Operational Semantics and Logic for Weak Consistency in Transactional Systems*

Subtitle†

ANONYMOUS AUTHOR(S)

Contents of this set of notes: History heaps. Semantics of Programs running under weak consistency models using history heaps as states. Simulation technique for comparing weak consistency models defined using history heaps. Verification of implementations. **Points following Dagstuhl: Viktor seemed positive about the history heap work. His question was whether the framework is generic enough to capture the protocols that they are developing with Azalea. Alexey's opinion is that the framework may have some use if we manage to prove implementations of protocols correct. I would also like to have Azalea's opinion on a semantics based on history heaps.**

Additional Key Words and Phrases: keyword1, keyword2, keyword3

AR: I have imported the cleveref package! This means that all reference will be printed consistently and we DO NOT need custom names such as \fig etc. Every time you need to refer to something, please write\cref{label}, e.g. Theorem 2.1, and the label (e.g. Def.) will be printed correctly. These labels can be customised. I have introduced the necessary ones in the macros file.

1 INTRODUCTION

AC: Obviously this will be rewritten again and again, but it should give an idea of how the paper is going to be structured.

Modern distributed systems s often rely on databases that achieve scalability by weakening consistency guarantees of distributed transaction processing. These databases are said to implement weak consistency models. Such weakly consistent databases allow for faster transaction processing, but exhibit anomalous behaviours, which do not arise under a database with a strong consistency guarantee, such as *serialisability*.

Recently, the research community has made an effort to give formal specifications of weak consistency models for transactional distributed databases [??????]. However, the problem of giving the semantics describing the operational behaviour of clients interacting with a weakly consistent databases, has been largely neglected. The only work that we are aware of in this field is given by [?]. There the authors study the behaviour of functional programs interacting with a relational database, and develop a program logic for proving invariants of programs executed by clients of the database.

AC: Insert sentence to swiftly kill their paper without being an asshole here.

*Title note
†Subtitle note

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

AR: It needs to be a bit more than a sentence. We need to point out that:

- The lack of soundness/completeness result for the operational semantics is a serious shortcoming: it has no relation to existing formal semantics for e.g. SI. The lack of completeness result is particularly damaging as it may prohibit many results allowed by the underlying consistency model. This in turn damages the usability of their logic as many behaviours cannot be verified.
- We should also point out that the logic is not so much a logic but a bunch of semantic rules, i.e. no syntactic support. Also that their logic is based on first order logic, and offers no compositionality. Lastly, that it is geared towards databases and as such cannot be adapted for shared memory STMs.

In this paper we focus on the semantics of clients of distributed key-value stores which provide weak consistency guarantees. We propose a *coarse-grained* approach to evaluating transactions, where transactions are executed by clients in a single, atomic step; furthermore, transactions of concurrent clients are executed in an interleaving fashion. This is In contrast with the work of [**?**], which takes a fine-grained approach in which transactions are evaluated one step at the time on a local copy of the database, and thus it is possible to interleave the execution of transaction; we find this complication to be unnecessary, at least in a setting where transactions enjoy *atomic visibility*: either none or all the updates of a transaction are made visible to another transaction.

We take a multi-version approach to model the state of a key-value store. Each key is mapped to a set of versions. A Version consists of a value and the meta-data of the transactions that wrote and read the version. At any given instant of time, concurrent clients can observe different versions for the same key. This approach is necessary to capture the non-serialisable behaviour of programs, while still retaining interleaving concurrency and atomic reduction of transactions.

Because we want to model different consistency models, in our semantics transactions are executed only prior to passing a particular execution test. For example, to constrain a program to only exhibit serialisable behaviours, we require that transactions can be executed by a client only if it observes the most recent available version for each key. By tweaking the execution test of transactions, we tweak the consistency model under which programs are executed. In **??** we give examples of execution tests that can be used to model all the consistency models formalised in [**?**]. The notion of execution test is inspired by [**?**]. There a notion of *commit test* is introduced to determine whether a transaction can be executed safely. However, the notion of commit test requires the complete knowledge of how the system of the key-value store evolved from its initial state (i.e. it requires knowing the total order in which transactions executed in the computation); execution tests, on the other hand, only require knowing the list of versions stored by each key, and the information about which version of each key is observed by the client executing the transaction.

AR: I'm worried that these execution tests are too technical for the intro? If we decide to keep it we should expand the description a little bit more.

AC: Paragraph about the thoroughness of the semantics and correspondence of our specifications and the declarative ones from the CONCUR'15 paper. Paragraph about the logic. Here we should stress that clients that execute correctly under serialisability, are not necessarily correct under a weaker consistency model. There should be a sentence explaining that thoroughness of the semantics is necessary to obtain soundness of the logic.

Contributions of the paper:

(1) An operational semantics of programs interacting with a key-value store;

(2) Definition of different execution tests for capturing several consistency models, and a proof that the consistency models captured by the execution tests we propose are *sound and complete* (equivalent) with respect to their respective respective declarative specifications given in[?];

AR: Perhaps enumerate them here? e.g. SI, PSI, serialisability, etc.

(3) A proof that the operational semantics is *thorough*: for each of the consistency models we formalise, and for any program P, our semantics captures all the behaviours that P can display under said consistency model;

AR: How is thoroughness different from completeness? I am not sure I understand this. Perhaps this needs clarification?

(4) A separation logic based on our semantics to reason about properties of clients interacting with a weakly consistent key-value store.

AC: Got bored of writing the introduction, in any case it will need to be changed a lot in the future.

2 SEMANTICS

SX: program vs. transaction

We focus on an abstract computational model where multiple client programs can access and update keys in a key-value store using atomic transactions. Transactions in our model execute atomically, though they have different effects on the key-value stores depending on their associated *consistency model*. A consistency model controls how the key-value store evolves. A common model is *serialisability*, where transactions appear to execute one after another in a sequential order. This notion of sequential execution is however not necessary for many weaker models. As such, upon commencing execution, a transaction may not observe the most up-to-date values for keys.

To address this, we first model the state of the system using *multi-version key-value stores (MKVSs)* (§2.1). An MKVS keeps track of all versions (values) written for keys, as well as the information about the transactions that read and wrote such versions. To model the potential out-of-date observation, we introduce *views*. A view decides the observable versions of keys for a client. Therefore, in order to execute a transaction, the client first takes *a snapshot* of the system with the view, executes the transaction locally with respect to its snapshot (§2.2.2), and afterwards commits the effect of the transaction if the change is allowed by the underlying consistency model (§2.2.3).

2.1 Multi-version Key-value Stores and Views

AR: I have paraphrased quite a bit here. Please make sure you are happy with this.

AC:

SX: Partial is better for logic

Maybe it's better to keep k fixed and say that we look at only a fragment of the key value store. Alternatively, we can go for partial mappings to represent MKVSs, but still avoiding allocation and deallocation of keys.

.

AR: I have changed the values to v_0 and v_1 (from 0 and 1) to help clarify the distinction between indexes and values. I have also paraphrased, please double check.

We model the state of a system using *multi-version key-value stores* (MKVSs). An MKVS is a map from keys to lists of versions (values). More concretely, each version associated with a key k

is a tuple of the form $\vartheta = (n, t, \mathcal{T})$, where n is a natural number denoting the *current value* of k, the t is a transaction identifier t, identifying its *writer*, i.e. the transaction responsible for writing value n, and $\mathcal{T}$ is a set of transaction identifiers, denoting its *readers*, i.e. those transactions who read from k (see Theorem 2.1). Fig. 1a depicts an example of an MKVS. Ignoring the lines labelled cl_1 and cl_2, the depicted MKVS contains two keys k_1 and k_2, each of which associated with two versions (with values v_0 and v_1 for k_1, and values v_0' and v_1' for k_2). The versions of a key are listed in chronological order from left to right. We represent each version as a three-cell box, with the left cell storing the value, the top right cell recording the writer, and the bottom right cell recording the readers.

Given a version $\vartheta = (n, t, \mathcal{T})$, we write $\mathsf{Value}(\vartheta)$ for its value (n), $\mathsf{Write}(\vartheta)$ for its writer (t), and $\mathsf{Reads}(\vartheta)$ for its readers ($\mathcal{T}$). Given a list of versions $L = [\vartheta_0, \cdots, \vartheta_{s-1}]$, we write $|L|$ for its length (s). Given an MKVS kv and a key k, we write $kv(k)$ for the list of versions associated with k in kv, and write $kv(k, i)$ for the i^{th} entry (indexed from 0) in $kv(k)$.

AR: I think kv is too close to k and we should use a different symbol. Maybe H?

We assume a countably infinite set of *keys*, Keys, and a countably infinite set of *transactions identifiers*, TransID. We use k and its variants (e.g. k_1, k' and so forth) as meta-variables for keys in Keys; and use t and its variants as meta-variables for transaction identifiers in TransID.

Definition 2.1 (Multi-version key-value stores). A *multi-version key-value store (MKVS)*, $kv \in \text{MKVSs}$, is a partial finite function from keys to lists of *versions*. A *version* is a triple containing a natural number n, a transaction identifier t and a set of transactions identifiers $\mathcal{T}$:

$$\vartheta \in \textsc{Versions} \triangleq \left\{(n, t, \mathcal{T}) \;\middle|\; n \in \mathbb{N} \wedge t \in \textsc{TransID} \wedge \mathcal{T} \subseteq \textsc{TransID} \wedge t \notin \mathcal{T}\right\}$$
$$kv \in \text{MKVSs} \triangleq \textsc{Keys} \xrightarrow{fin} \textsc{Versions}^*$$

Given two versions ϑ_1, ϑ_2 in an MKVS kv, the ϑ_1 *directly depends* on ϑ_2, written $\vartheta_1 \xrightarrow{\text{ddep}} \vartheta_2$, iff $\mathsf{Write}(\vartheta_2) \in \mathsf{Reads}(\vartheta_1)$; that is, a transaction t wrote the version ϑ_2 after reading ϑ_1.

An MKVS is *well-formed*, written $\mathtt{wfHH}(kv)$, iff i) it does not contain circular dependencies across its versions; and ii) a transaction identifier appears in all versions of a key at most twice, once as the writer and once as a reader. More concretely, the first well-formedness condition (i) of an KVMS ensures that the *transitive closure* of the direct dependency relation $\left(\xrightarrow{\text{ddep}(kv)}\right)^+$ is acyclic:

$$\mathtt{wfHH}(kv) \triangleq \mathtt{acyclic}\left(\left(\xrightarrow{\text{ddep}(kv)}\right)^+\right) \wedge \forall k, t, i, j.\ \begin{pmatrix}\mathsf{Write}(kv(k,i)) = \mathsf{Write}(kv(k,j)) \vee \\ (t \in \mathsf{Reads}(kv(k,i)) \wedge t \in \mathsf{Reads}(kv(k,j))\end{pmatrix} \implies i = j$$

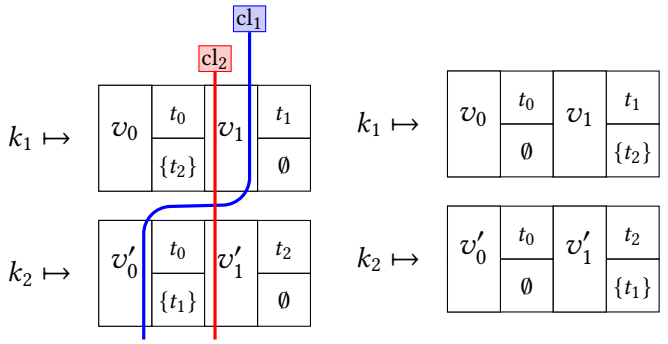
The *partial commutative monoid (pcm) of MKVSs* is $(\text{MKVSs}, \bullet_{kv}, \{\mathbf{1}_{kv}\})$, where $\bullet_{kv} : \text{MKVSs} \times \text{MKVSs} \rightharpoonup \text{MKVSs}$ denotes the *pcm composition* defined as the standard function disjointed union: $\bullet_{kv} \triangleq \uplus$; and $\mathbf{1}_{kv} \in \text{MKVSs}$ denotes the *pcm unit element*: $\mathbf{1}_{kv} \triangleq \emptyset$, where $\emptyset$ denotes a function with an empty domain.

AC: A point that this does not ensure a real causal dependency between the two versions, yet it is consistent with the notion of causality employed in databases, should be made

.

SX: Snapshot has not be explained yet

Let us elaborate on the first well-formedness constraint of an MKVS kv in Theorem 2.1. As stated above, this states that there is no circularity in the dependency relation. This in turn ensures that no versions are created *out of thin-air*. An example of the out of thin-air anomaly is given in Fig. 1b, where transaction t_2 reads the value of k_1 written by t_1; conversely, transaction t_1 reads the



(a) A configuration with a well-formed MSKV and two views

(b) An ill-formed MKVS

Fig. 1. Multi-version key-value stores

value of k_2 written by t_2. As we assume transactions read a state of the key-value store from an atomic snapshot fixed at the moment they execute, this situation cannot arise. For t_2 to read the version written by t_1, transaction t_2 must start after t_1, i.e. $kv_1(k_2, 2) \xrightarrow{\text{ddep}(kv_1)} kv(k_1, 2)$. Similarly, t_1 must starts after t_2, i.e. $kv_1(k_1, 2) \xrightarrow{\text{ddep}(kv_1)} kv_1(k_2, 2)$. This however violates the well-formedness of MKVSs that $\xrightarrow{\text{ddep}(kv_1)}$ is acyclic.

AR: Why kv_1 for the store and not kv? Also I thought the versions are indexed from 0 in which case index 2 does not make sense here?

AR: common Latin abbreviations such as i.e., e.g., and et al. do not need to be italicised. I have adjusted the macros. I have also rephrased the definitions quite a bit. Make sure you're happy with this.

SX: configuration is bad term? as we often use a configuration of the semantics

An MKVS tracks the the overall state of the system; however, different *clients* may observe different versions of the same key. To model this, we introduce the notion of *views* and *configurations* (Theorem 2.2). A view V reflects the particular version for each key that a client observes upon executing a transaction. We present an example of views in Fig. 1a with two views: cl_1 in red and cl_2 in blue. More concretely, the view for cl_1 is given formally as $u_1 = \{\text{k}_1 \mapsto 1, \text{k}_2 \mapsto 0\}$. That is, the client with view u_1 observes the second version (at index 1) of key k_1 with value v_1, and the first version (at index 0) of key k_2 with value v'_0.

Definition 2.2 (Views). *A view* is a partial finite function from keys to indexes: $u \in \textsc{Views} \triangleq \textsc{Keys} \xrightarrow{fin} \mathbb{N}$. The *view composition*, $\bullet_u : \textsc{Views} \times \textsc{Views} \rightharpoonup \textsc{Views}$ is defined as the standard disjoint function union: $\bullet_u \triangleq \uplus$. The *unit view*, $\mathbf{1}_u \in \textsc{Views}$, is a function with an empty domain: $\mathbf{1}_u \triangleq \emptyset$.

The *order relation* on views, $\leq$: $\textsc{Views} \times \textsc{Views}$, is defined between two views with the same domain as the point-wise comparison of their indexes for each entry:

$$u \leq u' \overset{\text{def}}{\iff} \text{dom}(u) = \text{dom}(u') \wedge \forall k.\, u(k) \leq u'(k)$$

A configuration comprises an MKVS, and the views associated with clients. In Fig. 1a we present an example of a configuration comprising an MKVS and the two views associated with clients cl_1 and cl_2. For brevity, we write $\mathsf{versionOf}(kv, k, u)$ for $kv(k, u(k))$; and write $\mathsf{Value}(kv, k, u)$ as a shorthand for $\mathsf{Value}(\mathsf{versionOf}(kv, \mathrm{k}, V))$; similarly for Write, Reads. When $\vartheta = \mathsf{versionOf}(kv, k, u)$, we say that u *k-points to* ϑ *in* kv. When $\vartheta = kv(k, i)$ for some $0 \leq i \leq u(k)$, we say that u *k-includes* ϑ *in* kv. Lastly, we always assume that MKVSs, views, and configurations are well-formed, unless otherwise stated.

Definition 2.3 (Configurations). A view u is *well-formed with respect to an MKVS* kv, written ????, iff they have the same domain and every index from u is within the range of the corresponding entry in kv:

$$???? \stackrel{\text{def}}{\iff} \mathrm{dom}(kv) = \mathrm{dom}(u) \wedge \forall k \in \mathrm{dom}(u).\ 0 \leq u(k) \leq |kv(k)|$$

AR: We need a symbol for this to fill the ???? above. Also ???? below.

A *configuration* C is a pair of the form $(kv, \mathcal{U})$, where kv denotes an MKVS, and $\mathcal{U} : \textsc{ClientID} \stackrel{fin}{\rightharpoonup} \textsc{Views}$ is a partial finite function from clients to views. A configuration $C = (kv, \mathcal{U})$ is *well-formed*, written ????, iff for all clients $\mathrm{cl} \in \mathrm{dom}(\mathcal{U})$, the view $\mathcal{U}(t)$ is well-formed with respect to kv.

When a client executes a transaction, it extracts a *snapshot* via the snapshot (kv, u) function, extracting the values corresponding to the versions indexed by the view u (Theorem 2.4). For instance, for client cl_1 in Fig. 1a, the snapshot $(\cdots)$ functions yields a state where key k_1 carries value v_1 and second key k_2 carries value v_0'.

AR: Before in MKVSs we had values drawn from $\mathbb{N}$ in Theorem 2.1. Now we use Val. I think you mean to use Val in both places?

Definition 2.4 (Snapshots). Given the sets of values Val and keys Keys (Theorem 2.1), the set of *snapshots* is: $ss \in \textsc{Snapshot} \triangleq \textsc{Keys} \stackrel{fin}{\rightharpoonup} \textsc{Val}$. The *snapshot composition function*, $\bullet_{ss} : \textsc{Snapshot} \times \textsc{Snapshot} \rightharpoonup \textsc{Snapshot}$, is defined as $\bullet_{ss} \triangleq \uplus$, where $\uplus$ denotes the standard disjoint function union. The *snapshot unit element* is $\mathbf{1}_{ss} \triangleq \emptyset$, denoting a function with an empty domain. The *partial commutative monoid of snapshots* is $(\textsc{Snapshot}, \bullet_{ss}, \{\mathbf{1}_{ss}\})$. Given an MKVS kv and a view u, the snapshot of u in kv, written $\mathsf{snapshot}(kv, u)$, is defined as: $\mathsf{snapshot}(kv, u) \triangleq \lambda k.\, \mathsf{Value}(kv, k, u)$.

AC: General Comment on this Section: it is too abstract. We should give either here or in the introduction an example of computation - the write skew program should be okay that helps the reader understanding what's going on. Also, it could be also good to illustrate the notions of execution tests and consistency models.

