SYNCOPE: Automatic Enforcement of Distributed Consistency Guarantees

Kia Rahmani[1], Gowtham Kaki[1], and Suresh Jagannathan[1]

Purdue University, West Lafayette IN 47906, USA
`{rahmank,gkaki,suresh}@purdue.edu`

Abstract. Designing reliable and highly available distributed applications typically requires data to be replicated over geo-distributed stores. But, such architectures force application developers to make an undesirable tradeoff between ease of reasoning, possible when replicated data is required to be strongly consistent, and performance, possible only when such guarantees are weakened. Unfortunately, undesirable behaviors may arise under weak consistency that can violate application correctness, forcing designers to either implement ad-hoc mechanisms to avoid these anomalies, or choose to run applications using stronger levels of consistency than necessary. The former approach introduces unwanted complexity, while the latter sacrifices performance.
In this paper, we describe a lightweight runtime verification system that relieves developers from having to make such tradeoffs. Instead, our approach leverages declarative axiomatic specifications that reflect the necessary constraints any correct implementation must satisfy to guide a runtime consistency enforcement mechanism. This mechanism guarantees a *provably optimal* strategy that imposes no additional communication or blocking overhead beyond what is required to satisfy the specification, allowing distributed operations to run in a *provably safe* environment. Experimental results show that the performance of our automatically derived mechanisms is better than both specialized hand-written protocols and common store-offered consistency guarantees, providing strong evidence of its practical utility.

Keywords: runtime safety enforcement, weak consistency, distributed systems, Haskell

1 Introduction

Historically, the *de facto* system abstraction for developing distributed programs has always included the ACID[1] properties. These properties, guarantee replication transparency (i.e. requiring distributed systems to *appear* as a single compute and storage server to users), and have resulted in development of standardized implementation and reasoning techniques around *strongly consistent* (SC) distributed stores. Although strong notions of consistency, are ideal for developement and reasoning about distributed applications, they require extensive

[1] Atomicity, Consistency, Isolation and Durability

synchronization overhead which is unacceptable for web-scale applications that wish to be "always-on" despite network partitioning. Applications are therefore usually designed to tolerate certain *inconsistencies*, in exchange for availability and low-latency. An extreme example is *eventual consistency* (EC), where the local state of each node at all time, only represents an *unspecified order* of an *unspecified subset* of the set of all updates submitted to the system globally. Applications that cannot tolerate the anomalous behaviors allowed under EC, may choose to use various stronger instantiations, that are collectively refered to as *weak consistency* guarantees. Unfotunately, weak notions of consistency, are closely tied to specific data-store implementations, and in very few cases, such as *causal consistency* (CC) for which there exists relatively standard definitions and known implementation techniques, users are usually offered with unnecessary levels of consistency and potental performance loss[2]. In order to face this problem, developers are forced to inject their code with *ad-hoc* anomaly tolerance mechanisms that are closely tied to the application logic and conflate it with concerns orthogonal to its semantics. To illustrate this problem, we will present an example in section 3, where we introduce a simple distributed application developed on top of an off-the-shelf eventually consistent data-store (ECDS), and explain how it must be re-engineered from the scratch in order to preempt certain undesired behaviors (i.e. enforce fine-grained weak consistency requirements). As we will see, the ad-hoc nature of such mechanisms confounds standardization, and complicates reasoning, maintainability and reusability of the applications.

In this paper, we propose an alternative to the aforementioned approaches that overcomes their weaknesses. SYNCOPE is a lightweight runtime system for Haskell that allows application developers to take advantage of weak consistency without having to re-engineer their code to accommodate anomaly preemption mechanisms. The key insight that drives SYNCOPE's design is that the hardness of reasoning about the integrity of a distributed application stems from conflating application logic with the consistency enforcement logic, and reasoning about both *operationally*. By separating application semantics from consistency enforcement semantics, admitting operational reasoning for the former, and declarative reasoning for the latter programmers are liberated from having to worry about implementation details of anomaly preemption mechanisms, and instead focus on reasoning about application semantics, under the assumption that specified consistency requirements are automatically enforced by the data store at runtime. Our approach admits declarative reasoning for consistency enforcement via a specification language that allows programmers formally specify the consistency requirements of their application. The design of our specification language is based on the observation that all anomalous behaviors allowed under EC, occur as the result of nodes executing operations, before a certain set of *dependencies* arrive at that node. Users in SYNCOPE, can specify arbitary dependency relations between updates, and the runtime system working on top

[2] In fact, CC is the strongest consistency guarantee that remains available under network partitioning

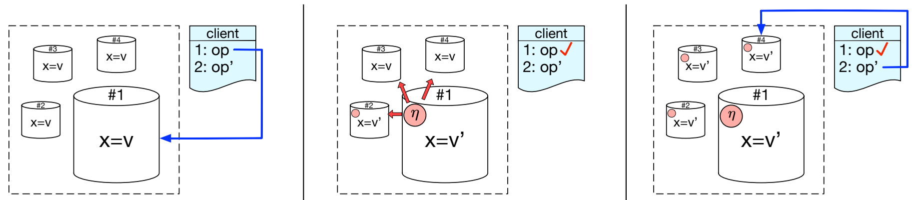
of each ECDS replica, guarantees that an operation will only proceed if it can witness all of its dependencies. For example, *lost-updates*, which is a very well known anomaly under EC, occures when an operation from a session is routed to a replica different than the replica that served the earlier operaitons of the same session (because of transient system properties, such as load balancing or network partitions), and is successfully executed without witnessing the update from those earlier operations. In this case, the dependency of the operations can be defined as the updates from *all previous operations from the same sessions*, and SYNCOPE is guaranteed to temporarily block operations untill all such dependencies become available at a replica.

To summarize the contributions of this paper: (i) We propose a specification language to express the fine-grained consistency requirements of applications in terms of the dependencies between operations. (ii) We describe a generic consistency enforcement runtime that analyzes each operation's consistency specification, and ensures that its dependencies are available before it is executed. We formalize the operational semantics of the runtime, and prove its correctness and optimality (including *minimum blocking* and *minimum staleness*) guarantees. (iii) We describe an implementation of our specification language and consistency enforcement runtime in a tool called SYNCOPE, which works on top of an off-the-shelf EC data store. We evaluate SYNCOPE over realistic applications and microbenchmarks, and present results demonstrating the performance benefits of making fine-grained distinctions between consistency guarantees, and the ease of doing so via our specification language.

The remainder of the paper is organized as follows. A system model that describes the key notions of consistency and replication is presented in Sec. 2. In Sec. 3 we provide a detailed example to further motivate the problem. In Sec. 4 and Sec. 5, we formally present our specification language and the high level operational semantics of the runtime system, with correctness and optimality theorems. Sec. 6 elaborates on the algorithmic aspects of our runtime that is key to its efficient realization. Sec. 7 describes implementation of SYNCOPE, and evaluates its applicability and practical utility. Related works and conclusion are presented in Sec. 8 and Sec. 9

2 System Model

A data store in our system model is a collection of *replicas* (`#1`,`#2`,...), each of which maintains a copy of a set of replicated *data object* (`x`,`y`,...). Each data object includes and maintains a *state value* (`v`,`v'`,...) and is equipped with a set of *operations* (`op`,`op'`,...). Operations may read the state of an object residing in a replica, and modify it by generating *update effects* (η,η',...). Update effects or simply effects are then asynchronously sent to all other replicas, where, by using a user-defined function, are *applied* to the state of the object instance at the recipient replica. Fig. 1a and 1b illustrate this process, where the example shows how effects are locally created and remotely applied.



(a) A client submits an operation `op` to the store, which is routed to the replica `#1`.

(b) The state of the replica `#1` is updated, an effect is created and is being propagated

(c) Second operation `op′` is submitted to the store, which is routed to the replica `#4`

Fig. 1: system model of SYNCOPE

Observe that in our system model, there is no direct synchronization between replicas when an operation is executed, which means concurrent and possibly conflicting updates can be generated at different replicas. We require the user-defined *apply* function to implement a conflict resolution strategy for replicas to eventually converge. This model admits all inconsistencies and anomalies associated with eventual consistency [1, 2], and our goal is equip applications and implementations with mechanisms to specify and prevent such inconsistencies.

Clients in our model, interact with the store by invoking operations on objects. A *session* is a sequence of operations invoked by a particular client. Consequently, operations (and effects) can be uniquely identified by the *session id* that invoked them, and their *sequence number* in that particular session, which is used by replicas, to record the set of all updates that are locally applied. Since, the data store is concurrently accessed by a typically large number of clients, and as a result of the load balancing regulations, operations (even from the same session) might be routed to different replicas (Fig. 1a and 1c).

Lastly, we define two relations over effects created in the store. *Session order* (so) is an irreflexive, transitive relation that relates an effect to all subsequent effects from the same session. Moreover, we define *visibility* (vis) as an irreflexive and assymetric relation that relates an effects to all others that are influenced by it (witnessed its update) at the time of their generation. For example, in Fig. 1c $\mathtt{vis}(\eta, \eta')$ holds, since η (the effect of `op`) has already been delivered and applied to the replica #4, when $\mathtt{op}'$ is executed and thus has influenced generation of η'.

3 Motivation

3.1 Replicated Data Types in ECDS

To provide further motivation, consider a highly available (low latency) application for managing comments on posts in a photo sharing web site. Fig. 2a presents a simple Haskell implementation of such an application cognizant of our system model.

