** (2017), http://arxiv.org/abs/1707.01873

[10] Capriotti, P., Danielsson, N.A., Vezzosi, A.: Higher lenses. In: 36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021. pp. 1–13. IEEE (2021), http://doi.org/10.1109/LICS52264.2021.9470613

[11] Carr, H., Jenkins, C., Miraldo, V.C., Moir, M., Silva, L.: Towards formal verification of HotStuff-based byzantine fault tolerant consensus in Agda. In: Proceedings of the 14th NASA Formal Methods Symposium. NFM '22, Springer-Verlag, Berlin, Heidelberg (May 2022)

[12] Carr, H., Jenkins, C., Moir, M., Miraldo, V.C., Silva, L.: Towards formal verification of HotStuff-based byzantine fault tolerant consensus in Agda: Extended version (2022), arxiv.org/abs/2203.14711

[13] CoverTranslator, http://github.com/langston-barrett/CoverTranslator/blob/master/README.md, fetched December 21, 2021
[14] Danielsson, N.A.: Dependent lenses (2015), http://www.cse.chalmers.se/~nad/publications/danielsson-dependent-lenses.pdf
[15] Danielsson, N.A.: dependent-lenses (2021), http://github.com/nad/dependent-lenses
[16] Diehl, S.: RWS monad, http://dev.stephendiehl.com/hask/#rws-monad
[17] Filliâtre, J.C., Letouzey, P.: Program extraction, http://coq.inria.fr/refman/addendum/extraction.html
[18] FPComplete: Lenses, http://www.fpcomplete.com/haskell/tutorial/lens/
[19] Haftmann, F.: From higher-order logic to haskell: there and back again. In: PEPM '10 (2010)
[20] van Laarhoven, T.: Cps based functional references (2009), http://www.twanvl.nl/blog/haskell/cps-functional-references
[21] Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16**(2), 133–169 (May 1998), http://doi.org/10.1145/279227.279229
[22] Lean team: Programming in lean, http://leanprover.github.io/programming_in_lean/#01_Introduction.html
[23] Maillard, K., Ahman, D., Atkey, R., Martínez, G., Hritcu, C., Rivas, E., Tanter, É.: Dijkstra monads for all. Proc. ACM Program. Lang. **3**(ICFP), 104:1–104:29 (2019), http://doi.org/10.1145/3341708
[24] Milewshi, B.: Error handling (2015), http://www.schoolofhaskell.com/school/starting-with-haskell/basics-of-haskell/10_Error_Handling
[25] Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-Order Logic, Lecture Notes in Computer Science, vol. 2283. Springer (2002)
[26] Norell, U., Chapman, J., Melkonian, O., Cockx, J., Abel, A., Escot, L., Sabharwal, D.: agda2hs (2021), http://github.com/agda/agda2hs
[27] Ozaki, H., Takeyama, M., Kinoshita, Y.: Agate—an agda-to-haskell compiler. Computer Software **26**(4), 4_107–4_119 (2009), http://doi.org/10.11309/jssst.26.4_107
[28] Silver, L., Zdancewic: Dijkstra Monads Forever: Termination-Sensitive Specifications for Interaction Trees (Nov 2020), http://doi.org/10.5281/zenodo.4312937
[29] Spector-Zabusky, A., Breitner, J., Rizkallah, C., Weirich, S.: Total haskell is reasonable coq. In: Andronick, J., Felty, A.P. (eds.) Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018. pp. 14–27. ACM (2018), http://doi.org/10.1145/3167092
[30] Swamy, N., Weinberger, J., Schlesinger, C., Chen, J., Livshits, B.: Verifying higher-order programs with the Dijkstra monad. In: Proceedings of the 34th annual ACM SIGPLAN conference on Programming Language Design and Implementation. pp. 387–398. PLDI

'13 (2013), http://www.microsoft.com/en-us/research/publication/verifying-higher-order-programs-with-the-dijkstra-monad/

[31] Swierstra, W., Baanen, T.: A predicate transformer semantics for effects (functional pearl). Proc. ACM Program. Lang. **3**(ICFP), 103:1–103:26 (2019), http://doi.org/10.1145/3341707

[32] The LibraBFT Team: State machine replication in the libra blockchain (5 2020), http://developers.diem.com/papers/diem-consensus-state-machine-replication-in-the-diem-blockchain/2020-05-26.pdf

[33] Tholoniat, P., Gramoli, V.: Formal verification of blockchain byzantine fault tolerance (Oct 2019). https://doi.org/10.48550/ARXIV.1909.07453, https://arxiv.org/abs/1909.07453

[34] Yin, M., Malkhi, D., Reiter, M.K., Gueta, G.G., Abraham, I.: Hotstuff: Bft consensus with linearity and responsiveness. In: Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing. p. 347–356. PODC '19, Association for Computing Machinery, New York, NY, USA (2019), http://doi.org/10.1145/3293611.3331591