2.2 Programming Language and Operational Semantics

We define a simple programming language for client programs interacting with an MKVS where clients may only interact with the MKVS via transactions. For simplicity, we abstract away from aborting transactions: rather than assuming that a transaction may abort due to a violation of the consistency guarantees given by the MKVS, we only allow the execution of a transaction when its effects are guaranteed to not violate the consistency model. This emulates the setting in which clients always restart a transaction if it aborts.

SX: The sentence is too long. I think before the ":" is enough

2.2.1 Programming Language. A program $\mathtt{P}$ contains a fixed number of clients, where each client is associated with a unique identifier $\mathrm{cl} \in \textsc{ClientID}$, and executes a sequential *command*. We thus model a program $\mathtt{P}$ as a function from client identifiers to commands (Theorem 2.5). For clarity, we often write $\mathtt{C}_1 \parallel \cdots \parallel \mathtt{C}_n$ as a syntactic sugar for a program $\mathtt{P}$ with n implicit clients associated with identifiers, $\mathrm{cl}_1 \ldots \mathrm{cl}_n$, where each client cl_i executes the command $\mathtt{C}_i$: $\mathtt{P} = \{\mathrm{cl}_1 \mapsto \mathtt{C}_1, \ldots, \mathrm{cl}_n \mapsto \mathtt{C}_n\}$. Sequential *commands*, ranged over by $\mathtt{C}$, are defined by an inductive grammar comprising the standard constructs of `skip`, sequential composition ($\mathtt{C};\mathtt{C}$), non-deterministic choice ($\mathtt{C} + \mathtt{C}$) and loops ($\mathtt{C}^*$). To simulate conditional branching and loops, commands include the standard constructs of assume ($\mathtt{assume(E)}$), and assignment ($\mathtt{x} := \mathtt{E}$), where $\mathtt{x}$ denotes a local variable (on the stack), and $\mathtt{E}$ denotes an arithmetic expression with no side effect. evaluated with respect to a stack with no side effect. Arithmetic expressions are evaluated with respect to a local stack (variable store) – see Theorems 2.6 and 2.6 below. We assume a countably infinite set of local variables, $\textsc{Vars}$, and use the `typewriter` font for its meta variables, e.g. $\mathtt{x}$.

Commands additionally include the *transaction* construct, $\left[\mathtt{T}\right]$, denoting the *atomic* execution of the transaction $\mathtt{T}$. The atomicity guarantees the execution are dictated by the underlying consistency model. *Transactions*, ranged over by $\mathtt{T}$, are similarly defined by an inductive grammar comprising `skip`, *primitive commands* $\mathtt{T}_p$, non-deterministic choice, loops and sequential composition. Primitive commands include assignment ($\mathtt{x} := \mathtt{E}$), lookup ($\mathtt{E} := [\mathtt{E}]$), mutation ($[\mathtt{E}] := \mathtt{E}$) and assume ($\mathtt{assume(E)}$). Note that transactions do *not* contain the *parallel* composition construct ($\parallel$) as they are executed atomically.

Definition 2.5 (Programming language). A *program*, $\mathtt{P} \in \textsc{Prog}$, is a partial finite function from client identifiers to commands. The sequential *commands*, $\mathtt{C} \in \textsc{Cmd}$, are defined by the following grammar, where $v \in \textsc{Val} \triangleq \mathbb{N} \cup \textsc{Keys}$ denotes the set of *program values*, and $k \in \textsc{Keys}$ denotes the set of MKVS keys (Theorem 2.1):

$$\mathtt{C} ::= \mathtt{skip} \mid \mathtt{x} := \mathtt{E} \mid \mathtt{assume(E)} \mid \left[\mathtt{T}\right] \mid \mathtt{C};\mathtt{C} \mid \mathtt{C} + \mathtt{C} \mid \mathtt{C}^* \qquad \mathtt{T} ::= \mathtt{skip} \mid \mathtt{T}_p \mid \mathtt{T};\mathtt{T} \mid \mathtt{T} + \mathtt{T} \mid \mathtt{T}^*$$
$$\mathtt{E} ::= v \mid \mathtt{x} \mid \mathtt{E} + \mathtt{E} \mid \mathtt{E} \times \mathtt{E} \mid \ldots \qquad \mathtt{T}_p ::= \mathtt{a} := \mathtt{E} \mid \mathtt{a} := [\mathtt{E}] \mid [\mathtt{E}] := \mathtt{E} \mid \mathtt{assume(E)}$$

Definition 2.6 (Stacks). A *stack*, $s \in \textsc{Stacks}$, is a partial finite function from variables to values: $\textsc{Stacks} \triangleq \textsc{Vars} \overset{fin}{\rightharpoonup} \textsc{Val}$. Given a stack $s \in \textsc{Stacks}$, the *arithmetic expression evaluation* function, $[\![.]\!]_{(.)} : \textsc{Expr} \times \textsc{Stacks} \rightharpoonup \textsc{Val}$, is defined inductively over the structure of expressions:

$$[\![v]\!]_s \triangleq v \qquad [\![\mathtt{x}]\!]_s \triangleq s(\mathtt{x}) \qquad [\![\mathtt{E}_1 + \mathtt{E}_2]\!]_s \triangleq [\![\mathtt{E}_1]\!]_s + [\![\mathtt{E}_2]\!]_s \qquad [\![\mathtt{E}_1 \times \mathtt{E}_2]\!]_s \triangleq [\![\mathtt{E}_1]\!]_s \times [\![\mathtt{E}_2]\!]_s \qquad \ldots$$

2.2.2 Semantics for transactions. When a transaction starts, it determines a local snapshot from the current state of the system and a view, which we will explain the process in § 2.2.3. The transaction also installed with *fingerprints* which is initially empty. The fingerprints of a transaction corresponds to the set of all its interaction with the key-value store. Each time the transaction executes a primitive command $\mathtt{T}_p$ internal, the fingerprints get updated to trace the effect of the command. At the end, the transaction throw away the local snapshot, but commit the fingerprints to the system.

The fingerprints include the *first read preceding a write* and *last write* for each key. This is because a transaction is executed atomically, all the intermediate steps are not observable from the outside world. The *fingerprints* formally is a set of *operations* $\textsc{Ops}$ which are either read $(\mathtt{R}, k, v)$ from the k with the value v, or write $(\mathtt{W}, k, v)$ to the key k with the value v (Theorem 2.7). Note that In the Theorem 2.7, the $(.)\lfloor_{(.)}$ denotes projection. For a tuple, for example $o\lfloor_i$, it gives the *i-th* element of the tuple. It is lifted to a set of tuples, for example $\mathcal{F}\lfloor_i$, which gives a set of all the *i-th* elements. The well-formedness condition for fingerprints asserts it is a set of operations in which there are at

most one read and one write for each key. The composition, then, is defined as set disjointed union as long as the result is well-formed.

Definition 2.7 (operation and fingerprints). A *transaction operation* $o \in \textsc{Ops}$ is a tuple of *an operation tag* that is either read or write, an key and a value.

$$o \in \textsc{Ops} \triangleq \mathcal{P}\left(\{\mathtt{R}, \mathtt{W}\}\right) \times \textsc{Keys} \times \textsc{Val}$$

A well-formed fingerprint, $\mathcal{F} \in \textsc{Fp}$, is a subset of $\textsc{Ops}$ in which any two elements contain either different tags or different key.

$$\textsc{Fp} \triangleq \left\{\mathcal{F} \;\middle|\; \mathcal{F} \subseteq \textsc{Ops} \wedge \forall o, o' \in \mathcal{F}.\, o\lfloor_1 \neq o'\lfloor_1 \vee o\lfloor_2 \neq o'\lfloor_2\right\}$$

The unit element is $\mathbf{1}_{\mathcal{F}} \triangleq \emptyset$ and the composition of two fingerprints is the disjointed union when the two sets contain disjointed keyes,

$$\mathcal{F} \bullet_{\mathcal{F}} \mathcal{F}' \triangleq \begin{cases} \mathcal{F} \uplus \mathcal{F}' & \text{if } \mathcal{F}\lfloor_2 \cap \mathcal{F}'\lfloor_2 = \emptyset \\ \text{undefined} & \text{otherwise} \end{cases}$$

PG: $\mathcal{F} \lll \mathcal{F}'$?.

SX: We can define this but not sure it is useful

Lemma 2.8. *The well-formedness of fingerprints is closed under* $\lll$.

The operational semantics for transactions $\mathtt{T}$ (Fig. 2) is defined with respect to a transactional state of the form $(s, ss, \mathcal{F})$ comprising a stack, a snapshot and *fingerprints*. We first define a state transformers on pairs of stacks and snapshots for the primitive commands $\mathtt{T}_p$. We also define its fingerprint by fp function, which denotes the contribution of the primitive command that might observed by the external environment, i.e. transactions from other threads. The fp extracts the read or write operation from loop-up and mutation respectively, otherwise ϵ. We also define a binary operator $\mathcal{F} \lll o$ that specifies the effects of adding a new operation o to the fingerprints $\mathcal{F}$. If the new operation is a read, for example $(\mathtt{R}, k, v)$ where k is the key and v is the associated value, and there is no other operation related to the same key, this new read operation will be included in the result. Meanwhile, if the new operation is a write, it will overwrite all preview write operations to the same key. This ensures the fingerprints contains only the first read preceding a write, and only the last write for each key. This choice is motivated by the fact that we only focus on atomically visible transactions: keys are read from a snapshot of the database, and new version are written only at the moment the transaction commits. For technical reasons, if the right hand side is a special token ϵ corresponding to a command does not result in an interaction with key-value store. Therefore, the semantics for primitive command TPrimitive updates the stack and heap by the transformers relation and updates the operation set by first extracting the operation and adding it via $\lll$ operator. The semantics for non-deterministic choices TChoice, sequential compositions TSeqSkip and TSeq, and iteration TIter have the expected behaviours.

2.2.3 Program Semantics. The semantics for commands is in Fig. 3, taking the form:

$$(kv, s, u), \mathtt{C} \xrightarrow{\mathsf{ET}} (kv', s', u'), \mathtt{C}'$$

where kv, kv' are multi-version key-value stores (Theorem 2.1), s, s' are the stacks (Theorem 2.6), u, u' are views (Theorem 2.2), and ET is *an execution test* (Theorem 2.9), i.e. a condition that must be satisfied in order for a client to execute a transaction safely. We parametrise the execution test in the operational semantics of commands (Fig. 3), while we will give examples of different execution tests in **??**.

The state transformers on pairs of stacks and snapshots for the primitive commands T_p (left), and the op for extracting the operation from the primitive commands (right):

$$(s, ss) \xrightarrow{\mathsf{x:=E}} (s\left[\mathsf{x} \mapsto [\![\mathsf{E}]\!]_s\right], ss)$$
$$(s, ss) \xrightarrow{\mathsf{x:=[E]}} (s\left[\mathsf{x} \mapsto ss([\![\mathsf{E}]\!]_s)\right], ss)$$
$$(s, ss) \xrightarrow{\mathsf{[E_1]:=E_2}} (s, ss\left[[\![\mathsf{E}_1]\!]_s \mapsto [\![\mathsf{E}_2]\!]_s\right])$$
$$(s, ss) \xrightarrow{\mathsf{assume(E)}} (s, ss) \text{ where } [\![\mathsf{E}]\!]_s \neq 0$$

$$\begin{aligned}
\mathsf{fp}\,(s, ss, \mathsf{x := E}) &\triangleq \epsilon \\
\mathsf{fp}\,(s, ss, \mathsf{x := [E]}) &\triangleq (\mathsf{R}, [\![\mathsf{E}]\!]_s, ss([\![\mathsf{E}]\!]_s)) \\
\mathsf{fp}\,(s, ss, \mathsf{[E_1] := E_2}) &\triangleq (\mathsf{W}, [\![\mathsf{E}_1]\!]_s, [\![\mathsf{E}_2]\!]_s) \\
\mathsf{fp}\,(s, ss, \mathsf{assume(E)}) &\triangleq \epsilon \\
\mathsf{fp}\,(s, ss, \mathsf{return(E)}) &\triangleq \epsilon
\end{aligned}$$

The binary operator $\mathcal{F} \lll o$ that specifies the effects of adding a new operation o to the set $\mathcal{F}$:

$$\mathcal{F} \lll (\mathsf{R}, k, v) \triangleq \begin{cases} \mathcal{F} \uplus \{(\mathsf{R}, k, v)\} & (-, k, -) \notin \mathcal{F} \\ \mathcal{F} & \text{otherwise} \end{cases} \qquad \mathcal{F} \lll (\mathsf{W}, k, v) \triangleq (\mathcal{F} \setminus \{(\mathsf{W}, k, -)\}) \uplus \{(\mathsf{W}, k, v)\}$$
$$\mathcal{F} \lll \epsilon \triangleq \mathcal{F}$$

Given the set of stacks Stacks (Theorem 2.6), heaps Snapshot (Theorem 2.4) and transactions Trans (Theorem 2.5) and the arithmetic expression evaluation $[\![\mathsf{E}]\!]_s$ (Theorem 2.5), the *operational semantics of transactions*:

$$\rightsquigarrow : ((\textsc{Stacks} \times \textsc{Snapshot} \times \textsc{Fp}) \times \textsc{Trans}) \times ((\textsc{Stacks} \times \textsc{Snapshot} \times \textsc{Fp}) \times \textsc{Trans})$$

TPrimitive
$$\frac{(s, ss) \xrightarrow{\mathsf{T}_p} (s', ss') \qquad o = \mathsf{fp}\,(s, ss, \mathsf{T}_p)}{(s, ss, \mathcal{F}), \mathsf{T}_p \rightsquigarrow (s', ss', \mathcal{F} \lll o), \mathsf{skip}\rangle}$$

TChoice
$$\frac{i \in \{1, 2\}}{(s, ss, \mathcal{F}), \mathsf{T}_1 + \mathsf{T}_2 \rightsquigarrow (s, ss, \mathcal{F}), \mathsf{T}_i}$$

TIter
$$\frac{}{(s, ss, \mathcal{F}), \mathsf{T}^* \rightsquigarrow (s, ss, \mathcal{F}), \mathsf{skip} + (\mathsf{T}\,;\mathsf{T}^*)}$$

TSeqSkip
$$\frac{}{(s, ss, \mathcal{F}), \mathsf{skip}\,;\mathsf{T} \rightsquigarrow (s, ss, \mathcal{F}), \mathsf{T}}$$

TSeq
$$\frac{(s, ss, \mathcal{F}), \mathsf{T}_1 \rightsquigarrow (s', ss', \mathcal{F}'), \mathsf{T}_1'}{(s, ss, \mathcal{F}), \mathsf{T}_1\,;\mathsf{T}_2 \rightsquigarrow (s', ss', \mathcal{F}'), \mathsf{T}_1'\,;\mathsf{T}_2}$$

Fig. 2. Operational semantics for transactions

Execution tests is a set of quadruples $(kv, u, \mathcal{F}, u')$ consisting of a key-value store, a view before execution, a operation set and a view after committing of the operation set. We often write $(kv, u) \triangleright_{\mathsf{ET}} \mathcal{F} : u'$ in lieu of $(kv, u, \mathcal{F}, u') \in \mathsf{ET}$. The quadruple describes that when the state of the key-value store is kv, a client who has view u is allowed to execute a single transaction that has the fingerprints $\mathcal{F}$, and then after the commit the thread view must be updated to at least u'.

AC: There we also note that by tweaking the execution test used by the semantics, we capture different consistency models of key-value stores.

Definition 2.9 (Execution Tests). Given the set of key-value stores $kv \in$ MKVSs (Theorem 2.1), fingerprints $\mathcal{F} \in$ Fp (Theorem 2.7) and views $u, u' \in$ Views (Theorem 2.2), *execution tests* $\mathsf{ET} \in$ ET is a set of quadruples in the form of $(kv, u, \mathcal{F}, u')$:

$$\mathsf{ET} \in \mathrm{ET} \triangleq \mathcal{P}\,(\text{MKVSs} \times \textsc{Views} \times \textsc{Fp} \times \textsc{Views})$$

Well-formed execution tests ET require the domain of views and the key-value store are the same, and the fingerprints only have keys included in the previous domain:

$$\forall kv, u, u', \mathcal{F}.\ (kv, u, \mathcal{F}, u') \in \mathsf{ET} \implies \mathcal{F}|_2 \subseteq \mathrm{dom}(u) = \mathrm{dom}(u') = \mathrm{dom}(kv)$$

SX: Maybe also some version of composition requirement. For example, the composition of two should be also included in the consistency model.

$$\forall m, m'.\ m \in \mathsf{ET} \wedge m' \in \mathsf{ET} \implies m \bullet m' \in \mathsf{ET}$$

where $\bullet \triangleq (\bullet_{kv}, \bullet_u, \bullet_{\mathcal{F}}, \bullet_u)$.

The main rule in the operational semantics for commands is the one corresponding to the execution of a transaction PCommit. First, the view can shift to later versions before executing the transaction to model the client might gain more information about the key-value store since its last commit. To recall The order between two views with the same domain, for example $u'' \geq u$ in PCommit, is defined by the order of the indexes (Theorem 2.2). This new local view u'' should also be consistent with the key-value stores, i.e. it leads to a situation where the current client is allowed to execute the transaction. The transaction code T is executed locally given an initial snapshot $ss = \mathrm{snapshot}(kv, u'')$ (Theorem 2.4) decided by the current state of key-value store kv and the local view u''. After local execution via the semantics for transactions (Fig. 2), we propagate the stack s' and more importantly obtain the fingerprints $\mathcal{F}$, while the snapshot ss' will be throw away. Then the transaction picks a fresh identifier t, i.e. one that does not appear in the key-value store, and commits the fingerprint $\mathcal{F}$, which will update the key-value store and local view. The updateMKVS function updates the history heap using the fingerprint, $kv' = \mathrm{updateMKVS}\,(kv, u'', t, \mathcal{F})$. For read operations, it includes the new identifier to the read set of the version the is pointed by the local view u''. For write operations of the form $(\mathtt{W}, k, n)$, it extends a new version written by the new transaction $(n, t, \emptyset)$ to the tail of $kv(k)$. For updating the view, we set a lower bound for the new local view by updateView function. Assuming the commands executed by clients are wrapped with in a single session, the lower bound of the view corresponds to the strong session guarantees introduced by [?]. This function shifts the view to the up-to-date version in the new key-value store if the version is installed by the current transaction. This guarantees strong program order, meaning the following transaction will at least read its own write. Finally, the actual new local view u' is any view greater than the lower bound $u' \geq \mathrm{updateView}\,(kv', u'', \mathcal{F})$. The overall execution satisfied the execution tests, i.e. $(kv, u'') \triangleright_{\mathsf{ET}} \mathcal{F} : u'$.