In the implementation, `Effect` and `State` strings are respectively defined as the text of a single comment, and the concatentation of all visible comments

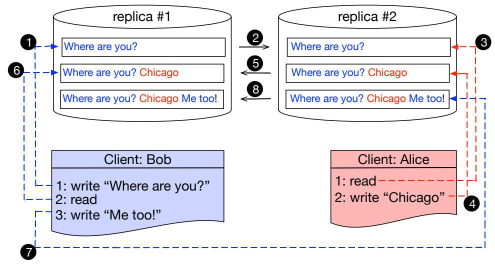
```
type Effect = String
type State =  String

read :: State -> (String,Maybe Effect)
read s = (s,Nothing)

write :: String -> ((),Maybe Effect)
write comment = ((),Just comment)

apply :: State -> Effect -> State
apply s comment = s ++ comment
```

(a) A simple implementation



(b) Example execution

Fig. 2: A distributed application for comment section management

associated with a post. A new `Effect` is generated every time a user wants to comment on a post by calling the `write` function, and a `read` call simply returns the `State` of the object at the serving replica. The `apply` function, simply returns the updated state of the replica, which is the concatination of the old state and the given effect. For perspicuity, we omit any conflict resolution strategy in the code; however, developers (using roll-backs, etc) can design the `apply` function to resolve conflicting concurrent updates as they desire.

An example of how users interact with this application is presented in Fig.2b, where Alice and Bob are invoking operations on an object (here, a photo of Alice in Chicago), and the chronological order of events is given in black circles. At time ❶, Bob writes a comment, which is routed to replica #1, whose effect is then propagated and delivered to replica #2 at ❷; where Alice's first read operation is routed to at ❸. Alice and Bob then keep talking through more read and write events, while updates are propagated between the two replica.

As mentioned before, lost-updates, is a well known undesirable behavior admitted by ECDS. An example of such anomaly can occure here if at time ❻, Bob is temporarily disconnected from both replicas in the figure, and his read operation is routed to another replica #3, that has not yet received any updates from #1 or #2. Consequently, Bob cannot see his first comment and would retry submitting it, assuming the first time it was failed.

3.2 Ad-hoc Anomaly Prevention

A known technique to prevent the above anomaly, is to tag each effect using a unique identifier as mentioned in Sec. 2. Using these tags, replicas will be able to track all locally available effects, and temporarily *block* operations, until all the preceding effects from the same session arrive at the replica. For example, the replica #3 that receives Bob's read in the above undesired scenario, can simply postpone its execution until all dependencies arrive.

In order to reduce the overhead of tracking dependencies per operation, the above idea is further realized using another technique called *filteration*, which is based on separating the locally available effects at each replica that have not been applied to the state yet and those who have. By this separation, in the above example each replica can maintain a *safe environment* for operations (e.g. using a soft-state cache), that contains an effect only if it also contains all the previous effects from the same session. This way, an operation can proceed, when the effect of the very exact previous operation from the same session is already applied to the state (which transitively yields the presence of all dependencies).

We present a modified version of the running example in appendix A, which is updated to tolerate the lost-update anomaly by implementing the blocking mechanism in the `read` function and the filteration in the `apply` function as explained above. Unfortunately, these modifications require fundamental and pervasive changes to the original code including almost all type and function definitions. Additionally, the changes are heavily tangled with application logic, complicating reasoning and hampering correctness arguments.

A major drawback of this approach in stores that do not admit metadata queries (e.g. Cassandra), is the *lost histories*[3] problem. To face this problem, for each new session joining, the replicas must perforom a table alteration at the data store level, to accomodate the data on the newly joined session. This requires strong synchronization of replicas, degrading application performance and availability. Moreover, to make the matter worse, new anomalies are constantly found in the systems after the design phase, which require non-trivial, further polluting, ad-hoc solutions that leave the existing implementation obsolete.

3.3 An Alternative

We now present our generic consistency management tool. SYNCOPE allows developers to define a consistency level for each operation *a priory*, and rely on the runtime system for its satisfaction. Our approach is consisted of generalized blocking and filteration mechanisms, which admits arbitrary user-defined dependency relations for each operation and maintains a multi-consistent *shim layer* on top of each ECDS replica.

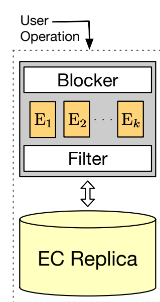


Fig. 3: SYNCOPE

The SYNCOPE shim layer maintains multiple safe environments $(E_1, E_2, ...)$ by periodic (or on-demand) reads from the underlying ECDS database, and adding effects to each environment, only if its dependencies have already been added (Fig.3). SYNCOPE realizes this idea efficiently, using a simple tagging mechanism that represents effects in an environment by giving them a tag associated with that environment. Each operation in SYNCOPE only witnesses its associated environement, and is blocked by the runtime system, if the necessary effects are not in there yet.

Users in our tool can specify arbitrary consistency guarantees in a language that is seeded with `so` and `vis` relations and allows them to define constraints on read operations, that can be used to synthesize appropriate filteration and blocking mechanisms. For example, the following *contract*, eliminates the possibilty of lost-update anomaly, by establishing the appropriate condition under which an effect may be witnessed by the current operation:

$$\psi : \forall a. \xrightarrow{\mathsf{so}} \hat{\eta} \Rightarrow a \xrightarrow{\mathsf{vis}} \hat{\eta}$$

$$
\begin{array}{lll}
\mathtt{r} \in \mathtt{rel.seed} & := \mathsf{vis} \mid \mathsf{so} \mid \mathtt{r} \cup \mathtt{r} \\
\mathsf{R} \in \mathtt{relation} & := \mathtt{r} \mid \mathsf{R};\mathtt{r} \mid \mathtt{null} \\
\pi \in \mathtt{prop} & := \forall a.\ a \xrightarrow{R} \hat{\eta} \ \Rightarrow\ a \xrightarrow{\mathsf{vis}} \hat{\eta} \\
\psi \in \mathtt{spec} & := \pi \mid \pi \wedge \pi
\end{array}
$$

(a) syntax of contracts

Guarantee	Contract
RMW	$\forall a.a \xrightarrow{\mathsf{so}} \hat{\eta} \Rightarrow a \xrightarrow{\mathsf{vis}} \hat{\eta}$
MW	$\forall a.a \xrightarrow{\mathsf{so;vis}} \hat{\eta} \Rightarrow a \xrightarrow{\mathsf{vis}} \hat{\eta}$
MR	$\forall a.a \xrightarrow{\mathsf{vis;so}} \hat{\eta} \Rightarrow a \xrightarrow{\mathsf{vis}} \hat{\eta}$
2VIS	$\forall a.a \xrightarrow{\mathsf{vis;vis}} \hat{\eta} \Rightarrow a \xrightarrow{\mathsf{vis}} \hat{\eta}$

(b) examples

Fig. 4: SYNCOPE Specification Language

4 Specification Language

The formal syntax of our specification (or contract) language, presented in Fig.4a, allows definition of `prop`, that is a FOL formula establishing dependency relations between effects, which is necessary to determine effects an operation may witness, under a consistency requirement. The language is seeded with so and vis, respectively representing session order and visibility over effects, and defines dependency `relation` as a sequence[3] of seeds, where $(a \xrightarrow{\mathtt{r}_1;\ldots;\mathtt{r}_k} b)$ must be interpreted as $\exists c.(a \xrightarrow{\mathtt{r}_1;\ldots;\mathtt{r}_{k-1}} c \wedge c \xrightarrow{\mathtt{r}_k} b)$ in meta-language. We also define `null` as a relation that never holds. Additionally, the language allows definition of `spec`, a conjunction of propositions, that is used to define a safe environment free from *multiple* inconsistencies. Our language is crafted to capture all fine-grained weak consistency levels, including the famous session guarantees proposed by Terry et al. [2], presented in Fig.4b.

We finish this section by introducing two syntactical classes of contracts, and explaining how they can be satisfied with different enforcement techniques.

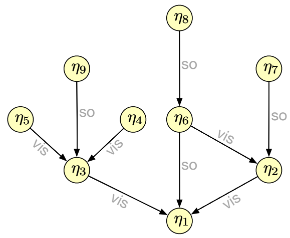
- LB: If all the defined dependency relations for a contract end with an so, i.e. are of the following form: $(\forall a.a \xrightarrow{r_1;r_2;\ldots;\mathsf{so}} \hat{\eta} \Rightarrow a \xrightarrow{\mathsf{vis}} \hat{\eta})$, we call it a *lower bound* (LB) contract, since it specifies the smallest set of effects that any operation should witness to maintain consistency, e.g. RMW and MR in Fig.4b.
- UB: Similarly, we define the *upper bound* (UB) contracts, as the ones with all dependency relations ending with a vis. These contracts define constraints on the set of effects made visible to each operation, by enforcing that if an effect is in the set, certain dependencies of that effect must also be included, e.g. 2VIS and MW in Fig.4b.

Our consistency enforcement approach is based on blocking operations with LB contracts to make sure that they witness *all effects that they are supposed to*, and filteration for UB contracts to make sure that they would not witness *effects that they are not supposed to.* A combination of both approaches is also taken for contracts that are neither LB nor UB, i.e hybrid contracts.