AC: The paragraph below should probably go when discussing the rules of the semantics: Note that the way in which MKVSs and views are updated ensure the following: • a client always reads its own preceding writes; • clients always read from an increasingly up-to-date state of the database; • the order in which clients update a key k is consistent with the order of the versions for such keys in the MKVS; • writes take place after reads on which they depend.

Lemma 2.10. *The function* UpdateMKVS *is well-defined over well-formed MKVS, fingerprints and views. Given a well-formed MKVS kv, a view u that is well-formed with kv, well-formed fingerprints $\mathcal{F}$, and a t that does not appear in kv, then* $\mathrm{UpdateMKVS}(kv, \mathcal{F}, t, u)$ *is uniquely determined and yields a well-formed MKVS.*

Lemma 2.11. *The function* UpdateView *is well-defined. Given a well-formed MKVS kv, a view u that is well-formed with kv, well-formed fingerprints $\mathcal{F}$, then* $\mathrm{UpdateView}(kv, \mathcal{F}, u)$ *is uniquely determined a view and the view is well-formed with the MKVS.*

Last, the program has standard interleaving semantics by picking a client and then progressing one step (Fig. 3). To achieve that a thread environment holds the stacks and views associated with all active clients $Env \in \textsc{ThdEnv}$. We assume the client identifiers from client environment match with those in the program $\mathtt{P}$. We also assume all the stacks and views initially are the same respectively.

2.2.4 Example of Operational Semantics. To conclude our discussion on the operational semantics, we show in detail one possible computation of a program $\mathtt{P}_1$ consisting of two transactions executing in parallel:

$$\mathtt{P}_1 \equiv t_1 : \begin{bmatrix} \mathtt{[x]} := 1; \\ \mathtt{[y]} := 1; \end{bmatrix} \Bigg\| \; t_2 : \begin{bmatrix} \mathtt{a} := \mathtt{[x]}; \\ \mathtt{b} := \mathtt{[y]}; \\ \mathtt{if}\,(\mathtt{a} = 1 \wedge \mathtt{b} = 0)\{ \\ \quad \mathtt{ret} := \text{☹}\} \end{bmatrix}$$

The $\mathtt{if}\,(\mathtt{E})\{\mathtt{C}_1\}\mathtt{else}\{\mathtt{C}_2\}$ is encoded as $(\mathtt{assume}(\mathtt{E})\,;\,\mathtt{C}_1) + (\neg\mathtt{assume}(\mathtt{E})\,;\,\mathtt{C}_2)$. To recall, we often write $\mathtt{C}_1 \,\|\, \mathtt{C}_2 \,\|\, \cdots \,\|\, \mathtt{C}_n$ as a syntactic sugar for a program $\mathtt{P}$ with implicit unique thread identifiers $\mathtt{P} = \{\mathrm{cl}_1 \mapsto \mathtt{C}_1, \mathrm{cl}_2 \mapsto \mathtt{C}_2, \ldots, \mathrm{cl}_n \mapsto \mathtt{C}_n\}$. For better presentation, we annotated transactions with unique identifiers, yet they are allocated dynamically in the semantics. We also treat the value assigned to the `ret` variable as *returned value*. Assume the variables `x` and `y` refer to two key, and `a` and `b` are local variables to threads.

The special symbol ☹, for example the returned value by the transaction t_2, is to emphasise some undesirable behaviour of a transaction. In this case, the undesirable behaviour corresponds to the transaction to the right t_2 observing only one of the updates from t_1. Intuitively, this behaviour violates the constraints that transactions should be executed atomically (further discussed in **??**), we want to show that if no restrictions are placed on the consistency model specification, it is possible for $\mathtt{P}_1$ to reach a configuration where the second transaction t_2 returns ☹. To illustrate this and also explain the semantics, we instantiate the operation semantics with the most permissive execution tests $\triangleright_{\top}$, i.e. the view after u' at least observes its own writes and no other constraint:

$$(kv, u) \triangleright_{\top} \mathcal{F} : u' \triangleq \mathsf{updView}\,(kv, u, \mathcal{F}) \leq u'$$

AC: The condition on V' is not really needed.

Before any computation, the initial configuration for $\mathtt{P}_1$ is the one in which there are two keys k_{x} and k_{y} where each key is associated with an initial version with value zero written by an initialisation transaction t_0, $kv_0 = \{k_{\mathsf{x}} \mapsto [(0, t, \emptyset)], k_{\mathsf{y}} \mapsto [(0, t, \emptyset)]\}$. The initial view of each thread points to the initial version of each key, $u_0^1 = u_0^2 = \{k_{\mathsf{x}} \mapsto 1, k_{\mathsf{y}} \mapsto 1\}$. The two threads have the same initial stack containing two variables `x` and `y` referring to the only keys in the key-value store respectively, i.e. $s_0^1 = s_0^2 = \{\mathsf{x} \mapsto k_{\mathsf{x}}, \mathsf{y} \mapsto k_{\mathsf{y}}\}$. Therefore the initial configuration is $(kv_0, Env_0, \mathtt{P}_1)$ where $Env_0 = \{\mathrm{cl}_1 \mapsto (s_0^1, u_0^1), \mathrm{cl}_2 \mapsto (s_0^2, u_0^2)\}$. Fig. 4a gives a graphical representation of the initial configuration.

We are now ready to show how to derive a computation of $\mathtt{P}_1$ that violates *atomic visibility* and it will be explained formally in **??**. In the specific computation, we choose to let the transaction cl_1 commit first. According to rule PCommit, we need to perform the following steps:

SX: Not sure the bullet points work here? Just typesetting.

- Arbitrarily shift the view u_0^1 for thread cl_1 to the right as long as it is within the bound of key-value store and obtain a view u'' such that $u'' \geq u_0^1$. Because kv_0 contains only one version per key, here the only possibility is that $u'' = u_0^1$.
- Determine the initial snapshot $ss = \mathsf{snapshot}(kv_0, u'')$ for the transaction cl_1. In this case, we have that $ss = \{k_x \mapsto 0, k_y \mapsto 0\}$.

- Given the initial snapshot ss, initially empty fingerprint $\mathbf{1}_{\mathcal{F}}$ and the stack s_0^1, by the operational semantics for transaction (Fig. 2), after executing the transactional codes [x] := 1; [y] := 1, the finial fingerprints include two write operations as $\mathcal{F} = \{(\mathtt{W}, k_{\mathtt{x}}, 1), (\mathtt{W}, k_{\mathtt{y}}, 1)\}$.

AC:

SX: Not sure those details are necessary.

This amounts to execute the transaction in isolation from the external environment, using the rules in the operational semantics for transactions. Because this execution must match the premiss of Rule $(C - Tx)$, The code is run using h as the initial heap, σ_0 as the initial thread stack, $\tau_0 = \lambda_a.0$ as the initial transaction stack, and $\emptyset$ as the initial fingerprint. We only need to apply Rule $(Tx - prim)$ twice, in which case we obtain

$$\begin{array}{rcl} & \sigma_0 \vdash \left\langle h_0, \_, \emptyset, \begin{array}{l}[k_x] := 1; \\ [k_y] := 1\end{array} \right\rangle & \rightarrow \\ \rightarrow & \left\langle h_0[[k_x] \mapsto 1], \_, (\emptyset \oplus \mathrm{WR}\ [k_x] : 1), [k_y] := 1 \right\rangle & = \\ = & \left\langle h_0[[k_x] \mapsto 1], \_, \{\mathrm{WR}\ [k_x] : 1\}, [k_y] := 1 \right\rangle & \rightarrow \\ \rightarrow & \left\langle h_0[[k_x] \mapsto_1, [k_y] \mapsto 1], \_, (\{\mathrm{WR}\ [k_x] : 1\} \oplus (\mathrm{WR}\ [k_y] : 1)), - \right\rangle & = \\ = & \left\langle \_, \_, \{\mathrm{WR}\ [k_x] : 1, \mathrm{WR}\ [k_y] : 1\}, - \right\rangle & \end{array} \quad (1)$$

Therefore, we conclude $O = \{\mathrm{WR}\ [k_x] : 1, \mathrm{WR}\ [k_y] : 1\}$.

- The transaction throws away the local snapshot and commits the fingerprints to the key-value store. A fresh transaction identifier t_1 is picked. The new key-value store kv_1 is determined by the functions $kv_1 = \mathsf{updHisHp}\,(kv_0, u'', t_1, \mathcal{F})$ and the lower bound of the new view is given by the function $\mathsf{updView}\,(kv_1, u'', \mathcal{F})$.
- Last, the local view shift to the right so that it satisfies the execution tests, $u' \geq \mathsf{updView}\,(kv_1, u'', \mathcal{F}) \wedge (kv_0, u'') \triangleright_{\top} u' : \mathcal{F}$. In this case, the permissive model does not constraint the view at all. Therefore the overall final state is in Fig. 4b.

Next, thread cl_2 executes its own transaction. Note that the view for cl_2 still points to initial versions and the semantics allows the view get updated arbitrarily before executing the transaction. Because the key-value store now has two versions for each key, there are exactly four possible views for the transaction t_2. In particular, assume it updates the view for k_x but not k_y, i.e. $\{k_{\mathtt{x}} \mapsto 1, k_{\mathtt{x}} \mapsto 0\}$ (Fig. 4c). Given the view, the transaction t_2 will assign ☹ to the `ret` and this transaction is allowed to commit since the commit test does not stop this (Fig. 4d).

3 LOGIC

SX: small intro for why a logic

This example distinguishes serialisibility from snapshot isolation. To understand better, we associate unique capabilities for addresses, which indicates the latest values. Under serialisibility, a thread always observes be the latest values. While under snapshot isolation, it is sufficient to only see the latest values for addresses before the next transaction, if those addresses will be overwrite by the next transaction. After transaction a thread updates its view to the latest, yet it does not need to be always the latest while other threads interfere with the database which change the history heap.

Since the capabilities always remember the latest values, therefore in the interference I, if a transaction writes to an address, it also updates the capabilities to the value written, for example $[\mathtt{x}(\mathtt{N})]^{\mathrm{R}} \overset{\mathrm{U}}{\hookrightarrow} [\mathtt{x}(1)]^{\mathrm{R}}$. This capabilities also helps to understand the repartition especially under

snapshot isolation, in a way that after committing a transaction, the view should be updated to the latest for all addresses.

3.1 Local/Transaction

Definition 3.1 (Logical Expressions). Assume a countably infinite set of *logical variables* $\mathrm{x} \in \textsc{LVar}$. The set of *logical expressions*, $\mathrm{e} \in \textsc{LExpr}$ is defined by the following inductive grammar, where $v \in \textsc{Val}$ (Theorem 2.5) and $\mathrm{x} \in \textsc{Vars}$(Theorem 2.6),

$$\mathrm{e} ::= v \mid \mathrm{x} \mid \mathsf{x} \mid \mathrm{e} + \mathrm{e} \mid \mathrm{e} \times \mathrm{e} \mid \dots$$

Assume a set of *logical environments* $\iota \in \textsc{LEnv} : \textsc{LVar} \rightharpoonup \textsc{Val}$ which associates logical variables with values. Given a stack $s \in \textsc{Stacks}$ (Theorem 2.6) and a logical environment $\iota \in \textsc{LEnv}$, the *logical expression evaluation* function, $[\![.]\!]_{(.,.)} : \textsc{LExpr} \times \textsc{Stacks} \times \textsc{LEnv} \rightharpoonup \textsc{Val}$, is defined inductively over the structure of logical expressions as follows,

$$\begin{array}{c}
[\![v]\!]_{\iota,s} \triangleq v \\
[\![\mathrm{x}]\!]_{\iota,s} \triangleq \iota(\mathrm{x}) \\
[\![\mathsf{x}]\!]_{\iota,s} \triangleq \tau(\mathsf{x}) \\
[\![\mathrm{e}_1 + \mathrm{e}_2]\!]_{\iota,s} \triangleq [\![\mathrm{e}_1]\!]_{\iota,s} + [\![\mathrm{e}_2]\!]_{\iota,s} \\
[\![\mathrm{e}_1 \times \mathrm{e}_2]\!]_{\iota,s} \triangleq [\![\mathrm{e}_1]\!]_{\iota,s} \times [\![\mathrm{e}_2]\!]_{\iota,s} \\
\dots \triangleq \dots
\end{array}$$

Note that the stack s includes transaction variables and thread variables.

SX: The variable κ is a bit confused, since it is used in the model and syntax in assertion

Definition 3.2 (Capabilities). Assume a *partial commutative monoid (PCM)* of *client-specified capabilities* $(\textsc{Kap}, \bullet_\kappa, \mathbf{1}_\kappa)$ with $\kappa \in \textsc{Kap}$, the composition $\bullet_\kappa$ the units set $\mathbf{1}_\kappa$. Then given a set of *region identifiers* $r \in \textsc{RegionID}$, the *capabilities* $c \in \textsc{Cap} \triangleq \textsc{RegionID} \rightharpoonup \textsc{Kap}$, where the composition $\bullet_c$ is defined as the follows:

$$(c_l \bullet_c c_r)(r) \triangleq \begin{cases} c_l(r) \bullet_\kappa c_r(r) & r \in \mathrm{dom}(c_l) \cap \mathrm{dom}(c_l) \\ c_l(r) & r \in \mathrm{dom}(c_l) \setminus \mathrm{dom}(c_l) \\ c_r(r) & r \in \mathrm{dom}(c_r) \setminus \mathrm{dom}(c_l) \\ \text{undefined} & \text{otherwise} \end{cases}$$

and the units set $\mathbf{1}_c \triangleq \left\{ c \mid \forall r.\, c(r) \in \mathbf{1}_\kappa \right\}$. A capability assertion is in the form of $\left[\kappa(\vec{\mathrm{x}})\right]^{\mathrm{R}} \in \textsc{CAst}$, where $\kappa(\vec{\mathrm{x}})$ is a token parametrised by logical variables and R is the region identifiers. The capability assertion is interpreted to a capability in the model by interpreting all the logical expressions,

$$[\![\left[\kappa(\vec{\mathrm{x}})\right]^{\mathrm{R}}]\!]_{\iota,s} \triangleq \left\{ \mathrm{R} \mapsto \kappa([\![\vec{\mathrm{x}}]\!]_{\iota,s}) \right\}$$

The capability are used, first, as a token that grants some abilities to access to access the database if a thread holds the token. Second, it is ghost resources to help the reasoning.

A *fingerprint assertion* describes the possible global effect from a transaction. It includes the first read $\mathrm{e}_1 \overset{\mathsf{r}}{\hookrightarrow} \mathrm{e}_2$, last write $\mathrm{e}_1 \overset{\mathsf{w}}{\hookrightarrow} \mathrm{e}_2$ for the address e_1 and the combination of them $\mathrm{e}_1 \overset{\mathsf{rw}}{\hookrightarrow} \mathrm{e}_2$.

AR: I have changed it from $\overset{\perp}{\hookrightarrow}$ to $\hookrightarrow$, as $\perp$ has weird connotations. Default is no fingerprint so far and thus empty tag. I have also dropped the $\setminus$ from $\overset{\mathsf{a}}{\hookrightarrow}$ and $\overset{\mathsf{d}}{\hookrightarrow}$ as it looks a little strange.

The $\mathtt{E}_1 \hookrightarrow \mathtt{E}_2$ means the address $\mathtt{E}_1$ has no associated fingerprint and the initial value is $\mathtt{E}_2$. The $\mathtt{E}_1 \overset{\mathsf{r}}{\hookrightarrow} \mathtt{E}_2$ means the address has been read before any other write and the current value is $\mathtt{E}_2$. The $\mathtt{E}_1 \overset{\mathsf{w}}{\hookrightarrow} \mathtt{E}_2$ means the address has been written at least once, and the last written value is $\mathtt{E}_2$. Last, The $\mathtt{E}_1 \overset{\mathsf{rw}}{\hookrightarrow} (\mathtt{E}_2, \mathtt{E}_3)$ means the address has been read and then written, the first read value is $\mathtt{E}_2$ and the last written value is $\mathtt{E}_3$.

We will use fingerprint assertions to specify interference (shown later), therefore we also have special fingerprint for transferring of capabilities, i.e. adding to the shared state $\overset{\mathsf{a}}{\hookrightarrow} [\kappa]^{\mathsf{R}}$, deleting from the shared state $\overset{\mathsf{d}}{\hookrightarrow} [\kappa]^{\mathsf{R}}$ and updating the capabilities $\left[\kappa(\vec{\mathtt{x}})\right]^{\mathsf{R}} \overset{\mathsf{u}}{\hookrightarrow} \left[\kappa(\vec{\mathtt{x}})\right]^{\mathsf{R}}$. The shared state, intuitively, is the state aware by all threads and we will explain the details later.

SX: ALREADY: We might be able to eliminate all the heap assertions by combining them to fingerprints.

Definition 3.3 (Fingerprint Assertions). The *fingerprint assertion*, $f \in \textsc{FAst}$, is defined as the follows,

$$f, f' ::= \mathtt{E} \hookrightarrow \mathtt{E} \mid \mathtt{E} \overset{\mathsf{r}}{\hookrightarrow} \mathtt{E} \mid \mathtt{E} \overset{\mathsf{w}}{\hookrightarrow} \mathtt{E} \mid \mathtt{E} \overset{\mathsf{rw}}{\hookrightarrow} (\mathtt{E}, \mathtt{E}) \mid \overset{\mathsf{a}}{\hookrightarrow} \left[\kappa(\vec{\mathtt{x}})\right]^{\mathsf{R}} \mid \overset{\mathsf{d}}{\hookrightarrow} \left[\kappa(\vec{\mathtt{x}})\right]^{\mathsf{R}} \mid \left[\kappa(\vec{\mathtt{x}})\right]^{\mathsf{R}} \overset{\mathsf{u}}{\hookrightarrow} \left[\kappa(\vec{\mathtt{x}})\right]^{\mathsf{R}} \mid f * f'$$

Given a logical environment $\iota \in \textsc{LEnv}$ and a stack $s \in \textsc{Stacks}$, the *fingerprint* is interpreted through function, $[\![.]\!]_{(\iota,.)} : \textsc{FAst} \times \textsc{LEnv} \times \textsc{Stacks} \rightharpoonup \textsc{Snapshot} \times \textsc{Fp} \times \textsc{Cap} \times \textsc{Cap}$.

SX: Capabilities updates should be able to encode as add and remove.

AR:

SX: Keep the set but separate the capabilities from the set.