5 Semantics

In this section, we present the consistency enforcement mechanism of SYNCOPE, abstracted as a formal operational semantics. Our approach is complete for the

[3] SYNCOPE also allows using closures of seeds, which is omitted here for simplicity



(a) An execution state E

$$\begin{aligned}
\mathsf{E.A} &= \{\eta_1, \eta_2, \eta_3, \eta_4, \eta_5, \\
&\quad\ \eta_6, \eta_7, \eta_8, \eta_9\} \\
\mathsf{E.vis} &= \{(\eta_5, \eta_3), (\eta_4, \eta_3), \\
&\quad\ (\eta_3, \eta_1), (\eta_2, \eta_1), \\
&\quad\ (\eta_6, \eta_2)\} \\
\mathsf{E.so} &= \{(\eta_9, \eta_3), (\eta_8, \eta_6), \\
&\quad\ (\eta_6, \eta_1), (\eta_8, \eta_1), \\
&\quad\ (\eta_7, \eta_2)\}
\end{aligned}$$

(b) Primitive relations

$$\begin{aligned}
\mathsf{vis}^{-1}(\eta_1) &= \{\eta_2, \eta_3\} \\
\mathsf{so}^{-1}(\eta_1) &= \{\eta_6, \eta_8\} \\
(\mathsf{so} \cup \mathsf{vis})^{-1}(\eta_1) &= \{\eta_2, \eta_3, \eta_6, \eta_8\} \\
(\mathsf{vis}; \mathsf{vis})^{-1}(\eta_1) &= \{\eta_4, \eta_5, \eta_6\} \\
(\mathsf{so}; \mathsf{vis})^{-1}(\eta_1) &= \{\eta_7, \eta_9\}
\end{aligned}$$

(c) Inverse of relations

Fig. 5: A simple execution state

specification language defined in Sec.4, however for better comprehensibility, here we present the semantics and the theorems paramterized over a non-hybrid contract consisting of a single proposition. Therefore, in the rest of this section, we will assume a given contracat ψ of the following form:

$$\psi = \forall a.a \xrightarrow{\mathtt{r}_1;\mathtt{r}_2;\ldots;\mathtt{r}_k} \hat{\eta} \Rightarrow a \xrightarrow{\mathsf{vis}} \hat{\eta} \qquad\qquad \mathtt{r}_i \in \{\mathsf{vis}; \mathsf{so}\}$$

The operational semantics defines a small-step relation over *execution states*, which are tuples of the form $\mathsf{E} = (\mathsf{A}, \mathsf{vis}, \mathsf{so})$. The *effect soup* A, stands for the set of all effects produced in the system, and *primitive relations* $\mathsf{vis}, \mathsf{so} \subseteq \mathsf{A} \times \mathsf{A}$, respectively represent the visibility and session order among such effects. Figures 5a and 5b present a simple execution state consisting of 9 effects with associated primitive relations[4]. We denote the subset of A consisting of effects that satisfy a certain condition as $\mathsf{A}_{(\mathtt{condition})}$.

Note that SYNCOPE's contracts are in fact constraints over execution states, where the domain of quantification is fixed to the effect soup A, and interpretation for so and vis relations (which occur free in the contract formulae) are also provided. Thus, execution states are potential models for any first-order formula expressable in the specification language. If an execution state E is in fact a valid model for a contract ψ, we say that E satisfies ψ, written as $\mathsf{E} \models \psi$.

The reduction relation in our semantics is of the form $(\mathsf{E}, \mathsf{op}_{<s,i>}) \xrightarrow{V} (\mathsf{E}', \eta)$, which can be interpreted as the transformation of the initial execution state E, caused by a replica with a local set of effects V, when it executes op, the i^{th} operation from the session s. During this reduction step, a new effect η is produced and added to the system, resulting in a new execution state E' composed of an updated effect soup and new primitive relations.

5.1 Preliminaries

In this part, we will present the formal definitions and notations required to explain our operational semantics. We start by defining the interpretation of an *inversed* dependency relation R^{-1} under an execution state E, which is utilized in the basis of our consistency enforcement mechanism. We previously mentioned

[4] we omit drawing transitive so edges (e.g. between η_8 and η_1) for better readability

our interpretation for so and vis between effects under E, which can now be straightforwardly extended to their inverse as follows[5]:

$$\mathtt{r}^{-1}(S) = \bigcup_{b \in S} \{a | (a, b) \in \mathsf{E}.\mathtt{r}\} \qquad \mathtt{r} \in \{\mathsf{so}, \mathsf{vis}\} \tag{1}$$

Additionally, based on our interpretation of the sequences of seed relations given in Sec.4, we can extend the above definition to the following:

$$b \in (\mathsf{R}'; \mathtt{r})^{-1}(a) \iff \exists c. c \in \mathtt{r}^{-1}(a) \wedge b \in (\mathsf{R}')^{-1}(c) \tag{2}$$

Now, It might seem that we are ready to define any R^{-1} based on the two definitions above; however, note that definition (2) fails to capture the reality of our system model, where all computations are performed by replicas independently, which at any given moment, might have access to only a *subset of all produced effects* in the system. For example, consider $(\mathsf{so}; \mathsf{vis})^{-1}(\eta_1)$ under the execution state presented in Fig.5. In order to compute this set, based on (2) we have:

$$b \in (\mathsf{so}; \mathsf{vis})^{-1}(\eta_1) \iff \exists c. c \in \mathsf{vis}^{-1}(\eta_1) \wedge b \in (\mathsf{so})^{-1}(c)$$

Now, since there exist *mid-level* effects $c = \eta_2$ and $c = \eta_3$, such that satisfy the above definition respectively for η_7 and η_9, we can conclude: $(\mathsf{so}; \mathsf{vis})^{-1}(\eta_1) = \{\eta_7, \eta_9\}$. Now consider a replica that contains $\{\eta_1, \eta_6, \eta_7, \eta_9\}$ at the moment, and wants to check if the dependencies of η_1 are locally present or not. Even though based on the above definition, the answer is affirmativ (since the replica does contain $\{\eta_7, \eta_9\}$), but in reality the replica has no way of verifying it, since the mid-level effects η_2 and η_3 are not present at the replica yet.

To capture the above property, we now partially define the inverse of R, *according to a set of available effects* V, only if all the required mid-level effects are present in V. The following is our definition, based on (1) and a more strict version of (2):

$$b \in \mathsf{R}_V^{-1}(a) \iff \begin{cases} \bot & \mathtt{if} \;\; \mathsf{R} = \mathtt{null} \\ b \in \mathtt{r}^{-1}(a) & \mathtt{if} \;\; \mathsf{R} = \mathtt{r} \\ \exists c. c \in \mathtt{r}^{-1}(a) \wedge b \in (\mathsf{R}')^{-1}(c) \wedge \mathtt{r}^{-1}(a) \subseteq V & \mathtt{if} \;\; \mathsf{R} = \mathsf{R}'; \mathtt{r} \end{cases} \tag{3}$$

For example, in Fig.5, $(\eta_9 \in (\mathsf{so}; \mathsf{vis})^{-1}_{\{\eta_1, \eta_3\}}(\eta_1))$ holds, but $(\eta_9 \notin (\mathsf{so}; \mathsf{vis})^{-1}_{\{\eta_1\}}(\eta_1))$. Furthermore, we can now define a set V to be *self-contained* for a given effect η, written as $\mathbb{SC}_\eta^R(V)$, if V contains all the required mid-level effects to compute R inverse of η in totality, i.e.

$$\mathbb{SC}_\eta^R(V) \iff R_V^{-1}(\eta) = R_{E.A}^{-1}(\eta) \tag{4}$$

For example in Fig.5, $\mathbb{SC}_{\eta_1}^R(V)$ holds for an arbitrary R and for any V that is a superset of $\{\eta_1, \eta_2, \eta_3, \eta_4, \eta_5\}$.

Now, we define $\mathsf{trunc}()$ as a function that given $R \in \mathtt{relation}$, returns a new relation by removing the last element from the sequence in R:

$$\mathsf{trunc}(R) = \begin{cases} \mathtt{null} & \mathtt{if} \quad R = \mathtt{r} \quad \mathtt{or} \quad R = \mathtt{null} \\ R' & \mathtt{if} \quad R = R'; \mathtt{r} \end{cases} \tag{5}$$

[5] Note that when the input of an inversed relation is a singleton $\{\eta\}$, we drop the brackets and simply write it as $\mathtt{r}^{-1}(\eta)$

Auxiliary Definitions

$$\begin{array}{llll} op & \in \texttt{Oper. Name} & F_{op} & \in \texttt{Op.Def.} := \mathcal{P}(\eta) \mapsto v \\ v & \in \texttt{Ret. Val.} & \mathsf{A} & \in \texttt{Eff Soup} := \mathcal{P}(\eta) \\ s & \in \texttt{Sess. ID} & \mathsf{vis}, \mathsf{so} & \in \texttt{Relations} := \mathcal{P}((\eta, \eta)) \\ \eta & \in \texttt{Effect} := (s, op, v) & \mathsf{E} & \in \texttt{Exec State} := (\mathsf{A}, \mathsf{vis}, \mathsf{so}) \end{array}$$

Auxiliary Reduction

$$\boxed{S \vdash (\mathsf{E}, op_{<s,i>}) \hookrightarrow (\mathsf{E}', \eta)}$$

[Oper]

$$\frac{S \subseteq \mathsf{A} \qquad F_{op}(S) = v \qquad \eta \notin S \qquad \eta = (s, op, v) \\ \mathsf{A}' = \mathsf{A} \cup \{\eta\} \qquad \mathsf{vis}' = \mathsf{vis} \cup (S \times \{\eta\}) \qquad \mathsf{so}' = \mathsf{so} \cup (\mathsf{A}_{(\mathsf{SessID}=s)} \times \{\eta\})}{S \vdash ((\mathsf{A}, \mathsf{vis}, \mathsf{so}), op_{<s,i>})) \hookrightarrow ((\mathsf{A}', \mathsf{vis}', \mathsf{so}'), \eta)}$$

Operational Semantics

$$\boxed{(\mathsf{E}, op_{<s,i>}) \xrightarrow{V} (\mathsf{E}', \eta)}$$

[UB Exec]

$$\frac{\mathtt{r}_k = \mathsf{vis} \quad V \subseteq E.A \quad V' = \lfloor V \rfloor_{\mathtt{max}} \\ V' \vdash (E, op_{<s,i>})) \hookrightarrow (E', \eta)}{(\mathsf{E}, op_{<s,i>}) \xrightarrow{V} (\mathsf{E}', \eta)}$$

[LB Exec]

$$\frac{\mathtt{r}_k = \mathsf{so} \quad V \subseteq E.A \quad \mathbb{SC}^R_\eta(V) \\ R_V^{-1}(\eta) \subseteq V \qquad V \vdash (E, op_{<s,i>})) \hookrightarrow (E', \eta)}{(\mathsf{E}, op_{<s,i>}) \xrightarrow{V} (\mathsf{E}', \eta)}$$

Fig. 6: Core Operational semantics of a replicated data store.

Finally, we define *closed subsets* of a given set V, as the subsets that are closed under $(\mathsf{trunc}(R)_V^{-1})$, that also contain all the required mid-level effects to compute $\mathsf{trunc}(R)^{-1}$. Moreover, we define the largest element among such subsets, as the *maximally closed subset* of V as follows[6]:

$$\begin{array}{rl} \texttt{closed subsets}: V' \in \lfloor V \rfloor & \iff V' \subseteq V \quad \wedge\ (\mathsf{trunc}(R))_V^{-1}(V') \subseteq V' \ \wedge\ \mathbb{SC}_\eta^{\mathsf{trunc}(R)}(V') \\ \texttt{maximally closed subset}: V' = \lfloor V \rfloor_{\mathtt{max}} & \iff V' \in \lfloor V \rfloor \ \wedge\ \nexists V'' \in \lfloor V \rfloor . |V''| > |V'| \end{array}$$

5.2 Core Operational Semantics

In this part we present the operational semantics, as a set of rules representing our consistency enforcement approach. Fig.6 presents the rules defining the auxiliary relation ($\hookrightarrow$) and then the small-step reduction relation ($\rightarrow$) over execution states, where the latter is parametrized over a set V, which stands for the locally available set of effects at the replica taking the reduction step. Trivially, V must be a subset of all effects in the system at the initial execution state, however, there is no other restrictions on V, since we only assume eventual consistency at the underlying store.

The rule [oper] defines the abstract procedure of generating a new effect η, by witnessing a set of effects S, using a user-defined function F_{op}. We formally define an effect as a tuple $\eta = (s, op, v)$, representing the session and the operation name whose execution created η, and the value v that the replica returns, responding

[6] We slightly abuse the previously defined notation in (3) and use a *set* of effects as the input of R^{-1}, which is defined as: $x \in R_V^{-1}(S) \iff \exists (y \in S). x \in R_V^{-1}(y)$

to that operation. Moreover, the rule explains how the execution state changes after a new effect is produced. Specifially, in the new execution state, the effect soup A' contains the newly created effect η, the relation vis' captures the fact that all effects in the set S were made visible to η, and so' states that all effects from the same session as the current operation, that are already presenet in the system, should be in session order with η in the final execution state.

Now we explain the rules for the reduction relation ($\xrightarrow{V}$), starting with [UB EXEC], which defines the execution of an operation in a replica under a UB contract. The rule requires operations to only witness V', the maximally closed subset of V, or in other words, the rule governs replicas to create safe environments for operations, by filtering out effects that may cause the specified anomaly.

The next rule, [LB EXEC], defines the step taken by a replica, when an operation is executed under an LB contract. The precondition $R_V^{-1}(\eta) \subseteq V$ in the rule, ensures that the reduction happens only if the effects necessary to avoid the specified anomaly are present in V, assuming that V contains all the mid-level effects to determine dependencies of the newly created effect η (i.e. is a self contained set). In other words, the rule governs replicas to block execution of an operation under an LB contract, if the replica is unable to verify the presence of all necessary dependent effects.

5.3 Soundness and Optimality

In this section we present our meta-theoretic results on the desired properties for our consistency enforcement mechanism. Three theorems are presented, regarding the correctness of our approach, maximality of witnessed effects by each operation (i.e. minimum staleness) and the liveness guarantee of the system assuming the eventual delivery of all effects at all replicas. Detailed proofs of all theorems can be found in appendix B.

Before presenting the theorems, we define a ψ-consistent set of effects S under an execution state E as a set that is closed under $(R = \mathtt{r}_1; ...; \mathtt{r}_k)$, i.e.

$$S \texttt{ is } \psi\texttt{-consistent} \iff \forall(\eta \in S).\forall(a \in \mathsf{E}.\mathsf{A}).R(a,\eta) \Rightarrow a \in S \tag{6}$$

Theorem 1. *For all reduction steps* $(\mathsf{E}, op_{<s,i>}) \xrightarrow{V} (\mathsf{E}', \eta)$*, the followins hold:*

$$\begin{array}{l}(i)\ \texttt{If } V \texttt{ is } \psi\texttt{-consistent under } \mathsf{E}, \texttt{ then } V \cup \{\eta\} \texttt{ is } \psi\texttt{-consistent under } E' \\ (ii)\ E' \models \psi[\eta/\hat{\eta}]\end{array}$$

The above theorem states the preservation of ψ-consistency at replicas, under reduction stpes. Moreover, it states the correctness of the enforced consistency guarantee at the final execution state.

Theorem 2. *For all reduction steps* $(\mathsf{E}, op_{<s,i>}) \xrightarrow{V} (\mathsf{E}', \eta)$*, the set of effects made visible to* η *is maximal. i.e. for all* $a \in V$*, if* $\mathbb{SC}_a^{\mathsf{trunc}(R)}(V)$*, then*

$$(a,\eta) \notin \mathsf{E}'.\mathsf{vis} \Rightarrow (\mathsf{E}'.A, \mathsf{E}'.\mathsf{vis} \cup \{(a,\eta)\}, \mathsf{E}'.so) \not\models \psi[\eta/\hat{\eta}]$$

Theorem 3. *For all execution states* E*, if there exists a set of effects* $S \subseteq \mathsf{E.A}$*, such that:*

$$S \vdash (\mathsf{E}, op_{<s,i>}) \hookrightarrow (\mathsf{E}', \eta) \quad \wedge \quad (\mathsf{S} \cup \{\eta\} \texttt{ is } \psi\texttt{-consistent under } E')$$

then there exist E''*,* η' *and* V *such that:* $((\mathsf{E}, op_{<s,i>}) \xrightarrow{V} (\mathsf{E}'', \eta'))$

A trivial corollary of the above theorem is the liveness of our operational semantics, since at least one set S with the requested properties always exists[7] at any execution state.

6 Algorithm

In Sec. 5, we presented a high-level abstraction of our system behavior, where we explained *what* subset of effects at a replica, must be witnessed by every operation. In this section, we explain SYNCOPE's algorithm to efficiently maintain a *consistent cache*, in order to avoid redundancies in the filteration mechanism.

SYNCOPE maintains a consistent cache on top of each replica, by periodic reads from the underlying ECDS, where an effect η is moved to the cache, only if the cache already includes $\mathsf{trunc}(R)_V^{-1}(\eta)$. Consequently, all operations under UB contracts can be immediately executed by witnessing the cache, which is a closed subset (not necessarily maximal all the time) of V, the set of effects present at the replica. Moreover, LB contracts can also be satisfied, by blocking operations until effects of all previous operations from the same session enter the cache, in which case current operation can proceed and witness *all* effects present at the replica.

Additionally, we implemented a simple memoization technique in SYNCOPE, that extends the binary notion of dependency presence to the *degree of dependency presence* (DDP), which represents the maximum *depth* of the dependencies of an effect, whose presence have been verified so far. Consequently, when verifying the presence of dependencies for an effect fails, the runtime system can avoid redundant computations, next time it tries to verify the same property for the same effect. SYNCOPE's runtime, by performing periodic DDP refreshes, tries to assign larger DDP values to each effect while more dependencies arrive at the replica. Specifically, at each refresh the DDP of an effect η is increased from i to $i+1$ if all effects in $r_{i+1}^{-1}(\eta)$ already have DDP values at least equal to i.

For example, consider a contract with dependency relation $\mathsf{R} = \mathsf{so}; \mathsf{vis}; \mathsf{so}$, and a newly arrived effect η to the replica, whose DDP is initially set to 0. During the next refreshes, η is given the value 1, if all effects in $\mathsf{so}^{-1}(\eta)$ have DDP equal to 0 (i.e. are present at the replica). Similarly, η is given the value 2, if all effects in $\mathsf{vis}^{-1}(\eta)$ have DDP value of at least 1, which means that $(\mathsf{so}; \mathsf{vis})_V^{-1}(\eta)$ is now present at the replica and consequently, η can be safely added to the consistent cache (Fig.7). Using this technique, SYNCOPE avoid redundant computations of potentially large dependency relations.