First, see my comment on page **??**.
The above definition is wrong; for instance, you can have $(\mathtt{A}, c \bullet_c c'), \in (\mathtt{A}, c) \in \widetilde{O}$, even though $c \bullet_c c' \neq c$.
Given my definition on **??**, we can then define $\widetilde{O} : (\textsc{Keys} \overset{fin}{\rightharpoonup} \textsc{Val} \times \mathcal{P}\left(\{\mathtt{R}, \mathtt{W}\}\right)) \times \textsc{Cap} \times \textsc{Cap}$, where the first part of the triple is as before, the second part tracks the capabilities added, the third part tracks the capabilities removed.
Again, you won't need the well-formedness.
You can define composition component-wise as $(\uplus, \bullet_c, \bullet_c)$.
You will need to adjust the interpretation below.

$$
\begin{aligned}
[\![\mathrm{E}_1 \hookrightarrow \mathrm{E}_2]\!]_{\iota,s} &\triangleq \left\{(ss, \mathbf{1}_{\mathcal{F}}, c, c') \;\middle|\; \begin{array}{l}\exists k, v.\, k = [\![\mathrm{E}_1]\!]_{\iota,s} \wedge v = [\![\mathrm{E}_2]\!]_{\iota,s} \\ \quad \wedge\, ss = \{k \mapsto v\} \wedge c, c' \in \mathbf{1}_c\end{array}\right\} \\
\left[\!\!\left[\mathrm{E}_1 \overset{\mathsf{r}}{\hookrightarrow} \mathrm{E}_2\right]\!\!\right]_{\iota,s} &\triangleq \left\{(ss, \mathcal{F}, c, c') \;\middle|\; \begin{array}{l}\exists k, v.\, k = [\![\mathrm{E}_1]\!]_{\iota,s} \wedge v = [\![\mathrm{E}_2]\!]_{\iota,s} \\ \quad \wedge\, ss = \{k \mapsto v\} \wedge \mathcal{F} = \{(\mathsf{R}, k, v)\} \wedge c, c' \in \mathbf{1}_c\end{array}\right\} \\
\left[\!\!\left[\mathrm{E}_1 \overset{\mathsf{w}}{\hookrightarrow} \mathrm{E}_2\right]\!\!\right]_{\iota,s} &\triangleq \left\{(ss, \mathcal{F}, c, c') \;\middle|\; \begin{array}{l}\exists k, v.\, k = [\![\mathrm{E}_1]\!]_{\iota,s} \wedge v = [\![\mathrm{E}_2]\!]_{\iota,s} \\ \quad \wedge\, ss = \{k \mapsto v\} \wedge \mathcal{F} = \{(\mathsf{W}, k, v)\} \wedge c, c' \in \mathbf{1}_c\end{array}\right\} \\
\left[\!\!\left[\mathrm{E}_1 \overset{\mathsf{rw}}{\hookrightarrow} (\mathrm{E}_2, \mathrm{E}_3)\right]\!\!\right]_{\iota,s} &\triangleq \left\{(ss, \mathcal{F}, c, c') \;\middle|\; \begin{array}{l}\exists k, v, v'.\, k = [\![\mathrm{E}_1]\!]_{\iota,s} \wedge v = [\![\mathrm{E}_2]\!]_{\iota,s} \wedge v' = [\![\mathrm{E}_3]\!]_{\iota,s} \\ \quad \wedge\, ss = \{k \mapsto v'\} \wedge \mathcal{F} = \{(\mathsf{R}, k, v), (\mathsf{W}, k, v')\} \wedge c, c' \in \mathbf{1}_c\end{array}\right\} \\
\left[\!\!\left[\overset{\mathsf{a}}{\hookrightarrow} \left[\kappa(\vec{\mathrm{x}})\right]^{\mathrm{R}}\right]\!\!\right]_{\iota,s} &\triangleq \left\{(\mathbf{1}_{ss}, \mathbf{1}_{\mathcal{F}}, c, c') \;\middle|\; c = \left[\!\!\left[\left[\kappa(\vec{\mathrm{x}})\right]^{\mathrm{R}}\right]\!\!\right]_{\iota,s} \wedge c' \in \mathbf{1}_c\right\} \\
\left[\!\!\left[\overset{\mathsf{d}}{\hookrightarrow} \left[\kappa(\vec{\mathrm{x}})\right]^{\mathrm{R}}\right]\!\!\right]_{\iota,s} &\triangleq \left\{(\mathbf{1}_{ss}, \mathbf{1}_{\mathcal{F}}, c, c') \;\middle|\; c \in \mathbf{1}_c \wedge c' = \left[\!\!\left[\left[\kappa(\vec{\mathrm{x}})\right]^{\mathrm{R}}\right]\!\!\right]_{\iota,s}\right\} \\
\left[\!\!\left[\left[\kappa(\vec{\mathrm{x}})\right]^{\mathrm{R}} \overset{\mathsf{u}}{\hookrightarrow} \left[\kappa'(\vec{\mathrm{x}}')\right]^{\mathrm{R}}\right]\!\!\right]_{\iota,s} &\triangleq \left\{(\mathbf{1}_{ss}, \mathbf{1}_{\mathcal{F}}, c, c') \;\middle|\; c = \left[\!\!\left[\left[\kappa(\vec{\mathrm{x}})\right]^{\mathrm{R}}\right]\!\!\right]_{\iota,s} \wedge c' = \left[\!\!\left[\left[\kappa'(\vec{\mathrm{x}}')\right]^{\mathrm{R}}\right]\!\!\right]_{\iota,s}\right\} \\
[\![f_1 * f_2]\!]_{\iota,s} &\triangleq \left\{(ss_1 \bullet_{ss} ss_2, \mathcal{F}_1 \bullet_{\mathcal{F}} \mathcal{F}_2, c_1 \bullet_c c_2, c_1' \bullet_c c_2') \;\middle|\; (ss_1, \mathcal{F}_1, c_1, c_1') \in [\![f_1]\!]_{\iota,s} \wedge (ss_2, \mathcal{F}_2, c_2, c_2')\right.
\end{aligned}
$$

The local assertions have true, false, conjunction $\wedge$, disjunction $\vee$, existential quantification $\exists$, implication $\implies$, empty emp, heap-related fingerprint assertions $\bar{f}$ and separation conjunction $*$. They describes the state of local heap used by a transaction and more importantly the fingerprint of the transaction. They are interpreted to pairs of heaps and operation sets.

Definition 3.4 (Local assertions). Given the set of logical expressions LExpr and logical variables LVar, the set of *local assertions*, $p, q \in \mathrm{LAst}$, is defined by lifting the *read-write fraction of fingerprint assertions* $\bar{f}$,

$$
\begin{aligned}
\bar{f} &::= \mathrm{E} \overset{\mathsf{r}}{\hookrightarrow} \mathrm{E} \mid \mathrm{E} \overset{\mathsf{w}}{\hookrightarrow} \mathrm{E} \mid \mathrm{E} \overset{\mathsf{rw}}{\hookrightarrow} (\mathrm{E}, \mathrm{E}) \mid \mathrm{E} \hookrightarrow \mathrm{E} \\
p, q &::= \mathsf{false} \mid \mathsf{true} \mid p \wedge q \mid p \vee q \mid \exists \mathrm{x}.\, p \mid p \implies q \mid \mathrm{emp} \mid \bar{f} \mid p * q
\end{aligned}
$$

Given a logical environment $\iota \in \mathrm{LEnv}$, the *local interpretation function*, $[\![.]\!]_{(.,.)} : \mathrm{LAst} \times \mathrm{LEnv} \times \mathrm{LAst} \rightharpoonup \mathcal{P}\,(\mathrm{Snapshot} \times \mathrm{Fp})$, is defined over the structure of local assertions as follows,

$$
\begin{aligned}
[\![\mathsf{false}]\!]_{\iota,s} &\triangleq \emptyset \\
[\![\mathsf{true}]\!]_{\iota,s} &\triangleq \mathrm{Snapshot} \times \mathrm{Fp} \\
[\![p \wedge q]\!]_{\iota,s} &\triangleq [\![p]\!]_{\iota,s} \cap [\![q]\!]_{\iota,s} \\
[\![p \vee q]\!]_{\iota,s} &\triangleq [\![p]\!]_{\iota,s} \cup [\![q]\!]_{\iota,s} \\
[\![\exists \mathrm{x}.\, p]\!]_{\iota,s} &\triangleq \bigcup_{v \in \mathrm{Val}} [\![p]\!]_{\iota[\mathrm{x} \mapsto v],s} \\
[\![p \implies q]\!]_{\iota,s} &\triangleq \left\{(ss, \mathcal{F}) \;\middle|\; (ss, \mathcal{F}) \in [\![p]\!]_{\iota,s} \implies (ss, \mathcal{F}) \in [\![q]\!]_{\iota,s}\right\} \\
[\![\mathrm{emp}]\!]_{\iota,s} &\triangleq \left\{(\mathbf{1}_{ss}, \mathbf{1}_{\mathcal{F}})\right\} \\
[\![\bar{f}]\!]_{\iota,s} &\triangleq \left\{(ss, \mathcal{F}) \;\middle|\; \exists ss, \mathcal{F}, c, c'.\, (ss, \mathcal{F}, c, c') \in \left[\!\!\left[\bar{f}\right]\!\!\right]_{\iota,s} \wedge c, c' \in \mathbf{1}_c\right\} \\
[\![p * q]\!]_{\iota,s} &\triangleq \left\{(ss_1 \bullet_{ss} ss_2, \mathcal{F}_1 \bullet_{\mathcal{F}} \mathcal{F}_2) \;\middle|\; (ss_1, \mathcal{F}_1) \in [\![p]\!]_{\iota,s} \wedge (ss_2, \mathcal{F}_2) \in [\![q]\!]_{\iota,s}\right\}
\end{aligned}
$$

AR:

SX: Leave this notation for now, when settle other parts, we can change the macro systematically for confused notations.

The definition of $\left[\!\!\left[\bar{f}\right]\!\!\right]_{\iota,s}$ looks cyclic as you use the same notation for interpreting local assertions and fingerprint assertions.

Observe that program expressions Expr (Theorem 2.5) are contained in logical expressions LExpr (Theorem 3.4 above), i.e. Expr ⊂ LExpr. Because the local assertions do not contain any assertions related to capabilities, it is enough to interpret them to heaps and sets of operations Fp.

Lemma 3.5. $\forall \bar{f}, c, c'.\ (-, -, c, c') \in [\![\bar{f}]\!]_{\iota,s} \implies c, c' \in \mathbf{1}_c$

Proof. Induction on the structures. □

3.2 Global/Program

The interference $\exists \vec{x}.\ [\kappa] : f$ says if a thread holds the capability $[\kappa]$, it is allowed to commit a transaction that has the fingerprint f. The existential is for binding variables between the capability and the fingerprint assertions.

SX: Parameter of the κ? The $\mathcal{I}$ might simplify to single view to single view.

Definition 3.6 (Interference). Given the fingerprint assertion $f \in$ FingerPrint (Theorem 3.3), the grammar of *interference assertions*, $I \in$ IAst, is defined as the follows,

$$I ::= \emptyset \mid \left\{\exists \vec{x}.\ [\kappa] : f\right\} \cup I$$

The interference assertions are interpreted to *interference environments* that is a set of transitions on history heap, a set of views, and capabilities,

$$\mathcal{I} \in \text{Inter} \triangleq \text{Kap} \rightarrow (\text{MKVSs} \times \text{Views} \times \text{Cap}) \times (\text{MKVSs} \times \text{Views} \times \text{Cap})$$

Given a logical environment $\iota \in$ LEnv and a stack $s \in$ Stacks, the *interference interpretation* function, $[\![.]\!]_{(.,.)} : \text{IAst} \times \text{LEnv} \times \text{Stacks} \rightarrow \text{Inter}$, is defined as follows,

SX: Notations are confused, there is different between syntactic κ which can be parametrised by logical variables, and client-specified capabilities κ'. Fix the typesetting later.

$$[\![\left\{\exists \vec{x}.\ [\kappa] : f\right\} \cup I]\!]_{\iota,s}(\kappa') \triangleq \begin{cases} \left\{ \begin{pmatrix} (kv, u, c_r \bullet_c c_f), \\ (kv', u', c_f \bullet_c c_a) \end{pmatrix} \;\middle|\; \begin{array}{l} \exists t, \mathcal{F}. \\ \quad (-, \mathcal{F}, c_a, c_r) \in [\![f]\!]_{\iota',s} \\ \quad \wedge\ t \in \mathsf{fresh}(kv) \\ \quad \wedge\ kv' = \mathsf{updHisHp}(kv, u, t, \mathcal{F}) \\ \quad \wedge\ u' \geq \mathsf{updView}(kv', u, \mathcal{F}) \end{array} \right\} \cup [\![I]\!]_{\iota,s}(\kappa') & \text{if } \dagger \\ [\![I]\!]_{\iota,s}(\kappa') & \text{other} \end{cases}$$

$$\dagger \equiv \exists \vec{v}, \iota'.\ \iota' = \iota[\vec{x} \mapsto \vec{v}] \wedge \kappa' = [\![\kappa]\!]_{\iota',s}$$

SX: Need some verbal explanation here for *worlds*. Later

Definition 3.7 (Worlds). Given the set of history heaps MKVSs (Theorem 2.1), views Views (Theorem 2.2), capabilities Cap (Theorem 3.2) and region identifiers RegionID, the set of *shared states* is SState $\triangleq$ RegionID $\rightharpoonup$ MKVSs $\times$ Views $\times$ Cap $\times$ Inter. Each region has its current state and the interference. The *shared state composition function*, $\bullet_g : \text{SState} \times \text{SState} \rightharpoonup \text{SState}$, is defined as $\bullet_g \triangleq \bullet_=$, where for all domains M and all $m, m' \in \text{M}$,

$$m \bullet_= m' \triangleq \begin{cases} m & \text{if } m = m' \\ \text{undefined} & \text{otherwise} \end{cases}$$

A *world* $w \in \textsc{World}$ is a pair of capabilities c (Theorem 3.2) and a shared state g in which regions are well-formed, i.e. (a) they are associated with disjointed part of history heaps; (b) the domain of the view in a region is the same as the domain of the history heap; and (c) the views should not be out of the range of history heaps. Separately, capabilities from regions and local capabilities are compatible. These constraints are derived by the clap $[\![(c, g)]\!] \neq \emptyset$. Finally, there is no garbage capability, a capability where the associated region identifier never appear in the shared state.

$$w \in \textsc{World} \triangleq \left\{ (c, g) \;\middle|\; \begin{array}{l} c \in \textsc{Cap} \land g \in \textsc{SState} \land \exists c'. \\ \quad (-, -, c') \in \mathsf{collapse}\,(g) \land \mathrm{dom}(c \bullet_c c') \subseteq \mathrm{dom}(g) \\ \quad \land\, \forall r.\, \exists kv, u.\, g(r) = (kv, u, -) \land \mathrm{dom}(kv) = \mathrm{dom}(u) \\ \quad \land\, \forall k \in \mathrm{dom}(u).\, 0 \leq u(k) \leq |kv(k)| \end{array} \right\}$$

where the collapse function collapses a shared state by erasing the region identifiers:

$$\begin{array}{c} \mathsf{collapse}\,(\emptyset) \triangleq \{(\mathbf{1}_{kv}, \mathbf{1}_u, \mathbf{1}_c)\} \\ \mathsf{collapse}\,(\{r \mapsto (kv, u, c, \mathcal{I})\} \uplus g) \triangleq \{(kv \bullet_{kv} kv', u \bullet_u u', c \bullet_c c') \mid \land (kv', u', c') \in \mathsf{collapse}\,(g)\} \end{array}$$

The *world composition function*, $\bullet_w : \textsc{World} \times \textsc{World} \rightharpoonup \textsc{World}$, is defined component-wise as: $\bullet_w \triangleq (\bullet_c, \bullet_g)$. The *world unit set* is $\mathbf{1}_w \triangleq \{(c, g) \mid (c, g) \in \textsc{World} \land c \in \mathbf{1}_c\}$. The *partial commutative monoid of worlds* is $(\textsc{World}, \bullet_w, \mathbf{1}_w)$. The function $[\![.]\!] : \textsc{World} \rightharpoonup \mathcal{P}\,(\textsc{MKVSs} \times \textsc{Views})$ collapse a world to a configuration:

$$[\![(c, g)]\!] \triangleq \{(kv, u) \mid (kv, u, c') \in \mathsf{collapse}\,(g)\}$$

SX: Explain here: box assertion, local part, shared part, intuitively how they get interpreted

Definition 3.8 (Assertions). Given the set of logical expression $\mathrm{E} \in \mathbb{E} \subseteq \textsc{LExpr}$, the set of *assertions*, $P, Q \in \textsc{Ast}$, are defined by the following inductive grammar:

$$\begin{array}{l} \bar{p}, \bar{q} ::= \mathsf{false} \mid \mathsf{true} \mid \bar{p} \land \bar{q} \mid \bar{p} \lor \bar{q} \mid \exists \mathrm{x}.\, \bar{p} \mid \bar{p} \implies \bar{q} \mid \mathsf{emp} \mid [\kappa]^{\mathrm{R}} \mid \mathrm{E} \mapsto \mathbb{E} \mid \bar{p} * \bar{q} \\ P, Q ::= \mathsf{false} \mid \mathsf{true} \mid P \land Q \mid P \lor Q \mid \exists \mathrm{x}.\, P \mid P \implies Q \mid \mathsf{emp} \mid [\kappa]^{\mathrm{R}} \mid P * Q \mid \boxed{\bar{p}}^{\mathrm{R}}_{I} \end{array}$$

where $\mathrm{x}, \mathrm{R} \in \textsc{LVar}$, $\mathrm{E}_1, \mathrm{E}_2 \in \textsc{LExpr}$ (Theorem 3.4), $\kappa \in \textsc{Kap}$ (Theorem 3.2) and $I \in \textsc{IAst}$ (Theorem 3.6). Given a logical environment $\iota \in \textsc{LEnv}$ and a stack $s \in \textsc{Stacks}$, the *assertion interpretation* function,