---

[7] $S = \mathsf{E.A}$. This requires the preservation of ψ-consistency under the resuction step, that is already shown in theorem 1.

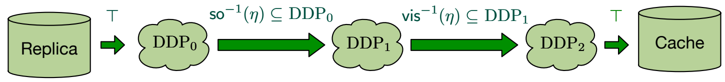


Fig. 7: Example of stepwise progress of effects before entering the cache

7 Evaluation

In this section we present an evaluation study of our implementation, including a report on benchmark applications that utilize fine-grained weak consistency requirements, expressable in SYNCOPE's specification language. Fig.8 presents seven of such programs, including individual data types as well as larger programs consisted of multiple data types.

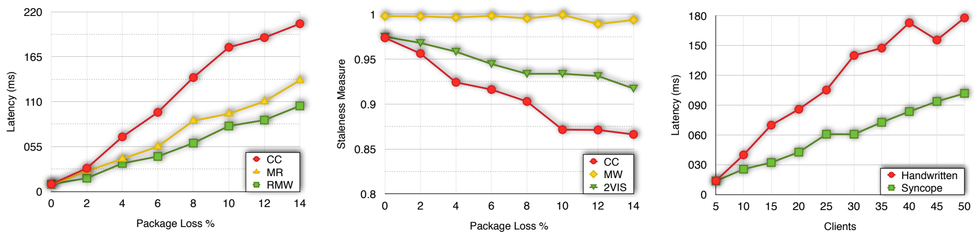
Each program offers various operations, each of which is assigned a potentially different consistency requirement, representing the need for a multi-consistnet environement for efficient execution of the programs. Surprisingly, we found no program intrinsically requiring causal consistency; all known consistency anomalies that operations may be involved in, are expressable with simple fine-grained contracts composed of dependency relations of length 1 or 2, which differs from what was knwon in the context before, where all such operations were considered to require CC.

Additionally, in many cases we found operations that may be involved in multiple anomalies, requiring simultaneous enforcement of different consistency guarantees, which shows the unfeasability of hand-writing such guarantees, considering the vast set of known consistency anomalies. For example, consider a bank account application, which offers `deposit`, `withdraw` and `get_balance` operations, where `withdraw` is a strongly consistent operation that succeeds only if there are sufficient funds in the account. There are two annomalous scenarios associate with `get_balance` in this program: (i) when a user performs a `deposit`, which is however, not refleceted in the subsequent `get_balance` (ii) when a `get_balance` witnesses a `withdraw` effect, without witnessing all the `deposit` effects that were visible to it, which may result in `get_balance` returning a negative balance. As it is presented in Fig. 8, in order to preempt these anomalies, `get_balance` requires both RMW and 2VIS guarantees simultaneously.

For our performance evaluation, we deploy SYNCOPE on a cloud cluster, consisting of three fully replicated Cassandra replicas, running on seperate machines

Benchmark	Consistency	Description
Counter	MR	Monotonicly increasing counter, e.g. YouTubes' watch count
DynamoDB	RMW	Integer register allowing various conditional puts and gets
Online Store	RMW	Online store with shopping carts and modifiable item prices
Bankaccount	2VIS $\wedge$ RMW	Offering deposit, withdraw and get balance operations
Shopping List	MW $\wedge$ RMW	A shopping list with concurrent adds and deletes functionality
Microblog	MW, RMW	A Twitter-like application modeled after Twissandra
Rubis	RMW, RMW$\wedge$2VIS	eBay-like application with browsing, supporting user wallet

Fig. 8: Fine-grained consistency requirement in benchmark programs



(a) Latency (b) Staleness (c) Manual RMW

Fig. 9: A distributed application for comment section management

within the same datacenter. Each machine is instantiated with a SYNCOPE shim layer, that responds to clients, which are instantiated on a virtual machine co-located with one of the replicas on a machine. We deploy the cluster on three `m4.4xlarge` Amazon EC2 instances in US-West (Oregon) region, with inter-machine communication time of 5ms.

Inter-replica communications in Cassandra use TCP connections, causing all messages get delivered with no loss and reordering, which is in practice, far more consistent than EC, and masks out the performance gain from our fine-grained consistency guarantees. Consequently, to simulate a realistic EC environment, we inject artificial message loss at the shim layers, where a message delivery is delayed for 1 second in case it is lost.

Fig. 9(a) and 9(b) represent our experimental results, with a workload generated by 50 concurrent clients repeatedly running sessions, each composed of three operations, where operations uniformly choose from 5 objects and are performed under the specified consistency level. We increase the percentage of delayed messages from 0 to 14, where each experiment ran for 100 repeaded sessions per client. Additional to client perceived latency, we also measure the staleness of operations, which we define as the average ratio of the number of visible effects, to the number of all available effects, at the time an operation is executed.

In the first set of experiments, we measure latency under three different LB contracts, all implemented in SYNCOPE. As expected, causal consistency and RMW experience respectively the highest and the lowest performance loss as the percentage of lost messages is increased[8]. At only 4 percent message loss rate, we see 17% higher latency under MR contract compared to RMW, and similarly 67% higher latency in CC compared to MR, whereas with 10 percent message loss, the numbers are increased to 18% and 87%.

Similarly, we repeated the experiment with 3 UB contracts, where *causal visibility* (CV) contract (i.e. $\forall a.a \xrightarrow{(\mathsf{so}\cup\mathsf{vis})^*;\mathsf{vis}} \hat{\eta} \Rightarrow a \xrightarrow{\mathsf{vis}} \hat{\eta}$), offers the most stale data when the percentage of lost messages is increased, whereas staleness in MW is the lowest and is barely effected. We report 3% (6%) difference between

[8] In fact, they define the storngest and the weakest LB dependency relations expressable in our language: $(\xrightarrow{\mathsf{so}})$ and $(\xrightarrow{(\mathsf{so}\cup\mathsf{vis})^*})$

staleness of data under MW and 2VIS, and 4% (7%) difference between 2VIS and CV, at four (ten) percante message loss rate.

Finally, in order to evidence the practicality of SYNCOPE, we implemented an ad-hoc mechanism to prevent lost-updates anomaly, for a simple counter application. Fig. 9(c) shows the latency results of this application compared to the same in SYNCOPE, under the same setting s before (albeit with no message loss). We report 78% higher latency for the handwritten code compared to SYNCOPE with 50 concurrent clients. We experienced many bookkeeping complications with the handwritten implementation, mainly because of the lack of meta-data queries in Cassandra which needs strongly consistent table alterations at the beginning and the end of each session, as mentioned before.

8 Related Works

9 Conclusion

References

1. KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. Declarative programming over eventually consistent data stores. *SIGPLAN Not.*, 50(6):413–424, June 2015.
2. Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer, and Brent B. Welch. Session guarantees for weakly consistent replicated data. pages 140–149, 1994.
3. Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Bolt-on causal consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 761–772, New York, NY, USA, 2013. ACM.
4. Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.
5. Mohsen Lesani, Christian J. Bell, and Adam Chlipala. Chapar: Certified causally consistent distributed key-value stores. *SIGPLAN Not.*, 51(1):357–370, January 2016.
6. James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: A framework for implementing and formally verifying distributed systems. *SIGPLAN Not.*, 50(6):357–368, June 2015.
7. Daniel J. Abadi. Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. IEEE Computer, 45(2), 2012.
8. M.P. Herlihy and J.M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.
9. Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, 1979.
10. Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *Operating Systems Review*, 44(2):35–40, 2010.

11. Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, pages 386–400, Berlin, Heidelberg, 2011. Springer-Verlag.
12. Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, August 2008.
13. Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.
14. P. Mahajan, L. Alvisi, and M. Dahlin. Consistency, availability, convergence. Technical Report TR-11-22, Computer Science Department, University of Texas at Austin, May 2011.
15. Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.
16. Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Providing high availability using lazy replication. *ACM Trans. Comput. Syst.*, 10(4):360–391, November 1992.
17. Kenneth Birman, André Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst.*, 9(3):272–314, August 1991.
18. Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM.

A Modified Haskell Program

```
data Sess = Bob | Alice
type ID = (Sess,Int)
type Effect= (ID,String)
type State = (String,Int,Int)

read :: ID -> State -> String
read (sess,seq) (st,sq1,sq2) =
 case sess of
  Bob ->    if (seq==sq1+1) then st
            else read (sess,seq)(st,sq1,sq2)
  Alice -> if (seq==sq2+1) then st
            else read (sess,seq)(st,sq1,sq2)
apply :: State -> Effect -> State
apply (st,sq1,sq2) ((sess,seq),cm) =
  case sess of
    Bob ->   if (sq1==seq-1)
             then (st++cm,sq1+1,sq2)
             else (st,sq1,sq2)
    Alice -> if (sq2==seq-1)
             then (st++cm,sq1,sq2+1)
             else (st,sq1,sq2)
```

Fig. 10: Guarded Application to Prevent Lost-updates Anomaly When Serving Bob and Alice

B Proofs

Here, we present the detailed proofs of the theorems of the paper. Let's first present a useful lemma:

Lemma 1. *For all relations R and execution steps:*

$$(\mathsf{E}, op_{<s,i>}) \xrightarrow{V} (\mathsf{E}', \eta)$$

interpretatin of R under E and E′ only differs considering η, i.e. $a, b \neq \eta \Rightarrow (R'(a,b) \Leftrightarrow R(a,b))$

Proof. We only prove ⇒, the other part can be done similarly. We have the following goal and hypotheses:

$$\begin{array}{l} H_0 : (\mathsf{E}, op_{<s,i>}) \xrightarrow{V} (\mathsf{E}', \eta) \\ H_1 : a, b \neq \eta \\ H_2 : R'(a,b) \\ G_0 : R(a,b) \end{array}$$

Now by destructing R we have the followings from new hypothesis and goal:

$$\begin{array}{l} H_3 : (\mathsf{trunc}(R); r)'(a,b) \\ G_1 : (\mathsf{trunc}(R); r)(a,b) \end{array}$$

which can be rewritten by the definition to get that y exists s.t.

$$\begin{array}{l} H_4 : (\mathsf{trunc}(R))'(a,y) \\ H_5 : r'(y,b) \\ G_1 : \exists x.(\mathsf{trunc}(R))(a,x) \wedge (r)(x,b) \end{array}$$

Now we instantiate the goal with y itself and by using induction on the length of R, the first conjunct is proved, and we are left with the following:

$$\begin{array}{l} H_6 : r'(y,b) \\ G_1 : r(y,b) \end{array}$$

Now by inversion on H_0 we get two cases, at both of which the following can be derived. (In one case V should be replaced by V' but has no effect on the proof):

$$\begin{array}{l} H_7 : \mathsf{vis}' = \mathsf{vis} \cup V \times \{\eta\} \\ H_8 : \mathsf{so}' = \mathsf{so}' = \mathsf{so} \cup \{(\eta', \eta) \mid \eta' \in \mathsf{A}_{(\mathsf{SessID}=s)} \end{array}$$

Now, because of H_1 (and the fact that $y \neq \eta$) it is easy to get the following from H_7 and H_8:

$$\begin{array}{l} H_9 : \ \mathsf{vis}(y,b) \Rightarrow \mathsf{vis}(y,b) = \\ H_{10} : \mathsf{so}(y,b) \Rightarrow \mathsf{so}(y,b) = \end{array}$$

Which directly prove the goal, after destructing r.

B.1 Proof of Theorem 1

(Part i) We have the following two hypotheses and the goal:

$$\frac{\begin{array}{l} H_0 : (\mathsf{E}, op_{<s,i>}) \xrightarrow{V} (\mathsf{E}', \eta) \\ H_1 : V \texttt{ is } \psi\texttt{-consistent under } \mathsf{E} \end{array}}{G_0 : V \cup \{\eta\} \texttt{ is } \psi\texttt{-consistent under } \mathsf{E}'}$$

Rewriting the definition in G_0 results in the following. We denote the interpretation of R under E' as R':

$$G_1 : \forall(b \in V \cup \{\eta\}).\forall(a \in E'.A).R'(a,b) \Rightarrow a \in V \cup \{\eta\}$$

By intros we have:

$$\frac{\begin{array}{l} H_2 : b \in V \cup \{\eta\} \\ H_3 : a \in E'.A \\ H_4 : R'(a,b) \end{array}}{G_2 : a \in V \cup \{\eta\}}$$

by inversion on H_0, there is two cases, in case one (UB reduction) we have the following:

$$T_1 : V' \vdash (\mathsf{E}, op_{<s,i>}) \hookrightarrow (\mathsf{E}', \eta)$$

by inversion on T_1 we will have the following:

$$T_2 : E'.A = E.A \cup \{\eta\}$$

Since the other case (LB reduction) also includes similar premises which yields T_2, we can add it to the hypothesis:

$$H_5 : E'.A = E.A \cup \{\eta\}$$

by rewriting H_5 in H_3 and by inversion, we get two cases: $a = \eta$ and $a \in E.A$. The first case immediatly proves G_2, so we only consider the second case where we have:

$$H_6 : a \in E.A$$

Now, by inversion on H_2, we have two cases:

- **Case 1:**

  $$b \in V$$

  by inversion in H_1 we have:

  $$H_7 : \forall(x \in V).\forall(y \in E.A).R(y,x) \Rightarrow y \in V$$

  by instantiation with a and b:

  $$H_8 : R(a,b) \Rightarrow a \in V$$

  Now by applying the lemma 1 on H_4 we get that $R(a,b)$ holds (since $a, b \neq \eta$), which can be applied on H_8 to get $a \in V$ which proves the goal G_2.
- **Case 2:**

  $$\begin{array}{rl} & H_9 : \; b = \eta \\ (\texttt{by rewriting } H_9 \texttt{ in } H_4) & H_{10} : R'(a,\eta) \end{array}$$

  Now we use inversion on H_0 and get two cases: (LB exec) and (UB exec)
  - **SCase (LB exec):** we have H_{11} and H_{12} from the reduction rule premises:

    $$H_{11} : R_V^{-1}(\eta) = R_{E'.A}^{-1}(\eta)$$
    $$H_{12} : R_V^{-1}(\eta) \subseteq V$$

now from H_{10} we have H_{13} which can be rewritten by H_{11} to get H_{H14}:

$$H_{13} : a \in R^{-1}_{E'.A}(\eta)$$
$$H_{14} : a \in R^{-1}_{V}(\eta)$$

The goal G_2 is now proved from H_{12} and H_{14}.

– **SCase (UB exec):** We have the following from the premises:

$$H_{15} : V' = \lfloor V \rfloor_{\mathtt{max}}$$
$$H_{16} : V' \subseteq V$$

now destruct R, the only non-trivial cases are $(R = \mathsf{trunc}(R); \mathsf{vis})$ and $(R = \mathsf{vis})$:
SSCase $(R = \mathsf{trunc}(R); \mathsf{vis})$:
From H_{10} we get H_{17} which based on the definition, yields that there exists c such that H_{18}, H_{19} and H_{20} hold:

$$H_{17} : a \in (\mathsf{trunc}(R)'; \mathsf{vis}')^{-1}_{E'.A}(\eta)$$
$$H_{18} : c \in \mathsf{vis}'^{-1}(\eta)$$
$$H_{19} : a \in \mathsf{trunc}(R)'^{-1}(c)$$
$$H_{20} : \mathsf{vis}'^{-1}(\eta) \subseteq E'.A$$

from H_{15} we have:

$$H_{21} : (\mathsf{trunc}(R))^{-1}_{V}(V') \subseteq V'$$

Now from H_{18} is straightforward to get:

$$H_{22} : c \in V'$$

which after appying the lemma 1 on H_{19}, and by H_{21} yields the following, which proves the goal G_2:

$$H_{23} : a \in V'$$

SSCase $(R = \mathsf{vis})$: From H_{10} we get that $\mathsf{vis}'(a, \eta)$, which -with a similar argument to the previous subcase- yields the following and the goal is proved:

$$H_{24} : a \in V'$$

QED.

(Part ii)
For this part we have the following hypothesis and the goal:

$$H_0 : (\mathsf{E}, op_{<s,i>}) \xrightarrow{V} (\mathsf{E}', \eta)$$
$$G_0 : E' \models [\eta/\hat{\eta}]$$

By inversion on H_0, we have two cases:
Case1 (UB exec):

$$H_1 : r_k = \mathsf{vis}$$
$$H_2 : V \subseteq E.A$$
$$H_3 : V' = \lfloor V \rfloor_{\mathtt{max}}$$
$$H_4 : V' \vdash (\mathsf{E}, op_{<s,i>}) \hookrightarrow (\mathsf{E}', \eta)$$

The goal G_0 can be rewritten as:

$$G_1 : E' \models \forall a.a \xrightarrow{R} \eta \Rightarrow a \xrightarrow{vis} \eta$$

Since the $E'.A$ gives the interpretation for the universe of quantification:

$$G_2 : \forall(a \in E'.A).E' \models a \xrightarrow{R} \eta \Rightarrow a \xrightarrow{vis} \eta$$

by intros:

$$H_5 : a \in E'.A$$
$$G_3 : E' \models a \xrightarrow{R} \eta \Rightarrow a \xrightarrow{vis} \eta$$

Now since $\big((\mathcal{M} \models A \Rightarrow B) \Leftrightarrow (\mathcal{M} \models A \Rightarrow \mathcal{M} \models B)\big)$ we can rewrite G_3 as:

$$G_4 : (E' \models a \xrightarrow{R} \eta) \Rightarrow (E' \models a \xrightarrow{vis} \eta)$$

intros:

$$H_6 : E' \models a \xrightarrow{R} \eta$$
$$G_5 : E' \models a \xrightarrow{vis} \eta$$

Now we use the interpretation given by E', to rewrite the relations as follows. Note that we denote the interpretation of R under E' as R' and $E.\mathsf{vis}$ as vis'.

$$H_7 : R'(a, \eta)$$
$$G_6 : \mathsf{vis}'(a, \eta)$$

by inversion on H_4:

$$H_8 : \mathsf{vis}' = \mathsf{vis} \cup V' \times \{\eta\}$$

Now since η is a fresh effect, we get that $a \in V' \Rightarrow \mathsf{vis}'(a, \eta)$ which can be applied to G_6 to get the following:

$$G_7 : a \in V'$$

Now, destructing R yileds multiple cases, only one of which is non-trivial: $R = \mathsf{trunc}(R); \mathsf{vis}$, which can be rewritten in H_7 to get:

$$H_9 : (\mathsf{trunc}(R); \mathsf{vis})'(a, \eta)$$

Now we can rewrite the definition in H_9, and derive that there exists b such that:

$$H_{10} : \mathsf{trunc}(R)'(a, b)$$
$$H_{11} : \mathsf{vis}'(b, \eta)$$

Now using a similar argument, from H_8 and H_{11} we get:

$$H_{12} : b \in V'$$

Now by applying the lemma 1 on H_{10} we get:

$$H_{13} : \mathsf{trunc}(R)(a, b)$$

since we have $V' \in \lfloor V \rfloor$, we get the following:

$$H_{14} : \forall(x \in V').(\mathsf{trunc}(R))^{-1}_{E.A}(V') \Rightarrow x \in V'$$

which yields the following from H_{12} and H_{13}:

$$H_{15} : a \in V'$$

which proves the goal G_7.