$[\![.]\!]_{(.,.)} : \textsc{Ast} \times \textsc{LEnv} \times \textsc{Stacks} \to \mathcal{P}(\textsc{World})$, is defined as follows,

$$
\begin{aligned}
[\![\text{false}]\!]_{\iota,s} &\triangleq \emptyset \\
[\![\text{true}]\!]_{\iota,s} &\triangleq \textsc{World} \\
[\![\text{emp}]\!]_{\iota,s} &\triangleq \mathbf{1}_w \\
[\![P \wedge Q]\!]_{\iota,s} &\triangleq [\![P]\!]_{\iota,s} \cap [\![Q]\!]_{\iota,s} \\
[\![P \vee Q]\!]_{\iota,s} &\triangleq [\![P]\!]_{\iota,s} \cup [\![Q]\!]_{\iota,s} \\
[\![\exists \mathrm{x}.\, P]\!]_{\iota,s} &\triangleq \bigcup_{v \in \textsc{Val}} [\![P]\!]_{\iota[\mathrm{x} \mapsto v],s} \\
[\![P \implies Q]\!]_{\iota,s} &\triangleq \left\{ w \;\middle|\; w \in [\![P]\!]_{\iota,s} \implies w \in [\![Q]\!]_{\iota,s} \right\} \\
[\![[\kappa]^{\mathrm{R}}]\!]_{\iota,s} &\triangleq \left\{ (\{\mathrm{R} \mapsto [\![\kappa]\!]_{\iota,s}\}, g) \;\middle|\; g \in \textsc{SState} \right\} \\
[\![P * Q]\!]_{\iota,s} &\triangleq \left\{ (w_1 \bullet_w w_2) \;\middle|\; w_1 \in [\![P]\!]_{\iota,s} \wedge w_2 \in [\![Q]\!]_{\iota,s} \right\} \\
[\![\boxed{\bar{p}}^{\mathrm{R}}_{I}]\!]_{\iota,s} &\triangleq \left\{ (c, g) \;\middle|\; \begin{array}{l} \exists kv, u, c', \mathcal{I}.\, c \in \mathbf{1}_c \\ \quad \wedge\, \mathcal{I} = [\![I]\!]_{\iota,s} \wedge g(\mathrm{R}) = (kv, u, c', \mathcal{I}) \wedge (kv, u, c') \in \mathsf{intp}\,(\bar{p}, \iota, s) \end{array} \right\}
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{intp}\,(\text{false}, \iota, s) &\triangleq \emptyset \\
\mathsf{intp}\,(\text{true}, \iota, s) &\triangleq \textsc{MKVSs} \times \textsc{Views} \times \textsc{Cap} \\
\mathsf{intp}\,(\text{emp}, \iota, s) &\triangleq \left\{ (\mathbf{1}_{kv}, \mathbf{1}_u, c) \;\middle|\; c \in \mathbf{1}_c \right\} \\
\mathsf{intp}\,(\bar{p} \wedge \bar{q}, \iota, s) &\triangleq \mathsf{intp}\,(\bar{p}, \iota, s) \cap \mathsf{intp}\,(\bar{q}, \iota, s) \\
\mathsf{intp}\,(\bar{p} \vee \bar{q}, \iota, s) &\triangleq \mathsf{intp}\,(\bar{p}, \iota, s) \cup \mathsf{intp}\,(\bar{q}, \iota, s) \\
\mathsf{intp}\,(\exists \mathrm{x}.\, \bar{p}, \iota, s) &\triangleq \bigcup_{v \in \textsc{Val}} \mathsf{intp}\,(\bar{p}, \iota[\mathrm{x} \mapsto v], s) \\
\mathsf{intp}\,(\bar{p} \implies \bar{q}, \iota, s) &\triangleq \left\{ (kv, u, c) \;\middle|\; (kv, u, c) \in \mathsf{intp}\,(\bar{p}, \iota, s) \implies (kv, u, c) \in \mathsf{intp}\,(\bar{q}, \iota, s) \right\} \\
\mathsf{intp}\,([\kappa]^{\mathrm{R}}, \iota, s) &\triangleq \left\{ (\mathbf{1}_{kv}, \mathbf{1}_u, \{\mathrm{R} \mapsto [\![\kappa]\!]_{\iota,s}\}) \right\} \\
\mathsf{intp}\,(\mathrm{E}_1 \mapsto \mathbb{E}, \iota, s) &\triangleq \left\{ (kv, u, c) \;\middle|\; \begin{array}{l} \forall u' \geq u.\, \exists \mathrm{E}_2 \in \mathbb{E}.\, c \in \mathbf{1}_c \wedge \\ \quad \{[\![\mathrm{E}_1]\!]_{\iota,s} \mapsto [\![\mathrm{E}_2]\!]_{\iota,s}\} = \mathsf{snapshot}(kv, u') \end{array} \right\} \\
\mathsf{intp}\,(\bar{p} * \bar{q}, \iota, s) &\triangleq \left\{ (kv \bullet_{kv} kv', u \bullet_u u', c \bullet_c c') \;\middle|\; (kv, u, c) \in \mathsf{intp}\,(\bar{p}, \iota, s) \wedge (kv', u', c') \in \mathsf{intp}\,(\bar{q}, \iota, s) \right\}
\end{aligned}
$$

3.3 Rules for Local

The proof rules (Fig. 6) are standard except TRLookup and TRMutate. The TRLookup rule adds read fingerprint only if there is no write fingerprint. This is because once an address has been re-written, the rest reads are local. TheTRMutate rule adds write fingerprint if there no write fingerprint and keeps the last written value.

3.3.1 Rely and Guarantee. The *rely* and *guarantee* describes the world transformation for the environment and for the current client (Theorem 3.9). To recall, a world includes local capabilities and a shared state, and a shared state is a client's view for the key-value store. The *rely* $\mathcal{R}$ is a set of pairs on worlds, describing how the environment can change the state of the key-value store. Given the local capabilities c_l, the environment might own any capabilities c_e that is compatible, i.e. $(c_l \bullet_c c_e)\downarrow$. Therefore, the environment can perform actions associated with the their capabilities c_e with their own view u_e to update the key-value store and shared capabilities. For technical reasons, even though the environment cannot change the view of the current client u_r, but it is allowed to arbitrarily shift to the later versions due to the fact that for certain execution tests, the old view might be valid under the new key-value store.

The *guarantee* $\mathcal{G}$ describes the allowed actions for the current client. The current client can perform actions associated with the local capabilities c_l to update the shared state and the local capabilities. Yet it should ensure no resource created or deleted by requiring the *orthogonal* of local capabilities and shared capabilities together remains unchanged. The orthogonal of capabilities c is

a set of capabilities that is compatible with the capabilities c. This constraint disallow any creation and deletion for capabilities, but it allows to update capabilities.

We allow a transaction to update several regions together, but each region can be updated at most once. Given that, the guarantee is a set of pairs of worlds that are allowed by the local capabilities. Each pairs assertions how a world can evolve. The rely is a set of pairs of worlds asserting how the history heaps are changed with respect to capabilities the current thread does not own. Note that the rely does not change the view, because this corresponds a thread from the environment that change the history heap with respect to its own view.

Definition 3.9 (Rely and guarantee). Given the set of worlds World (Theorem 3.7), the *rely* relation, $\mathcal{R} \subseteq \text{World} \times \text{World}$, is defined as follows,

SX: In case I get confused again, it is world to world so the shared and local capabilities should always make sense.

$$\mathcal{R} \triangleq \left\{ ((c_l, g), (c_l, g')) \;\middle|\; \begin{array}{l} \exists c_e.\, (c_e \bullet_c c_l) \downarrow \wedge \forall r.\, g(r) = g'(r) \vee \exists \kappa, kv, kv', u_r, u_r', u_e, u_e', c_r, c_r', \mathcal{I}. \\ \quad g(r) = (kv, u_r, c_r, \mathcal{I}) \wedge g'(r) = (kv', u_r', c_r', \mathcal{I}) \\ \quad \wedge\, \kappa \sqsubseteq c_e(r) \wedge ((kv, u_e, c_e), (kv', u_e', c_e')) \in \mathcal{I}(\kappa) \wedge u_r' \geq u_r \end{array} \right\}$$

The *guarantee* relation, $\mathcal{G} : \text{World} \times \text{World}$, is defined as follows:

$$\mathcal{G} \triangleq \left\{ ((c_l, g), (c_l', g')) \;\middle|\; \begin{array}{l} \forall r.\, g(r) = g'(r) \vee \exists \kappa, kv, kv', u_r, u_r', c_r, c_r', \mathcal{I}. \\ \quad g(r) = (kv, u_r, c_r, \mathcal{I}) \wedge g'(r) = (kv', u_r', c_r', \mathcal{I}) \wedge \kappa \sqsubseteq c_l(r) \\ \quad \wedge\, ((kv, u_r, c_r), (kv', u_r', c_r')) \in \mathcal{I}(\kappa) \wedge (c_l \bullet_c c_r)^{\perp} = (c_l' \bullet_c c_r')^{\perp} \end{array} \right\}$$

where for any element m from its domain M, the *orthogonal* is defined as:

$$m^{\perp} \triangleq \left\{ m' \;\middle|\; (m \bullet m') \downarrow \wedge m' \in \text{M} \right\}$$

The stabilisation says assertions remain true against the environment. Formally a set of worlds W is stable under certain execution tests ET if the set is closed under rely relation under the side conditions: (a) the key-value store transfer is allowed by the execution tests $(kv, kv') \in \text{ET}$; and (b) the new view under the new key-value store is able to progress. The first condition says there is at least one view from the environment that can trigger the transformation on the key-value store, and it is allowed by the execution tests. The second condition is more subtle, as it requires to also update the view to a new view u' so that there exists some non-empty fingerprints that is allowed to execute under the new view. It is especially useful when the execution tests are for serialisibility, which means that the view always shift to the end when the key-value store has been updated by the environment.

Definition 3.10 (Stable). A set of worlds $W \subseteq \text{World}$ is *stable*, written $\texttt{stable}\,(W, \text{ET})$, if and only if it is closed under the rely relation:

$$\begin{array}{rl} \texttt{stable}\,(W, \text{ET}) \triangleq \forall w, w'.\, w \in W \wedge (w, w') \in \mathcal{R} \wedge \exists kv, kv', u, u', \mathcal{F}. & \\ (kv, u) \in [\![w]\!] \wedge (kv', u') \in [\![w']\!] \wedge (kv, kv') \in \text{ET} & \\ \wedge\, \mathcal{F} \neq \mathbf{1}_{\mathcal{F}} \wedge u' \geq u \wedge (kv', u) \triangleright_{\text{ET}} \mathcal{F} : - & \\ \Longrightarrow\; w' \in W & \end{array}$$

If a update history heap update is allowed by consistency model, i.e. $(kv, kv') \in \text{ET}$, it means there exist some view u and operation set $\mathcal{F}$ allowed by the consistency model and the history is updated to kv' via them.

$$\begin{array}{l} (kv, kv') \in \text{ET} \triangleq kv = kv' \vee \exists u, u', \mathcal{F}, t. \\ \qquad (kv, u) \triangleright_{\text{ET}} \mathcal{F} : u' \wedge t \in \mathsf{fresh}\,(kv) \wedge kv' = \mathsf{UpdateMKVS}\,(kv, u, t, \mathcal{F}) \end{array}$$

3.4 Rules for Global

The PRCommit rule lifts the local effect of transaction T to global level by first converting global state to (local) observable state and then propagating the local fingerprint to the global state.

The UpdateMKVS and UpdateView in the repartition can be replaced by syntactic rules.

4 SOUNDNESS OF LOGIC

4.1 Transaction Soundness

Theorem 4.1 (Transaction soundness). *The transaction soundness is as follows:*

$$\forall p, \mathsf{T}, q. \vdash_I \{p\}\ \mathsf{T}\ \{q\} \implies \models_I \{p\}\ \mathsf{T}\ \{q\}$$

where,

$$\begin{aligned}\models_I \{p\}\ \mathsf{T}\ \{q\} \triangleq \forall \iota, s, s', ss, ss', \mathcal{F}, \mathcal{F}'.\ (ss, \mathcal{F}) \in [\![p]\!]_{\iota,s} \\ \wedge \vdash (s, ss, \mathcal{F}), \mathsf{T} \rightsquigarrow^* (s', ss', \mathcal{F}'), \mathtt{skip} \implies (ss', \mathcal{F}') \in [\![q]\!]_{\iota,s'}\end{aligned}$$

Proof. Induction on the derivations.

Base Case TRSkip.

We have $\mathsf{T} \equiv \mathtt{skip}$, $p \equiv q \equiv \mathrm{emp}$, thus $ss_p = ss_q = \mathbf{1}_{ss}$, $\mathcal{F} = \mathcal{F}'$ and $s = s'$, and then $(\mathbf{1}_{ss}, \mathbf{1}_{\mathcal{F}}) \in [\![\mathrm{emp}]\!]_{\iota,s'}$ holds.

Base Case TRAss.

We have $\mathsf{T} \equiv (\mathsf{x} := \mathsf{E})$, $p \equiv (\mathsf{x} \doteq \mathsf{E})$ and $q \equiv (\mathsf{x} \doteq \mathsf{E}[^{\mathsf{E}}/_{\mathsf{x}}])$ for some E, E and x such that $\mathsf{x} \notin \mathrm{fv}(\mathsf{E}) \wedge \mathsf{x} \in \mathrm{Vars}$. Given the transaction semantics (Fig. 3), it has $s' = s[\mathsf{x} \mapsto v]$ where $v = [\![\mathsf{E}[^{\mathsf{E}}/_{\mathsf{x}}]]\!]_{\iota,s}$. Since $\mathsf{x} \notin \mathrm{fv}(\mathsf{E})$, we know $[\![\mathsf{E}]\!]_{\iota,s} = [\![\mathsf{E}]\!]_{\iota,s'}$, and then $[\![\mathsf{E}[^{\mathsf{E}}/_{\mathsf{x}}]]\!]_{\iota,s} = [\![\mathsf{E}[^{\mathsf{E}}/_{\mathsf{x}}]]\!]_{\iota,s'}$. This means the assertions related to stack holds even thought the stack changes. Also because the heap and operation set remain unchanged, this is $ss = ss'$ and $\mathcal{F} = \mathcal{F}'$, we prove $(ss', \mathcal{F}') \in [\![q]\!]_{\iota,s'}$.

Base Case TRLookup.

We have $\mathsf{T} \equiv (\mathsf{x} := [\mathsf{E}])$ and four cases for pre- and post-conditions defined by the relation $\xrightarrow{\mathsf{R}(\mathsf{E},\mathsf{E})}$. In all the four cases, the heap remains the same $ss = ss' = \{k \mapsto v\}$ and the stack get updated to $s' = s[\mathsf{x} \mapsto v]$, yet since $\mathsf{x} \notin \mathrm{fv}(\mathsf{E})$, the logical value E and new logical address $\mathsf{E}[^{\mathsf{E}}/_{\mathsf{x}}]$ are evaluated to the same value v and address k as before. While different cases have different operation set. Also note that the evaluation of the pre- and post-conditions in all four cases are singleton sets. Now we do case analysis on four cases, especially on the operation set before and after.

If $p \equiv \mathsf{E} \hookrightarrow \mathsf{E}$ and $q \equiv \mathsf{E}[^{\mathsf{E}}/_{\mathsf{x}}] \overset{\mathsf{r}}{\hookrightarrow} \mathsf{E} * \mathsf{x} \doteq \mathsf{E}$, the only interpretation for pre-condition p is $(\{k \mapsto v\}, \emptyset)$ where $k = [\![\mathsf{E}]\!]_{\iota,s}$ and $v = [\![\mathsf{E}]\!]_{\iota,s}$. In this case, a read operation is added $\mathcal{F}' = \{(\mathsf{R}, k, v)\}$ and it is included in the interpretation of the post condition $\mathsf{E}[^{\mathsf{E}}/_{\mathsf{x}}] \overset{\mathsf{r}}{\hookrightarrow} \mathsf{E}$.

If $p \equiv \mathsf{E} \overset{\mathsf{r}}{\hookrightarrow} \mathsf{E}$ and $q \equiv \mathsf{E}[^{\mathsf{E}}/_{\mathsf{x}}] \overset{\mathsf{r}}{\hookrightarrow} \mathsf{E} * \mathsf{x} \doteq \mathsf{E}$, since there is already a read operation in the operation set, adding a new read operation to does not change the set, i.e. the new operation set is $\mathcal{F}' = \{(\mathsf{R}, k, v)\} \lll (\mathsf{R}, k, v) = \{(\mathsf{R}, k, v)\}$. This is exactly the post-condition. For the similar reason that the operation set remains the same in the rest two cases, so it is sound when $p \equiv \mathsf{E} \overset{\mathsf{w}}{\hookrightarrow} \mathsf{E}$ and $q \equiv \mathsf{E}[^{\mathsf{E}}/_{\mathsf{x}}] \overset{\mathsf{w}}{\hookrightarrow} \mathsf{E} * \mathsf{x} \doteq \mathsf{E}$, and when $p \equiv \mathsf{E} \overset{\mathsf{rw}}{\hookrightarrow} (\mathsf{E}, \mathsf{E}')$ and $q \equiv \mathsf{E}[^{\mathsf{E}}/_{\mathsf{x}}] \overset{\mathsf{rw}}{\hookrightarrow} (\mathsf{E}, \mathsf{E}') * \mathsf{x} \doteq \mathsf{E}'$.

Base Case TRMutate .

We have $\mathsf{T} \equiv ([\mathsf{E}_1] := \mathsf{E}_2)$ and four cases for pre- and post-conditions defined by the relation $\xrightarrow{\mathsf{W}(\mathsf{E},\mathsf{E})}$. In all the four cases, the stack remains untouched and the heap is updated to $ss' = \{k \mapsto v'\}$ where the logical address $[\![\mathsf{E}_1]\!]_{\iota,s'} = k$ and the new value $[\![\mathsf{E}_2]\!]_{\iota,s'} = v'$. Now we do case analysis on four case and focus on the operations before and after.

If $p \equiv \mathsf{E}_1 \hookrightarrow \mathrm{E}$ and $q \equiv \mathsf{E}_1 \overset{\mathsf{W}}{\hookrightarrow} \mathsf{E}_2$, a new write operation is added to the initially empty operation set, this is, $\mathcal{F}' = \{(\mathsf{W}, k, v)\}$ where $k = [\![\mathsf{E}_1]\!]_{\iota,s'}$ and $v = [\![\mathsf{E}_1]\!]_{\iota,s'}$. This is exactly the post-condition $\mathsf{E}_1 \overset{\mathsf{W}}{\hookrightarrow} \mathsf{E}_2$. If $p \equiv \mathsf{E}_1 \overset{\mathsf{W}}{\hookrightarrow} \mathrm{E}$ and $q \equiv \mathsf{E}_1 \overset{\mathsf{W}}{\hookrightarrow} \mathsf{E}_2$, the operation set before execution is $\mathcal{F} = \{(\mathsf{W}, k, v)\}$ where $k = [\![\mathsf{E}_1]\!]_{\iota,s}$ and $v = [\![\mathrm{E}]\!]_{\iota,s}$. Since the set only have the last write because of the property of the $\lll$ operator (Theorem 2.7), the set after is $\mathcal{F}' = \mathcal{F} \lll (\mathsf{W}, k, v') = \{(\mathsf{W}, k, v')\}$, where $v' = [\![\mathsf{E}_1]\!]_{\iota,s'}$. Note that the stack remains untouched, so we have $k = [\![\mathsf{E}_1]\!]_{\iota,s} = [\![\mathsf{E}_1]\!]_{\iota,s'}$. Thus, we have the proof for this case. The remaining two cases follow the same argument as the operations set only have the last write.

Inductive Case TRChoice.

We have $\mathsf{T} \equiv \mathsf{T}_1 + \mathsf{T}_2$, where $\vdash_I \{p\}\ \mathsf{T}_1\ \{q\}$ and $\vdash_I \{p\}\ \mathsf{T}_2\ \{q\}$ hold, for some $\mathsf{T}_1, \mathsf{T}_2, p, q$. Given the transaction semantics (Fig. 3), it either has $(s, ss, \mathcal{F}), \mathsf{T}_1 + \mathsf{T}_2 \rightsquigarrow (s, ss, \mathcal{F}), \mathsf{T}_1$ or $(s, ss, \mathcal{F}), \mathsf{T}_1 + \mathsf{T}_2 \rightsquigarrow (s, ss, \mathcal{F}), \mathsf{T}_2$. Let us pick T_1 and assume it can be reduced to skip from the initial state, i.e. $(\tau, ss, \mathcal{F}), \mathsf{T}_1 \rightsquigarrow^* (\tau', ss', \mathcal{F}'), \mathtt{skip}$. By the premiss of the rule $\vdash_I \{p\}\ \mathsf{T}_1\ \{q\}$ and the I.H., it implies $\vDash_I \{p\}\ \mathsf{T}_1\ \{q\}$, so we prove $(ss', \mathcal{F}') \in [\![q]\!]_{\iota,s'}$. Symmetrically, if we pick T_2, it gives the same result.

Inductive Case TRSeq.

We have $\mathsf{T} \equiv \mathsf{T}_1 ; \mathsf{T}_2$ where $\vdash_I \{p\}\ \mathsf{T}_1\ \{r\}$ and $\vdash_I \{r\}\ \mathsf{T}_2\ \{q\}$ hold, for some $\mathsf{T}_1, \mathsf{T}_2, p, q, r$. Given the transaction semantics (Fig. 3), it has $\vdash (s, ss, \mathcal{F}), \mathsf{T}_1 ; \mathsf{T}_2 \rightsquigarrow^* (s'', ss'', \mathcal{F}''), \mathtt{skip} ; \mathsf{T}_1 \rightsquigarrow (s'', ss'', \mathcal{F}''), \mathsf{T}_1 \rightsquigarrow^* (s', ss', \mathcal{F}'), \mathtt{skip}$ for some intermediate state $(s'', ss'', \mathcal{F}'')$. By the premiss of the rule and the I.H., we have $\vDash_I \{p\}\ \mathsf{T}_1\ \{r\}$ and so $(ss'', \mathcal{F}'') \in [\![r]\!]_{\iota,s''}$. The elimination of prefix skip does not change any state, so $(ss'', \mathcal{F}'') \in [\![r]\!]_{\iota,s''}$ still holds. Then, by the premiss and the I.H., we know $\vDash_I \{r\}\ \mathsf{T}_2\ \{q\}$ and therefore the proof that $(ss', \mathcal{F}') \in [\![q]\!]_{\iota,s'}$.

Inductive Case TRLoop.

Since the triple is only partial correct, meaning that if the transaction T terminates it will reach a state satisfying the post-condition q, it is sufficient to prove the follows,

$$\forall p, \mathsf{T}, n > 0. \vdash_I \{p\}\ \mathsf{T}^n\ \{p\} \implies \vDash_I \{p\}\ \mathsf{T}^n\ \{p\}$$

where,

$$\mathsf{T}^1 \triangleq \mathsf{T}$$
$$\mathsf{T}^n \triangleq \mathsf{T} ; \mathsf{T}^{n-1}$$

We prove that by induction on the number n. For $n = 1$, it is proven directly by the I.H. For $n > 1$, we have $\vdash (s, ss, \mathcal{F}), \mathsf{T} ; \mathsf{T}^{n-1} \rightsquigarrow^* (s'', ss'', \mathcal{F}''), \mathsf{T}^{n-1} \rightsquigarrow^* (s', ss', \mathcal{F}'), \mathtt{skip}$ for some intermediate state $(s'', ss'', \mathcal{F}'')$. By the premiss and the I.H., we have $\vDash_I \{p\}\ \mathsf{T}\ \{p\}$ and thus $(ss'', \mathcal{F}'') \in [\![p]\!]_{\iota,s''}$. Then by the I.H. that $\vDash_I \{p\}\ \mathsf{T}^{n-1}\ \{p\}$, we prove $(ss', \mathcal{F}') \in [\![p]\!]_{\iota,s'}$.

Inductive Case TRFrame.

We need to prove $\vDash_I \{p * r\}\ \mathsf{T}\ \{q * r\}$ given that $\vDash_I \{p\}\ \mathsf{T}\ \{q\}$. Assume variables $ss, ss', ss'', \mathcal{F}, \mathcal{F}', \mathcal{F}'', s, s'$ such that $(ss, \mathcal{F}) \in [\![p]\!]_{\iota,s}$, $(ss', \mathcal{F}') \in [\![q]\!]_{\iota,s'}$ and $(ss'', \mathcal{F}'') \in [\![r]\!]_{\iota,s}$. Since $p * r$, the point-wise composition is defined, i.e. $(ss \bullet_{ss} ss'', \mathcal{F} \bullet_{\mathcal{F}} \mathcal{F}'') \in [\![p * r]\!]_{\iota,s}$. The domain of the heaps and operations sets, therefore, are disjointed. The domain of a operations set is all the addresses $\mathrm{dom}(\mathcal{F}) \triangleq \mathcal{F}|_2$. We also know the domain of the operation set is a subset of the domain of the heap, $\mathrm{dom}(\mathcal{F}) \subseteq \mathrm{dom}(ss)$ which can be proven by induction on the structures of local assertions LAst. By the hypothesis $\vDash_I \{p\}\ \mathsf{T}\ \{q\}$, we know $(s, ss, \mathcal{F}), \mathsf{T} \rightsquigarrow^* (s', ss', \mathcal{F}'), \mathtt{skip}$. The heap after the execution should contain the same resources as before, this is $\mathrm{dom}(ss) = \mathrm{dom}(ss')$. Since $\mathrm{dom}(\mathcal{F}') \subseteq \mathrm{dom}(ss')$, we know the compositions $ss' \bullet_{ss} ss''$ and $\mathcal{F}' \bullet_{\mathcal{F}} \mathcal{F}''$ exist. This means the frame does not affect the semantic steps, i.e. $(s, ss \bullet_{ss} ss'', \mathcal{F} \bullet_{\mathcal{F}} \mathcal{F}''), \mathsf{T} \rightsquigarrow^* (s', ss' \bullet_{ss} ss'', \mathcal{F}' \bullet_{\mathcal{F}} \mathcal{F}''), \mathtt{skip}$. Finally, because there is no free variables overlap between r and p, q, the update of stack does not change the evaluation of the frame, this is, $[\![r]\!]_{\iota,s} = [\![r]\!]_{\iota,s'}$ which then gives us the result $(ss' \bullet_{ss} ss'', \mathcal{F}' \bullet_{\mathcal{F}} \mathcal{F}'') \in [\![q * r]\!]_{\iota,s'}$.

□

4.2 Program Soundness

The soundness judgement says for any terminated trace, a trace reaching skip, where every step in the trace is either a environment step that is allowed by the rely and consistency model, or the command get reduced one step, then if the precondition satisfies the initial world of the trace $w \in [\![P]\!]_{\iota,s}$ the postcondition should satisfy the final world $w' \in [\![Q]\!]_{\iota,s'}$.

Definition 4.2 (Soundness Judgement). The step predicate is defined as the follows,

$$\mathtt{step}\,(0, w, w', s, s', \mathtt{skip}, \mathsf{ET}) \triangleq s = s' \wedge w = w'$$

$$\begin{aligned}\mathtt{step}\,(n+1, w, w', s, s', \mathsf{C}, \mathsf{ET}) \triangleq\ & \exists kv, kv', w''.\\ & (w, w'') \in \mathcal{R} \wedge (kv, -, -) \in \lVert w \rVert \wedge (kv', -, -) \in \lVert w' \rVert\\ & \wedge (kv, kv') \in \mathsf{ET} \wedge \mathtt{step}\,(n, w'', w', s, s', \mathsf{C}, \mathsf{ET})\end{aligned}$$

$$\begin{aligned}\mathtt{step}\,(n+1, w, w', s, s', \mathsf{C}, \mathsf{ET}) \triangleq\ & \forall kv, kv'', u, u'', s'', \iota, \mathrm{cl}, \mathsf{C}'. (kv, u, -) \in \lVert w \rVert\\ & \wedge (s, kv, u), \mathsf{C} \xrightarrow{\mathsf{ET}} (s'', kv'', u''), \mathsf{C}'\\ & \wedge \exists w''. (w, w'') \in \mathcal{G} \wedge (kv'', u'', -) \in \lVert w'' \rVert \wedge \mathtt{step}\,(n, w'', w', s'', s', \mathsf{C}',\end{aligned}$$

Given above the soundness judgement is as the follows,

$$\mathsf{ET} \vDash_g \{P\}\ \mathsf{C}\ \{Q\} \triangleq \forall w, w', s, s', \iota, n.\ w \in [\![P]\!]_{\iota,s} \wedge \mathtt{step}\,(n, w, w', s, s', \mathsf{C}, \mathsf{ET}) \implies w' \in [\![Q]\!]_{\iota,s'}$$

Here it is a over approximation of the view environment that assumes the view environment does not intertwine with the current view. For many consistency models this judgement is good enough, because the specifications do not mentioned the view environment. Yet the soundness judgement for SI, for example, is not very strong since it includes some cases that are allowed locally but might be disallowed when the view environment comes in.

Theorem 4.3 (Per-thread soundness). *The per-thread soundness is the follows,*

$$\forall P, \mathsf{C}, Q.\ \mathsf{ET} \vdash_g \{P\}\ \mathsf{C}\ \{Q\} \implies \mathsf{ET} \vDash_g \{P\}\ \mathsf{C}\ \{Q\}$$

Proof. Induction on the derivations.

Base Case PRCommit.

We have $\mathsf{P} \equiv \lceil \mathsf{T} \rceil$. Because a transaction $\lceil \mathsf{T} \rceil$ is reduced by one step in the semantics, it is sufficient to prove for any world w that satisfies pre-condition, after arbitrary steps of rely, i.e. $(w, w') \in \mathcal{R}^*$ (Eq. (Stable Pre)) if the corresponding machine state $(kv', u', c') \in \lVert w' \rVert$, can transfers to a new state (kv'', u'', c'') (Eq. (Commit)) then again followed by arbitrary steps of rely $(w'', w''') \in \mathcal{R}^*$, the final world w''' should satisfy the post-condition Q (Eq. (Stable Post)).

SX: typesetting is a bit strange

$$\begin{aligned}&\mathtt{stable}\,(P, \mathsf{ET}) \implies \forall w, w', \iota, s.\ \exists kv, kv'.\\ &\quad w \in [\![P]\!]_{\iota,s} \wedge (w, w') \in \mathcal{R} && \text{(Stable Pre)}\\ &\qquad \wedge (kv, -, -) \in \lVert w \rVert \wedge (kv', -, -) \in \lVert w' \rVert \wedge (kv, kv') \in \mathsf{ET} \implies w' \in [\![P]\!]_{\iota,s}\\ &\qquad \vdash_I \{p\}\ \mathsf{T}\ \{q\} \wedge \mathsf{ET} \vdash P \Rrightarrow^{\{p\}\{q\}} Q \implies \forall w, kv, kv', u, u', \mathrm{cl}, \iota, s, s'.\\ &\qquad\quad w \in [\![P]\!]_{\iota,s} \wedge (kv, u, -) \in \lVert w \rVert \wedge \mathrm{cl}, (s, kv, u), \lceil \mathsf{T} \rceil \xrightarrow{\mathsf{ET}} (s', kv', u'), \mathtt{skip} && \text{(Commit)}\\ &\qquad\quad \wedge \exists w'. (w, w') \in \mathcal{G}(kv', u', -') \in \lVert w' \rVert \wedge w' \in [\![Q]\!]_{\iota,s'}\end{aligned}$$

$$\begin{aligned}&\mathtt{stable}\,(Q, \mathsf{ET}) \implies \forall w, w', \iota, s.\ \exists kv, kv'.\\ &\quad w \in [\![P]\!]_{\iota,s} \wedge (w, w') \in \mathcal{R} && \text{(Stable Post)}\\ &\qquad \wedge (kv, -, -) \in \lVert w \rVert \wedge (kv', -, -) \in \lVert w' \rVert \wedge (kv, kv') \in \mathsf{ET} \implies w' \in [\![Q]\!]_{\iota,s}\end{aligned}$$

SX: make sure the stack is correct

Stable pre-condition. The $\texttt{stable}\,(P, \mathsf{ET})$ predicate asserts any world w that satisfies the pre-condition P, if the world can transfer to another world w' through rely $\mathcal{R}$, and if the transfer also satisfies the consistency model ET, the new world w' satisfies the pre-condition, which implies Eq. (Stable Pre).

Commit. For any w, kv, u, ι, s such that $w \in [\![P]\!]_{\iota,s}$ and $(kv, u, c) \in \lfloor\!|w|\!\rfloor$, we know $(\text{snapshot}(kv, u), \mathbf{1}_{\mathcal{F}}) \in [\![p]\!]_{\iota,s}$, this is,

$$\forall w, kv, u, \iota, s.\ w \in [\![P]\!]_{\iota,s} \land (kv, u, -) \in \lfloor\!|w|\!\rfloor \implies (\text{snapshot}(kv, u), \mathbf{1}_{\mathcal{F}}) \in [\![p]\!]_{\iota,s} \tag{2}$$

Because of the soundness of transaction (Theorem 4.1), given an initial stack s and a logical environment ι, if the initial configuration $(s, ss, \mathbf{1}_{\mathcal{F}}), \mathsf{T}$ satisfies the pre-condition, i.e. $(ss, \mathbf{1}_{\mathcal{F}}) \in [\![p]\!]_{\iota,s}$, and if it can transfer to a final configuration $(s', ss', \mathcal{F}), \texttt{skip}$, the final state should satisfy the post-condition q. This is,

$$\begin{aligned}&\vdash_I \{p\}\ \mathsf{T}\ \{q\} \implies \\ &\quad \forall \iota, s, s', kv, u, ss', \mathcal{F}.\ (\text{snapshot}(kv, u), \mathbf{1}_{\mathcal{F}}) \in [\![p]\!]_{\iota,s} \land \vdash (s, ss, \mathbf{1}_{\mathcal{F}}), \mathsf{T} \rightsquigarrow (s', ss', \mathcal{F}), \texttt{skip} \implies (ss', \mathcal{F}) \in [\![q]\!]_{\iota,s}\end{aligned} \tag{3}$$

The repartition $\mathsf{ET} \vdash P \Rrightarrow^{\{p\}\{q\}} Q$ also asserts that any world w satisfying the pre-condition P, if the corresponding machine of the world (ignoring the capabilities here), i.e. (kv, u), can transfer to a new state (kv', u'), by committing the operations $\mathcal{F}$, then if a world w' can collapses to the new machine state (kv', u') and the transition (w, w') is allowed by both the guarantee and the consistency model, the new world w' should satisfy the post-condition.

$$\begin{aligned}&\forall w, kv, kv', u, u', \iota, s, s', t, \mathcal{F}.\ \exists w'. \\ &\quad w \in [\![P]\!]_{\iota,s} \land (kv, u, -) \in \lfloor\!|w|\!\rfloor \land t \in \text{fresh}\,(kv) \land (-, \mathcal{F}) \in [\![q]\!]_{\iota,s'} \\ &\quad \land kv' = \text{UpdateMKVS}\,(kv, u, t, \mathcal{F}) \land u' \geq \text{UpdateView}\,(kv', u, \mathcal{F}) \\ &\quad \land (kv', u', -) \in \lfloor\!|w'|\!\rfloor \land (w, w') \in \mathcal{G} \land (kv, u') \triangleright_{\mathsf{ET}} \mathcal{F} : u' \land w' \in [\![Q]\!]_{\iota,s'}\end{aligned} \tag{4}$$

First by Eq. (3), we substitute the $(-, \mathcal{F}) \in [\![q]\!]_{\iota,s'}$ in Eq. (4) by the transaction semantics,

$$\begin{aligned}&\forall w, kv, kv', u, u', \iota, s, s', t, \mathcal{F}.\ \exists w'. \\ &\quad w \in [\![P]\!]_{\iota,s} \land (kv, u, -) \in \lfloor\!|w|\!\rfloor \land t \in \text{fresh}\,(kv) \\ &\quad \land \vdash (s, \text{snapshot}(kv, u), \mathbf{1}_{\mathcal{F}}), \mathsf{T} \rightsquigarrow (s', ss', \mathcal{F}), \texttt{skip} \\ &\quad \land kv' = \text{UpdateMKVS}\,(kv, u, t, \mathcal{F}) \land u' \geq \text{UpdateView}\,(kv', u, \mathcal{F}) \\ &\quad \land (kv', u', -) \in \lfloor\!|w'|\!\rfloor \land (w, w') \in \mathcal{G} \land (kv, u') \triangleright_{\mathsf{ET}} \mathcal{F} : u' \land w' \in [\![Q]\!]_{\iota,s'}\end{aligned} \tag{5}$$

Hence we prove Eq. (Commit) by folding all the side conditions for a atomic transaction,

$$\begin{aligned}&\forall w, kv, kv', u, u', \iota, s, s', t, \mathcal{F}.\ \exists w'. \\ &\quad w \in [\![P]\!]_{\iota,s} \land (kv, u, -) \in \lfloor\!|w|\!\rfloor \land (s, kv, u), \lceil \mathsf{T} \rceil \xrightarrow{\mathsf{ET}} (s', kv', u'), \texttt{skip} \\ &\quad \land (kv', u', -) \in \lfloor\!|w'|\!\rfloor \land (w, w') \in \mathcal{G} \land w' \in [\![Q]\!]_{\iota,s'}\end{aligned}$$

Stable post-condition. It can be proven for the similar reason as the proof for stable pre-condition.

Inductive Case TRFrame.

Given $\vdash_g \{P\}\ \texttt{C}\ \{Q\}$ and $\texttt{stable}\,(R, \mathsf{ET})$, if $P * R$ holds, we need to prove $\vdash_g \{P * R\}\ \texttt{C}\ \{Q * R\} \implies \vDash_g \{P * R\}\ \texttt{C}\ \{Q * R\}$. By the I.H. that $\vdash_g \{P\}\ \texttt{C}\ \{Q\} \implies \vDash_g \{P\}\ \texttt{C}\ \{Q\}$, we know that for any trace defined by the `step` predicate with an initial world w satisfying the precondition P, the finial world w' satisfies the postcondition Q. We need to prove the frame R does not affect the trace.

□

SX: Need some help here to form the judgement

The program soundness is for the parallel rule. This is non-trivial as threads have the up-to-date history heap but not the views.

Definition 4.4 (Judgement).