Case2 (LB exec):

We prove this case by induction on the length of the given relation R. We have the followings, from the premises of the reduction rule:

$$\begin{array}{l} H_1 : r_k = \mathsf{so} \\ H_2 : V \subseteq E.A \\ H_3 : R_V^{-1}(\eta) = R_{E.A}^{-1}(\eta) \\ H_4 : R_V^{-1}(\eta) \subseteq V \\ H_5 : V \vdash (\mathsf{E}, op_{<s,i>}) \hookrightarrow (\mathsf{E}', \eta) \end{array}$$

Using the same argument as the previous section, we get the following new goal and hypotheses:

$$\begin{array}{l} H_6 : a \in E'.A \\ H_7 : R'(a, \eta) \\ G_1 : \mathsf{vis}'(a, \eta) \end{array}$$

We now destruct R to get H_8 from H_7, and rewrite the definition in it to get the next two hypotheses. Note that by destructing R, there are only two non-trivial cases $R = \mathsf{trunc}(R); \mathsf{so}$ and $R = \mathsf{so}$, which we are only consideing the former, since the latter can be proved similarly.

$$\begin{array}{l} H_8 : \ (\mathsf{trunc}(R); \mathsf{so})'(a, \eta) \\ H_9 : \ \mathsf{trunc}(R)'(a, b) \\ H_{10} : \mathsf{so}'(b, \eta) \end{array}$$

Now, from the previous section we know that $(so')^{-1}(\eta) \subseteq V$ which yields the following from H_{10}:

$$H_{11} : b \in V$$

The goal is proved by the induction hypothesis, H_9 and H_{11}.
$\boxed{\text{QED.}}$

B.2 Proof of Theorem 2

We prove the theorem by contradiction:

$$\begin{array}{l} H_0 : (\mathsf{E}, op_{<s,i>}) \xrightarrow{V} (\mathsf{E}', \eta) \\ H_1 : a \in V \\ H_2 : (a, \eta) \notin E'.\mathsf{vis} \\ H_3 : (E'.A, E'.\mathsf{vis} \cup \{(a, \eta)\}, E'.\mathsf{so}) \models \psi[\eta/\hat{\eta}] \\ H_4 : (\mathsf{trunc}(R)_V^{-1}(a) = \mathsf{trunc}(R)_{E.A}^{-1}(a)) \\ G_0 : \perp \end{array}$$

Now we call $(E'.A, E'.\mathsf{vis} \cup \{(a, \eta)\}, E'.\mathsf{so})$ as E'' and derive the following from H_3:

$$H_5 : E'' \models \forall x.x \xrightarrow{R} \eta \Rightarrow x \xrightarrow{\mathsf{vis}} \eta$$

because E'' defines the universe of quantification (and since $E''.A = E'.A$), we get the following:

$$H_6 : \forall (x \in E'.A).E'' \models x \xrightarrow{R} \eta \Rightarrow x \xrightarrow{\mathsf{vis}} \eta$$

and is rewritten as the following:

$$H_7 : \forall (x \in E'.A).(E'' \models x \xrightarrow{R} \eta) \Rightarrow (E'' \models x \xrightarrow{\mathsf{vis}} \eta)$$

Now by inversion on H_0 we get two cases, one of which is trivial. We skip the formal proof for it but it is easy to see that in [LB exec] case, ALL effects in V are made visible to η, so the set is trivially maximal, i.e. H_1 and H_2 yield $\perp$. For the other case (UB exec), we get the following:

$$H_8 : V' = \lfloor V \rfloor_{\max}$$
$$H_9 : V' \vdash (\mathsf{E}, op_{<s,i>}) \hookrightarrow (\mathsf{E}', \eta)$$

by inversion on H_9 we get H_{10} and from that and from H_2, following a similar argument from the proof of theorem 1, we get H_{11}:

$$H_{10} : \mathsf{vis}' = \mathsf{vis} \cup V' \times \{\eta\}$$
$$H_{11} : a \notin V'$$

Now by denoting the interpretation of R under E'' as R'', H_7 can be rewritten as follows:

$$H_{12} : \forall (x \in E'.A). R''(x, \eta) \Rightarrow \mathsf{vis}''(x, \eta)$$

Now by inversion on H_8, we get the following:

$$H_{13} : V' \in \lfloor V \rfloor$$
$$H_{14} : \nexists V'' \in \lfloor V \rfloor . |V''| > |V'|$$
$$(\texttt{from } \mathtt{H}_{13})\ H_{15} : V' \subseteq V \ \wedge \ (\mathsf{trunc}(R))_V^{-1}(V') \subseteq V' \ \wedge$$
$$(\mathsf{trunc}(R))_V^{-1}(V') = (\mathsf{trunc}(R))_{E.A}^{-1}(V')$$

Now we can destruct R, where we get multiple cases, only two of which are non-trivial, $(R = \mathsf{vis})$ and $(R = \mathsf{trunc}(R); \mathsf{vis})$

– **Case1**$(R = \mathsf{vis})$:
  $\mathsf{trunc}(R) = \mathtt{null}$, thus V itself satisfies the requirements in H_{15} and we get that $(V = \lfloor V \rfloor_{\max})$ and the following holds:

  $$H_{16} : V = V'$$

  which results in contradiction from H_1 and H_{11}.

– **Case2**$(R = \mathsf{trunc}(R); \mathsf{vis})$:
  Since $|V' \cup \{a\}| > |V'|$ we have the following:

  $$H_{17} : (V' \cup \{a\}) \notin \lfloor V \rfloor$$

  which based on the definition yields that the conditiones for holding the above relation are not true, i.e.

  $$H_{18} : \neg((V' \cup \{a\}) \subseteq V \ \wedge \ (\mathsf{trunc}(R))_V^{-1}(V' \cup \{a\}) \subseteq (V' \cup \{a\}) \ \wedge$$
  $$(\mathsf{trunc}(R))_V^{-1}(V' \cup \{a\}) = (\mathsf{trunc}(R))_{E.A}^{-1}(V' \cup \{a\}))$$

  or equally:

  $$H_{19} : (V' \cup \{a\}) \nsubseteq V \ \vee$$
  $$(\mathsf{trunc}(R))_V^{-1}(V' \cup \{a\}) \nsubseteq (V' \cup \{a\}) \ \vee$$
  $$(\mathsf{trunc}(R))_V^{-1}(V' \cup \{a\}) \neq (\mathsf{trunc}(R))_{E.A}^{-1}(V' \cup \{a\})$$

  By inversion on the above, we get three cases, two of which are trivial. The last conjunct can't hold because of H_4 and the first one also contradics with H_1 and H_{15}. Thus, we are left with only one case:

  $$H_{20} : (\mathsf{trunc}(R))_V^{-1}(V' \cup \{a\}) \nsubseteq (V' \cup \{a\})$$

  Now, from the second conjunct in H_{15} we know that it should be the case that:

  $$(\texttt{from } \mathtt{H}_{15} : (\mathsf{trunc}(\mathtt{R}))_{\mathtt{V}}^{-1}(\mathtt{V}') \subseteq \mathtt{V}') \quad H_{21} : ((\mathsf{trunc}(R))_V^{-1}(a) \nsubseteq (V' \cup \{a\}))$$

The above hypothesis yields the existance of $c \neq a$ such that:

$$H_{22} : c \in (\mathsf{trunc}(R))_V^{-1}(a)$$
$$H_{23} : c \notin V'$$

Now, by rewriting ($R = \mathsf{trunc}(R); \mathsf{vis}$) in H_{12} we get H_{24}, which can be rewritten again into H_{25} from the definition:

$$H_{24} : \forall(x \in E'.A).((\mathsf{trunc}(R); \mathsf{vis})''(x, \eta) \Rightarrow \mathsf{vis}''(x, \eta))$$
$$H_{25} : \forall(x \in E'.A).(\exists b.\mathsf{trunc}(R)''(x, b) \wedge \mathsf{vis}''(b, \eta) \Rightarrow \mathsf{vis}''(x, \eta))$$

Now, we instantiate H_{25} with $x = c$:

$$H_{26} : \exists b.\mathsf{trunc}(R)''(c, b) \ \wedge \ \mathsf{vis}''(b, \eta) \Rightarrow \mathsf{vis}''(c, \eta)$$

we can replace $\mathsf{trunc}(R)''$ with $\mathsf{trunc}(R)'$ in above definition, since from H_3, the only difference in interpretation under E' and E'' is the extra element (a, η) in $E''.\mathsf{vis}$ which does not effect $\mathsf{trunc}(R)''(c, b)$:

$$H_{27} : \exists b.\mathsf{trunc}(R)'(c, b) \ \wedge \ \mathsf{vis}''(b, \eta) \Rightarrow \mathsf{vis}''(c, \eta)$$

Moreover, since $c \neq a$, we can replace $\mathsf{vis}''(c, \eta)$ with $\mathsf{vis}'(c, \eta)$:

$$H_{28} : \exists b.\mathsf{trunc}(R)'(c, b) \ \wedge \ \mathsf{vis}''(b, \eta) \Rightarrow \mathsf{vis}'(c, \eta)$$

From H_{15} and H_{22} we get H_{29}, and H_{30} also holds trivially from H_3:

$$H_{29} : \mathsf{trunc}(R)'(c, a)$$
$$H_{30} : \mathsf{vis}''(a, \eta)$$

which can be used in instantiation of H_{28} with $b = a$ and derive the following:

$$H_{31} : \mathsf{vis}'(c, \eta)$$

However, we know -from the previously explained argument- that H_{31} results in H_{32}, which results in contradiction with H_{23}.

$$H_{32} : c \in V'$$

QED.