SX: The judgement is not correct, should be a lift for the `step`.

Theorem 4.5 (Program soundness). *For any program* P*, if a precondition* P *satisfies a history heap* kv *and a thread environment* Env*, and if there exist a final history heap and thread environment* kv', Env' *such that* $(kv, Env, \mathsf{P}) \Rightarrow_{\mathsf{ET}} (kv', Env', (\lambda \mathrm{cl}.\ \mathtt{skip}))$ *holds, then the post-condition* Q *derived from the logic rules should satisfies the final configuration. This is,*

$$
\begin{array}{l}
\forall P, Q, \mathsf{P}, \iota, \mathsf{ET}, s, w, kv, kv', u, Env, Env'. \\
\quad \vdash_g \{P\}\ \mathsf{P}\ \{Q\} \wedge w \in [\![P]\!]_{\iota,s} \wedge (kv, u, -) \in \lfloor\!\lfloor w \rfloor\!\rfloor \wedge \mathrm{dom}(Env) = \mathrm{dom}(\mathsf{P}) \\
\quad \wedge \forall \mathrm{cl}.\ Env(\mathrm{cl}) = (s, u) \wedge (kv, Env, \mathsf{P}) \Rightarrow_{\mathsf{ET}} (kv', Env', (\lambda \mathrm{cl}.\ \mathtt{skip})) \\
\quad\quad \Longrightarrow\ \exists s', u', w'. \\
\quad\quad\quad \left(\begin{array}{l} \forall k, \mathrm{cl}', \mathsf{x}, v.\ Env(\mathrm{cl}')\lfloor_1\ (\mathsf{x}) = v \implies \\ \quad s'(\mathsf{x}) = v \wedge u'(k) = \max\left\{ n \mid \exists \mathrm{cl}''.\ Env(\mathrm{cl}'')\lfloor_2\ (k) = n \right\} \end{array} \right) \\
\quad\quad\quad \wedge (kv', u', -) \in \lfloor\!\lfloor w' \rfloor\!\rfloor \wedge w' \in [\![Q]\!]_{\iota,s'}
\end{array}
$$

Proof. Induction on derivations.

Base Case $\mathrm{dom}(\mathsf{P}) \equiv \{t\}$.

If there is only one thread, it is proved by the Theorem 4.3.

Base Case $\mathrm{dom}(\mathsf{P}) \equiv \{t, t'\}$.

We have $\vdash_g \{P_1 * P_2\}\ \mathsf{C}_1 \parallel \mathsf{C}_2\ \{Q_1 * Q_2\}$ and $\vdash_g \{P_i\}\ \mathsf{C}_i\ \{Q_i\}$ for $i \in \{1, 2\}$. Give the I.H. and the soundness result per thread, we need to prove for any possible traces for the entire program θ, there exists traces τ_1 and τ_2 from the first thread and second thread with the same state for history heap in each step. For brevity, we use numbers 1 and 2 both as subscript for two traces and the thread identifiers. The following also proves the local capabilities from the two local traces always compatible.

$$
\begin{array}{l}
\forall \theta.\ \exists \tau_1, \tau_2. \\
\left(\begin{array}{l} \forall kv, kv', Env, Env'.\ ((kv, Env), (kv', Env')) \in \theta \\ \quad \wedge\ \mathrm{dom}(Env) = \mathrm{dom}(Env') = \{1, 2\} \wedge \exists w_1, w_2, w_1', w_2', i \in \{1, 2\}\ . \\ \quad\quad (w_i, w_i') \in \tau_i \wedge kv = \lfloor\!\lfloor w_i \rfloor\!\rfloor\lfloor_1 \wedge kv' = \lfloor\!\lfloor w_i' \rfloor\!\rfloor\lfloor_1 \wedge (w_1\lfloor_1 \bullet_c w_2\lfloor_1) \downarrow \wedge (w_1'\lfloor_1 \bullet_c w_2'\lfloor_1) \downarrow \\ \quad\quad \wedge\ Env(i)\lfloor_2 = \lfloor\!\lfloor w_i \rfloor\!\rfloor\lfloor_2 \wedge Env'(i)\lfloor_2 = \lfloor\!\lfloor w_i' \rfloor\!\rfloor\lfloor_2 \end{array} \right)
\end{array}
$$

We prove by constructing the traces tau_1 and τ_2 inductively. Initially, because $P_1 * P_2$ is defined, we pick any states w_1^0 and w_2^0 such that $w_1^0 \in [\![P_1]\!]_{\iota,s}$, $w_2^0 \in [\![P_2]\!]_{\iota,s}$ and $(w_1^0 \bullet_w w_2^0) \downarrow$. This means the initial history heaps and views match, i.e.

$$
\forall i \in \{1, 2\}\ .\ kv^0 = \lfloor\!\lfloor w_1^0 \bullet_w w_2^0 \rfloor\!\rfloor\lfloor_1 = \lfloor\!\lfloor w_i^0 \rfloor\!\rfloor\lfloor_1 \wedge Env^0(i)\lfloor_2 = \lfloor\!\lfloor w_i^0 \rfloor\!\rfloor\lfloor_2
$$

and the local capabilities are compatible,

$$
(w_1^0\lfloor_1 \bullet_c w_2^0\lfloor_1) \downarrow
$$

Assume there exists two traces τ_1, τ_2 after i steps, let consider the next step for the entire program. Now two traces τ_1 and τ_2 where the current finial states are w_1^i and w_2^j such that $\lfloor\!\lfloor w_1^i \rfloor\!\rfloor\lfloor_1 = \lfloor\!\lfloor w_2^j \rfloor\!\rfloor\lfloor_1$ and $(w_1^i\lfloor_1 \bullet_c w_2^j\lfloor_1) \downarrow$. There are two possibilities for the first thread: (a) it commits a transaction; or (b) it take a rely step which updates the history heap and shared capabilities. If the first thread commits a transaction, which means the last step of the new trace $\tau_1' = \tau_1 \cup \{(w_1^i, w_1^{i+1})\}$ it is

allowed by the guarantee $(w_1^i, w_1^{i+1}) \in \mathcal{G}$. By the Theorem 4.6, there exists w_2^{j+1} satisfying the conditions, therefore we can construct the new trace for the second thread as $\tau_2' = \tau_2 \cup \{(w_2^j, w_2^{j+1})\}$. If the first thread take a rely step, i.e. $\tau_1' = \tau_1 \cup \{(w_1^i, w_1^{i+1})\}$ such that $(w_1^i, w_1^{i+1}) \in \mathcal{R}$, there are two possibilities, this transfer is triggered by a capability is either included in the second thread or not. □

Lemma 4.6 (Locality of update). *When a thread commits a transaction that is allowed by guarantee, the effect to the history heap is included in the rely of other threads.*

$$\forall w, w', w''. (w\lfloor_1 \bullet_c w'\lfloor_1)\downarrow \wedge \lfloor\!\lfloor w \rfloor\!\rfloor\lfloor_1 = \lfloor\!\lfloor w' \rfloor\!\rfloor\lfloor_1 \wedge (w, w'') \in \mathcal{G} \\ \implies \exists w'''. (w', w''') \in \mathcal{R} \wedge (w''\lfloor_1 \bullet_c w'''\lfloor_1)\downarrow \wedge \lfloor\!\lfloor w'' \rfloor\!\rfloor\lfloor_1 = \lfloor\!\lfloor w''' \rfloor\!\rfloor\lfloor_1$$

Proof. Assume two worlds w and w' such that the composition of the local capabilities is defined $(w\lfloor_1 \bullet_c w'\lfloor_1)\downarrow$ and the global history heap states are the same, i.e. $\lfloor\!\lfloor w \rfloor\!\rfloor\lfloor_1 = \lfloor\!\lfloor w' \rfloor\!\rfloor\lfloor_1$. If the first world can transfer to a new world w'' allowed by guarantee $(w, w'') \in \mathcal{G}$, by the Theorem 3.9, we know that for any region r that has been updated, there exists a κ such that $\kappa \sqsubseteq c(r)$ which gives the permission. Formally, let $(g, c) = w$ and $(g', c') = w'$,

$$\forall r.\, g(r) = g'(r) \vee \\ g(r) \neq g'(r) \wedge \exists kv, kv', u, u', c'', c''', \mathcal{I}, \kappa. \\ g(r) = (kv, u, c'', \mathcal{I}) \wedge g'(r) = (kv', u', c''', \mathcal{I}) \wedge \kappa \sqsubseteq c(r) \\ \wedge ((kv, u, c''), (kv', u', c''')) \in \mathcal{I}(\kappa)$$

We can construct a world $w''' = (c''', g''')$ that satisfies the rely $(w', w''') \in \mathcal{R}$. First, we take the same local capabilities as the w'. This is,

$$c''' = w'\lfloor_1$$

Then we propagate history heaps and shared capabilities from w'' to w''' but keep the views as the same in the w',

$$\forall r.\, \exists kv, u, c, \mathcal{I}\,.\, g'''(r) = (kv, u, c, \mathcal{I}) \wedge w''\lfloor_2\,(r) = (kv, -, c, \mathcal{I}) \wedge w'\lfloor_2\,(r) = (-, u, -, \mathcal{I})$$

Combining the two formulae with the local capabilities are compatible $(c \bullet_c c')\downarrow$, It is easy to see $(w', w''') \in r^u$ by Theorem 3.9, therefore $(w', w''') \in \mathcal{R}$. Given the way we construct w''', it directly proves $\lfloor\!\lfloor w'' \rfloor\!\rfloor\lfloor_1 = \lfloor\!\lfloor w''' \rfloor\!\rfloor\lfloor_1$. Because the guarantee require the orthogonal of capabilities remains the same so alter the update, the local capabilities are still compatible. □

5 EXAMPLES

5.1 Litmus Test For SI

SX: This is bad words. Keep like this and change later

This example distinguishes serialisibility from snapshot isolation. To understand better, we associate unique capabilities for addresses, which indicates the latest values. Under serialisibility, a thread always observes be the latest values. While under snapshot isolation, it is sufficient to only see the latest values for addresses before the next transaction, if those addresses will be overwrite by the next transaction. After transaction a thread updates its view to the latest, yet it does not need to be always the latest while other threads interfere with the database which change the history heap.

Since the capabilities always remember the latest values, therefore in the interference I, if a transaction writes to an address, it also updates the capabilities to the value written, for example $[\mathrm{x}(\mathrm{N})]^{\mathrm{R}} \overset{\mathrm{U}}{\hookrightarrow} [\mathrm{x}(1)]^{\mathrm{R}}$. This capabilities also helps to understand the repartition especially under

snapshot isolation, in a way that after committing a transaction, the view should be updated to the latest for all addresses.

$$\boxed{\mathtt{x} \mapsto - * \mathtt{y} \mapsto -}^{\mathtt{R}}_{I}$$

$$P1 : I : \exists \mathtt{N}.\, \mathtt{R} : \mathtt{x} \overset{\mathtt{r}}{\hookrightarrow} 0 * \mathtt{y} \overset{\mathtt{w}}{\hookrightarrow} 1$$
$$\exists \mathtt{N}.\, \mathtt{L} : \mathtt{x} \overset{\mathtt{w}}{\hookrightarrow} 1 * \mathtt{y} \overset{\mathtt{r}}{\hookrightarrow} 0$$
$$\mathtt{0} : \exists \mathtt{A} \in \{\mathtt{x}, \mathtt{y}\}\,.\, \mathtt{A} \overset{\mathtt{r}}{\hookrightarrow} \mathtt{N}$$

$$\left\{\boxed{\mathtt{x} \mapsto \{0\} * \mathtt{y} \mapsto \{0\}}^{\mathtt{R}}_{I} * [\mathtt{L}]^{\mathtt{R}} * [\mathtt{R}]^{\mathtt{R}}\right\}$$

$$stable - PL : \left\{\begin{array}{l}[\mathtt{L}]^{\mathtt{R}} * \boxed{\mathtt{x} \mapsto \{0\} * \mathtt{y} \mapsto \{0\}}^{\mathtt{R}}_{I} \vee \\ \boxed{\mathtt{x} \mapsto \{0\} * \mathtt{y} \mapsto \{0,1\}}^{\mathtt{R}}_{I}\end{array}\right\}$$

$$\left[\begin{array}{l}\{\mathtt{x} \hookrightarrow 0 * (\mathtt{y} \hookrightarrow 0 \vee \mathtt{y} \hookrightarrow 1)\} \\ \mathtt{b} := [\mathtt{y}]; \\ \mathtt{if}\,(\mathtt{b} = 0)\{[\mathtt{x}] := 1;\} \\ \left\{\mathtt{x} \overset{\mathtt{w}}{\hookrightarrow} 1 * \mathtt{y} \overset{\mathtt{r}}{\hookrightarrow} 0 \vee \mathtt{x} \hookrightarrow 0 * \mathtt{y} \overset{\mathtt{r}}{\hookrightarrow} 1\right\}\end{array}\right]$$

$$QL : \left\{\begin{array}{l}[\mathtt{L}]^{\mathtt{R}} * \boxed{\mathtt{x} \mapsto \{1\} * \mathtt{y} \mapsto \{0\}}^{\mathtt{R}}_{I} \\ \vee \boxed{\mathtt{x} \mapsto \{1\} * \mathtt{y} \mapsto \{1\}}^{\mathtt{R}}_{I} \\ \vee \boxed{\mathtt{x} \mapsto \{0\} * \mathtt{y} \mapsto \{1\}}^{\mathtt{R}}_{I}\end{array}\right\}$$

$$stable - QL : \left\{\begin{array}{l}[\mathtt{L}]^{\mathtt{R}} * \boxed{\mathtt{x} \mapsto \{1\} * \mathtt{y} \mapsto \{0,1\}}^{\mathtt{R}}_{I} \\ \vee \boxed{\mathtt{x} \mapsto \{1\} * \mathtt{y} \mapsto \{1\}}^{\mathtt{R}}_{I} \\ \vee \boxed{\mathtt{x} \mapsto \{0\} * \mathtt{y} \mapsto \{1\}}^{\mathtt{R}}_{I}\end{array}\right\}$$

$$stable - PR : \left\{\begin{array}{l}[\mathtt{R}]^{\mathtt{R}} * \boxed{\mathtt{x} \mapsto \{0\} * \mathtt{y} \mapsto \{0\}}^{\mathtt{R}}_{I} \vee \\ \boxed{(\mathtt{x} \mapsto \{0,1\} * \mathtt{y} \mapsto \{0\}}^{\mathtt{R}}_{I}\end{array}\right\}$$

$$\left[\mathtt{a} := [\mathtt{x}];\quad \mathtt{if}\,(\mathtt{a} = 0)\{[\mathtt{y}] := 1;\}\right]$$

$$QR : \left\{\begin{array}{l}[\mathtt{R}]^{\mathtt{R}} * \boxed{\mathtt{x} \mapsto \{0\} * \mathtt{y} \mapsto \{1\}}^{\mathtt{R}}_{I} \\ \vee \boxed{\mathtt{x} \mapsto \{1\} * \mathtt{y} \mapsto \{1\}}^{\mathtt{R}}_{I} \\ \vee \boxed{\mathtt{x} \mapsto \{1\} * \mathtt{y} \mapsto \{0\}}^{\mathtt{R}}_{I}\end{array}\right\}$$

$$stable - QR : \left\{\begin{array}{l}[\mathtt{R}]^{\mathtt{R}} * \boxed{\mathtt{x} \mapsto \{0,1\} * \mathtt{y} \mapsto \{1\}}^{\mathtt{R}}_{I} \\ \vee \boxed{\mathtt{x} \mapsto \{1\} * \mathtt{y} \mapsto \{1\}}^{\mathtt{R}}_{I} \\ \vee \boxed{\mathtt{x} \mapsto \{1\} * \mathtt{y} \mapsto \{0\}}^{\mathtt{R}}_{I}\end{array}\right\}$$

$$QL * QR : \left\{\begin{array}{l}[\mathtt{L}]^{\mathtt{R}} * [\mathtt{R}]^{\mathtt{R}} * \boxed{\mathtt{x} \mapsto \{0\} * \mathtt{y} \mapsto \{1\}}^{\mathtt{R}}_{I} \\ \vee \boxed{\mathtt{x} \mapsto \{1\} * \mathtt{y} \mapsto \{1\}}^{\mathtt{R}}_{I} \vee \boxed{\mathtt{x} \mapsto \{1\} * \mathtt{y} \mapsto \{0\}}^{\mathtt{R}}_{I}\end{array}\right\}$$

For serialisibility, the stable PL and PR rules out those views that cannot progress via any possible transactions. This mean the view must at the end of the history heap. The post conditions QL and QR is stronger because the consistency model check in the repartition

$$\left\{\boxed{\mathtt{x} \mapsto 0 * \mathtt{y} \mapsto 0}^{\mathtt{R}}_{I} * [\mathtt{L}]^{\mathtt{R}} * [\mathtt{R}]^{\mathtt{R}}\right\}$$

$$stable - PL : \left\{[\mathtt{L}]^{\mathtt{R}} * \boxed{\mathtt{x} \mapsto \{0\} * (\mathtt{y} \mapsto \{0\} \vee \mathtt{y} \mapsto \{1\})}^{\mathtt{R}}_{I}\right\}$$

$$\left[\begin{array}{l}\{\mathtt{x} \hookrightarrow 0 * (\mathtt{y} \hookrightarrow 0 \vee \mathtt{y} \hookrightarrow 1)\} \\ \mathtt{b} := [\mathtt{y}]; \\ \mathtt{if}\,(\mathtt{b} = 0)\{[\mathtt{x}] := 1;\} \\ \left\{\mathtt{x} \overset{\mathtt{w}}{\hookrightarrow} 1 * \mathtt{y} \overset{\mathtt{r}}{\hookrightarrow} 0 \vee \mathtt{x} \hookrightarrow 0 * \mathtt{y} \overset{\mathtt{r}}{\hookrightarrow} 1\right\}\end{array}\right]$$

$$QL : \left\{\begin{array}{l}[\mathtt{L}]^{\mathtt{R}} * \boxed{\mathtt{x} \mapsto \{1\} * \mathtt{y} \mapsto \{0\}}^{\mathtt{R}}_{I} \\ \vee \boxed{\mathtt{x} \mapsto \{0\} * \mathtt{y} \mapsto \{1\}}^{\mathtt{R}}_{I}\end{array}\right\}$$

$$PR : \left\{[\mathtt{R}]^{\mathtt{R}} * \boxed{(\mathtt{x} \mapsto \{0\} \vee \mathtt{x} \mapsto \{1\}) * \mathtt{y} \mapsto \{0\}}^{\mathtt{R}}_{I}\right\}$$

$$\left[\mathtt{a} := [\mathtt{x}];\quad \mathtt{if}\,(\mathtt{a} = 0)\{[\mathtt{y}] := 1;\}\right]$$

$$QR : \left\{\begin{array}{l}[\mathtt{R}]^{\mathtt{R}} * \boxed{\mathtt{x} \mapsto \{0\} * \mathtt{y} \mapsto \{1\}}^{\mathtt{R}}_{I} \\ \vee \boxed{\mathtt{x} \mapsto \{1\} * \mathtt{y} \mapsto \{0\}}^{\mathtt{R}}_{I}\end{array}\right\}$$

$$QL * QR : \left\{[\mathtt{L}]^{\mathtt{R}} * [\mathtt{R}]^{\mathtt{R}} * \boxed{\mathtt{x} \mapsto \{0\} * \mathtt{y} \mapsto \{1\}}^{\mathtt{R}}_{I} \vee \boxed{\mathtt{x} \mapsto \{1\} * \mathtt{y} \mapsto \{0\}}^{\mathtt{R}}_{I}\right\}$$