B.3 Proof of Theorem 3

Before proving the theorem, we first present and prove a useful lemma and then we will present a new definition, regarding sets of effects.

Lemma 2. *Under an execution state E and for a given set $S \subseteq E.A$, if S is ψ-consistent under E, then $\forall(x \in S).R_S^{-1}(x) \subseteq S$ under E.*

Proof.

$$H_0 : Sis\psi\mathtt{-consistent}$$
$$G_0 : \forall(x \in S).R_S^{-1}(x) \subseteq S$$

after intros:

$$H_1 : x \in S$$
$$G_1 : R_S^{-1}(x) \subseteq S$$

inversion on H_0 gives the following:

$$H_2 : \forall(\eta \in S).\forall(a \in E.A).R(a,\eta) \Rightarrow a \in S$$

which can be rewritten to:

$$H_3 : \forall(\eta \in S).R^{-1}(\eta) \subseteq S$$

however, since $S \subseteq E.A$ then[9]:

$$H_4 : \forall(a \in E.A).R_S^{-1}(a) \subseteq R^{-1}(a)$$

Now we can instantiate H_3 and H_4 into:

$$H_5 : R^{-1}(x) \subseteq S$$
$$H_6 : R_S^{-1}(x) \subseteq R^{-1}(x)$$

which trivially yields G_1 and the proof is completed.
QED.

Definition 1. *We define the completment of a given set of effects S (under an execution state E) as the super set of S, containing ALL the mid-level effects required to determine ALL the dependencies of the effects in S, i.e.*

$$S' \in \lceil S \rceil \iff R_{S'}^{-1}(S) = R_{E.A}^{-1}(S)$$

Now, using the above theorem and lemma, we present the proof of the theorem 3, which starts by listing the following hypotheses and the goal:

$$H_0 : S \vdash (\mathsf{E}, op_{<s,i>}) \hookrightarrow (\mathsf{E}', \eta)$$
$$H_1 : S \cup \{\eta\} \; is \; \psi\text{−}\texttt{consistent}$$
$$G_0 : \exists E''.\exists\eta'.\exists V.((\mathsf{E}, op_{<s,i>}) \xrightarrow{V} (\mathsf{E}'', \eta'))$$

Now, by destructing R we get two non-trivial cases:

- **Case1**($R = \mathsf{trunc}(R); \mathsf{vis}$):
  In this case[10], we generate the premises of the [UB EXEC] to achieve the goal as follows. Firstly, we define S' and present η':

  $$H_3 : S' = \lfloor S \rfloor_{\mathsf{max}}$$
  $$H_4 : \eta' = (s, op, F_{op}(S'))$$

  Moreover, we will define the followings, which will be used when presenting E'':

  $$H_5 : \mathsf{so}'' = \mathsf{so} \cup A_{(\texttt{SessID=s})} \times \{\eta'\}$$
  $$H_6 : \mathsf{vis}'' = \mathsf{vis} \cup S' \times \{\eta'\}$$
  $$H_7 : A'' = E.A \cup \{\eta'\}$$

  Now we present V and E'' as follows and rewrite the goal:

  $$H_8 : V = S$$
  $$H_9 : E'' = (A'', \mathsf{so}'', \mathsf{vis}'')$$
  $$G_1 : (\mathsf{E}, op_{<s,i>}) \xrightarrow{V} (\mathsf{E}'', \eta')$$

---

[9] we skip the formal proof of this claim, however, since the only difference in the definitions of R^{-1} and R_S^{-1} is the extra requirement about mid-level effects in the latter, it should be a subset of the former.

[10] Note that in this case the goal G_0, trivially holds. That is because the contract in this case is [UB], which represents executions whitout blocking or waiting, that can always make progress by showing *some* set of effects to the operations

by applying [UB EXEC] on G_1 we get the following new goals (after rewriting H_9 and H_3):

$$\begin{aligned}
&G_2 : r_k = \mathsf{vis}\\
&G_3 : S \subseteq E.A\\
&G_4 : S' = \lfloor S \rfloor\\
&G_5 : S' \vdash (\mathsf{E}, op_{<s,i>}) \hookrightarrow (\mathsf{E}'', \eta')
\end{aligned}$$

first three goals are proved via the assumptions, and the last one can be easily shown to hold by applying [OPER] and deriving the following new goals:

$$\begin{aligned}
&G_6 : \ S' \subseteq E.A\\
&G_7 : \ F_{op}(S') = v\\
&G_8 : \ \eta' = (s, op, v)\\
&G_9 : \ \eta \notin S'\\
&G_{10} : E''.A = E.A \cup \{\eta'\}\\
&G_{11} : E''.\mathsf{vis} = E.\mathsf{vis} \cup S' \times \{\eta\}\\
&G_{12} : E''.\mathsf{so} = E.\mathsf{so} \cup (A_{(\mathtt{SessID=s})}) \times \{\eta\}
\end{aligned}$$

all the above goals have already been shown in the assumptions and the case is proved.

– **Case2**$(R = \mathsf{trunc}(R); \mathsf{so})$:
Similarly in this case we define the following:

$$H_{13} : V = \lceil S \cup \{\eta\} \rceil$$

which yields:

$$H_{14} : \forall (x \in S \cup \{\eta\}). R_V^{-1}(x) = R_{E'.A}^{-1}(x)$$

and also:

$$H_{15} : R_V^{-1}(\eta) = R_{E'.A}^{-1}(\eta)$$

Similar to the previous case, we now define the followings:

$$\begin{aligned}
&H_{16} : \eta' = (s, op, F_{op}(V))\\
&H_{17} : \mathsf{so}'' = \mathsf{so} \cup A_{(\mathtt{SessID=s})} \times \{\eta'\}\\
&H_{18} : \mathsf{vis}'' = \mathsf{vis} \cup V \times \{\eta'\}
\end{aligned}$$

Now we presentE'' as follows and rewrite the goal:

$$\begin{aligned}
&H_{19} : E'' = (A'', \mathsf{so}'', \mathsf{vis}'')\\
&G_1 : \ \ (\mathsf{E}, op_{<s,i>}) \xrightarrow{V} (\mathsf{E}'', \eta')
\end{aligned}$$

by applying [LB EXEC] on G_1 we get the following new goals

$$\begin{aligned}
&G_2 : r_k = \mathsf{so}\\
&G_3 : V \subseteq E.A\\
&G_4 : R_V^{-1}(\eta') = R_{E''.A}^{-1}(\eta')\\
&G_5 : R_V^{-1}(\eta') \subseteq V\\
&G_6 : V \vdash (\mathsf{E}, op_{<s,i>}) \hookrightarrow (\mathsf{E}'', \eta')
\end{aligned}$$

Now, G_2 and G_3 are trivially proved from the assumptions, and G_6 also can be easily proved following the argument from the previous case. We prove G_4 and G_5, by a new claim that $R_{E'.A}^{-1}(\eta) = R_{E''.A}^{-1}(\eta')$ which will be proved separately. Thus, we can rewrite the goals and add the new claim:

$$\begin{aligned}
&G_7 : R^{-1}(\eta) = R_{E'.A}^{-1}(\eta)\\
&G_8 : R^{-1}(\eta) \subseteq V\\
&G_9 : R_{E'.A}^{-1}(\eta) = R_{E''.A}^{-1}(\eta')
\end{aligned}$$

Now G_7 is equal to the assumption H_{15}, and G_8 is the direct result of applying the lemma 2 on H_1. Now by rewriting $R = \mathsf{trunc}(R); \mathsf{so}$ in G_9 we have the following:

$$G_{10} : (\mathsf{trunc}(R); \mathsf{so})^{-1}_{E'.A}(\eta) = (\mathsf{trunc}(R); \mathsf{so})^{-1}_{E''.A}(\eta')$$

Now, note that the only difference in E' and E'' is in how the update the vis relation from E, the former makes the set S visible to the operation and the latter the set $\lceil S \cup \{\eta\} \rceil$. Now since the given relation R ends with an so relation, it is straigtforward to show that $G_1 0$ holds and thus the case (and the theorem) is proved.

QED.