SX: ——————UNTIL HERE——————

5.2 Litmus Tests

This is used to test the semantics.

$$\begin{bmatrix} \texttt{x} := [k_x]; \\ \texttt{if (x = 0)\{} \\ \quad [k_y] := 1 \end{bmatrix} \Vert \begin{bmatrix} \texttt{y} := [k_y]; \\ \texttt{if (x = 0)\{} \\ \quad [k_x] := 1 \end{bmatrix} \Vert \left[[k_x] := 2\right] \Vert \left[[k_y] := 2\right]$$

Simple merging example.

$$\begin{bmatrix} \texttt{x} := [k_x]; \\ \texttt{if (x = 0)\{} \\ \quad [k_y] := 1 \end{bmatrix} \Vert \begin{bmatrix} \texttt{y} := [k_y]; \\ \texttt{if (x = 0)\{} \\ \quad [k_x] := 1 \end{bmatrix}$$

Long fork.

$$\begin{bmatrix} \left[[k_x] := 1;\right] \\ \texttt{x} := [k_x]; \\ \texttt{y} := [k_y]; \\ \texttt{if (x = 1 and y = 0)\{} \\ \quad [k_w] := 1 \end{bmatrix} \Vert \begin{bmatrix} \left[[k_y] := 1;\right] \\ \texttt{x} := [k_x]; \\ \texttt{y} := [k_y]; \\ \texttt{if (x = 0 and y = 1)\{} \\ \quad [k_z] := 1 \end{bmatrix}$$

Test the rely is "closed"" correctly, esp. location x and y transfer from (0,0) to (2,1).

$$\left[[k_x] := 1\right] \Vert \begin{bmatrix} \texttt{if (x = 0)\{} \\ \quad [k_y] := 1 \end{bmatrix} \Vert \begin{bmatrix} \texttt{x} := [k_x]; \\ \texttt{y} := [k_y]; \\ \texttt{if (x = 1 and y = 0)\{} \\ \quad [k_x] := 2 \end{bmatrix} \Vert \left[anything\right]$$

Test the soundness of merge

$$\begin{array}{c} \{(0,0)\} \\ \{(0,0) \vee (3,0)\} \\ \begin{bmatrix} \texttt{if (x = 0)\{} \\ \texttt{[y] := 1; \}} \end{bmatrix} \\ \{(0,1) \vee (3,0) \vee (3,1)\} \\ \begin{bmatrix} \texttt{if (x = 0)\{} \\ \texttt{[y] := 2; \}} \end{bmatrix} \\ \{(0,2) \vee (3,0) \vee (3,1)\} \end{array} \Vert \begin{array}{c} \\ \\ \{(0,0) \vee (0,1) \vee (0,2)\} \\ \begin{bmatrix} \texttt{if (y = 0)\{} \\ \texttt{[x] := 3; \}} \end{bmatrix} \\ \{(3,0) \vee (0,1) \vee (0,2) \vee (3,1) \vee (3,2)\} \end{array}$$

$$\{(0,2) \vee (3,0) \vee (3,1)\}$$

$$Miss(3,2)$$

In this example, even with a global part and local part, I am not sure it will work. A vector of values should work in this example.

$$
\begin{array}{c}
\{([0],[0])\} \\
\left\| \begin{array}{l}
\{([0],[0]) \vee ([0,3],[0])\} \\
\left[\begin{array}{l}\texttt{if}\,(x=0)\{\\ [y]:=1;\}\end{array}\right] \\
\{([0],[0,1]) \vee ([0,3],[0]) \\ \vee ([0,3],[0,1])\} \\
\left[\begin{array}{l}\texttt{if}\,(x=0)\{\\ [y]:=2;\}\end{array}\right] \\
\{([0],[0,1,2]) \vee ([0,3],[0]) \\ \vee ([0,3],[0,1]) \vee ([0,3],[0,1,2])\}
\end{array} \right. \\
\left. \begin{array}{l}
\{([0],[0]) \vee ([0],[0,1]) \vee ([0],[0,2]) \vee \\ ([0],[0,1,2]) \vee ([0],[0,2,1])\} \\
\left[\begin{array}{l}\texttt{if}\,(y=0)\{\\ [x]:=3;\}\end{array}\right] \\
\{([0,3],[0]) \vee ([0],[0,1]) \vee ([0],[0,2]) \vee \\ ([0],[0,1,2]) \vee ([0],[0,2,1]) \\ \vee([0,3],[0,1]) \vee ([0,3],[0,2]) \vee \\ ([0,3],[0,1,2]) \vee ([0,3],[0,2,1])\} \\
\{([0,3],[0]) \vee ([0],[0,1,2]) \\ \vee([0,3],[0,1]) \vee ([0,3],[0,1,2])\}
\end{array} \right.
\end{array}
$$

SX: game inventory, auction system, key-value benchmark

5.3 Single increment and multi-reader.

$\boxed{x \mapsto -}^{R}_{I}$
Inc $\bullet_\kappa$ Inc is undefined
$\emptyset$ is the unit element

$$I : \texttt{Inc} : x \overset{rw}{\hookrightarrow} (n, n+1)$$
$$\emptyset : x \overset{r}{\hookrightarrow} n$$

5.3.1 *SER,SI, Causal.*

$$
\begin{array}{c}
\{\boxed{x \mapsto 0}^{R}_{I} * [\texttt{Inc}]^{R}\} \\
\left\| \begin{array}{l}
\{\boxed{x \mapsto 0}^{R}_{I} * [\texttt{Inc}]^{R}\} \\
\left[\begin{array}{l}\{x \hookrightarrow 0\} \\ a := [x]; \\ \{x \overset{r}{\hookrightarrow} 0 \wedge a = 0\} \\ [x] := a+1; \\ \{x \overset{rw}{\hookrightarrow} (0,1) \wedge a = 0\}\end{array}\right] \\
\{\boxed{x \mapsto 1}^{R}_{I} * [\texttt{Inc}]^{R}\} \\
\left[\begin{array}{l}\{x \hookrightarrow 1\} \\ b := [x]; \\ \{x \overset{r}{\hookrightarrow} 1\}\end{array}\right] \\
\{\boxed{x \mapsto 1}^{R}_{I} * [\texttt{Inc}]^{R}\} \\
\left[\begin{array}{l}\{x \hookrightarrow 1\} \\ a := [x]; [x] := a+1; \\ \{x \overset{rw}{\hookrightarrow} (1,2)\}\end{array}\right] \\
\{\boxed{x \mapsto 2}^{R}_{I} * [\texttt{Inc}]^{R}\}
\end{array} \right. \\
\left. \begin{array}{l}
\{\exists n.\, \boxed{x \mapsto n}^{R}_{I} \wedge n \geq 0\} \\
\left[\begin{array}{l}\{x \hookrightarrow n\} \\ c := [x]; \\ \{x \overset{r}{\hookrightarrow} n\}\end{array}\right] \\
\{\exists n.\, \boxed{x \mapsto n}^{R}_{I} \wedge n \geq 0\}
\end{array} \right.
\end{array}
$$

5.4 Multiple increments

$$\boxed{x \mapsto - \mid x \mapsto -}^{R}_{I}$$

Inc is the unit element

5.4.1 SER.

5.4.2 Causal.

5.5 Two associated bank accounts

$$\boxed{x \mapsto - * y \mapsto -}^{R}_{I}$$

$L \bullet_{\kappa} L$ is undefined

$R \bullet_{\kappa} R$ is undefined

$\emptyset$ is the unit

$$\begin{aligned} I : L : x \overset{r}{\hookrightarrow} j * y \overset{rw}{\hookrightarrow} (k, k - n) \wedge j + k \geq n \\ R : x \overset{rw}{\hookrightarrow} (j, j - n) * y \overset{r}{\hookrightarrow} k \wedge j + k \geq n \\ \emptyset : x \overset{r}{\hookrightarrow} j * y \overset{r}{\hookrightarrow} k \end{aligned}$$

5.5.1 SER.

$$\left\{\boxed{x \mapsto 60 * y \mapsto 60}^{R}_{I} * [L]^{R} * [R]^{R}\right\}$$

$$\left\{\boxed{x \mapsto 60 * (y \mapsto 60 \vee y \mapsto -40)}^{R}_{I} * [L]^{R}\right\}$$

$$\left[\begin{array}{l} \{x \hookrightarrow 60 * (y \hookrightarrow 60 \vee y \hookrightarrow -40)\} \\ a := [x]; \\ b := [y]; \\ \texttt{if}\,(a + b \geq 100)\{ \\ \quad [x] := a - 100; \\ \} \\ \left\{\begin{array}{l} x \overset{rw}{\hookrightarrow} (60, -40) * y \overset{r}{\hookrightarrow} 60) \\ \vee\, (x \overset{r}{\hookrightarrow} 60 * y \overset{r}{\hookrightarrow} -40) \end{array}\right\} \end{array}\right]$$

$$\left\{\boxed{x \mapsto -40 * y \mapsto 60 \vee x \mapsto 60 * y \mapsto -40}^{R}_{I} * [L]^{R}\right\}$$

$$\left\{\boxed{(x \mapsto 60 \vee x \mapsto -40) * y \mapsto 60}^{R}_{I} * [R]^{R}\right\}$$

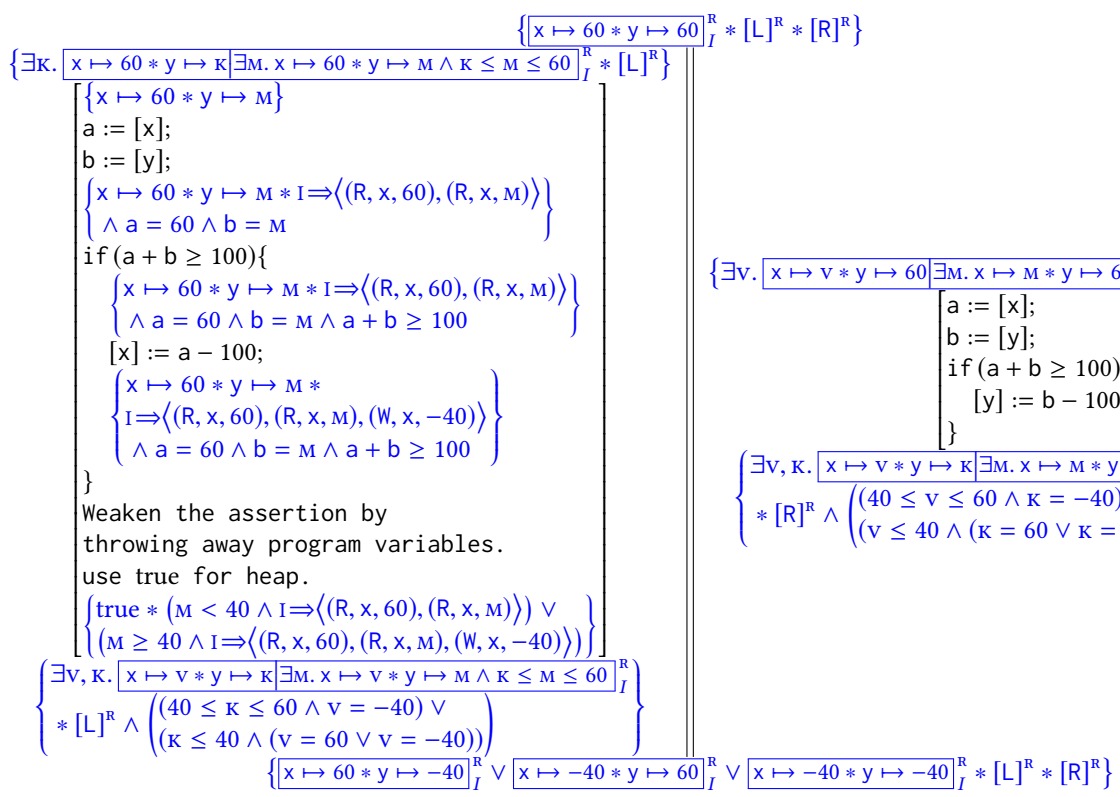
$$\left[\begin{array}{l} a := [x]; \\ b := [y]; \\ \texttt{if}\,(a + b \geq 100)\{ \\ \quad [y] := b - 100; \\ \} \end{array}\right]$$

$$\left\{\boxed{x \mapsto -40 * y \mapsto 60 \vee x \mapsto 60 * y \mapsto -40}^{R}_{I} * [R]^{R}\right\}$$

$$\left\{\boxed{x \mapsto -40 * y \mapsto 60 \vee x \mapsto 60 * y \mapsto -40}^{R}_{I} * [L]^{R} * [R]^{R}\right\}$$

5.5.2 SI/PSI.

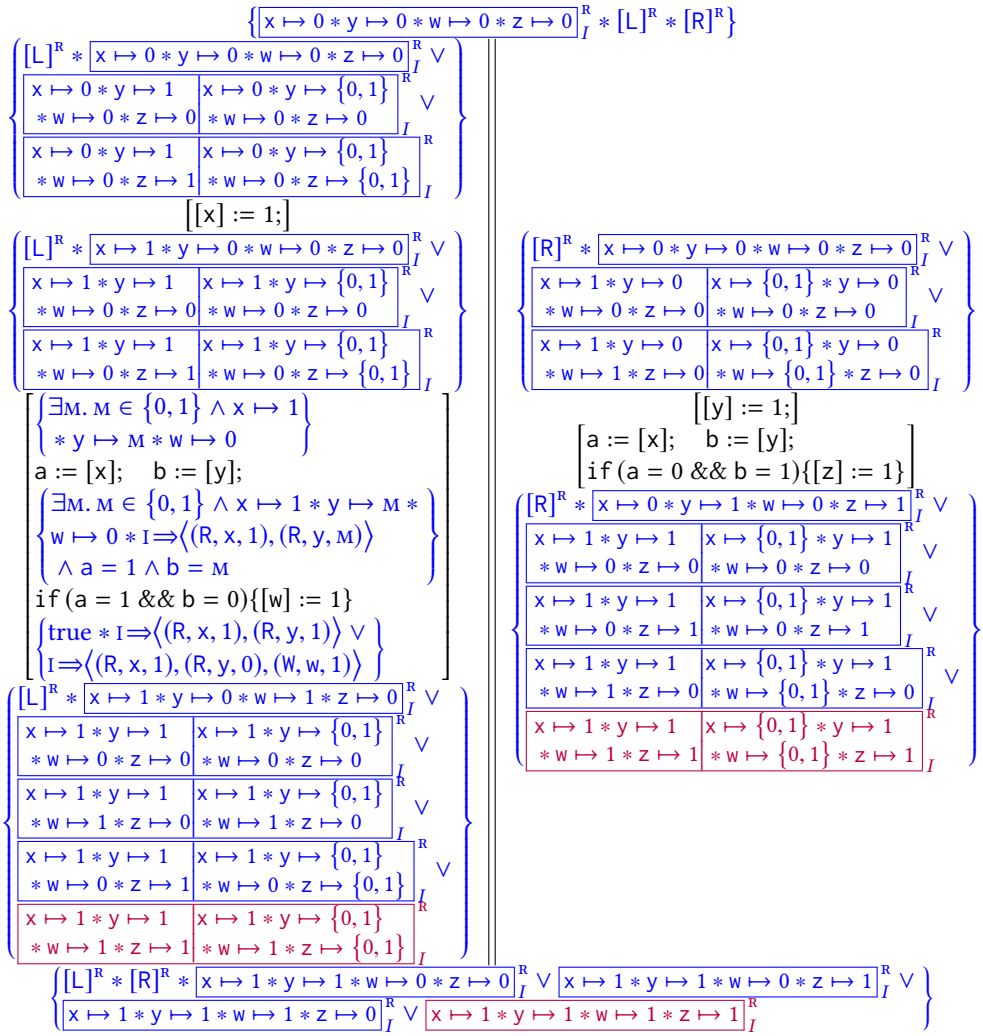
SX: This example not sure one assertion inside the box is enough only looking at the syntax, while from the point of the left thread, the environment might not update their view after the left thread commit its transaction. Then because of rely under SI, it should produce an extra case that -40 and -40.



5.6 Litmus Test For SI, Long fork

$$\left[\mathrm{x} \mapsto - * \mathrm{y} \mapsto - * \mathrm{w} \mapsto - * \mathrm{z} \mapsto -\right]^{\mathrm{R}}_{I}$$

$$\begin{aligned}
I:\ & \mathsf{L}: \mathrm{x} \mapsto - \ @\ \{(\mathrm{W}, \mathrm{x}, 1)\} \\
& \mathsf{L}: \mathrm{x} \mapsto 1 * \mathrm{y} \mapsto \{0,1\} * \mathrm{w} \mapsto - \ @\ \{(\mathrm{R}, \mathrm{x}, 1), (\mathrm{R}, \mathrm{y}, 0), (\mathrm{W}, \mathrm{w}, 1)\} \\
& \mathsf{L}: \mathrm{x} \mapsto 1 * \mathrm{y} \mapsto \{0,1\} * \mathrm{w} \mapsto - * \mathrm{z} \mapsto 0 \ @\ \{(\mathrm{R}, \mathrm{x}, 1), (\mathrm{R}, \mathrm{y}, 0), (\mathrm{W}, \mathrm{w}, 1)\} \\
& \mathsf{R}: \mathrm{y} \mapsto - \ @\ \{(\mathrm{W}, \mathrm{y}, 1)\} \\
& \mathsf{R}: \mathrm{x} \mapsto \{0,1\} * \mathrm{y} \mapsto 1 * \mathrm{z} \mapsto - \ @\ \{(\mathrm{R}, \mathrm{x}, 0), (\mathrm{R}, \mathrm{y}, 1), (\mathrm{W}, \mathrm{z}, 1)\} \\
& \mathsf{R}: \mathrm{x} \mapsto \{0,1\} * \mathrm{y} \mapsto 1 * \mathrm{w} \mapsto 0 * \mathrm{z} \mapsto - \ @\ \{(\mathrm{R}, \mathrm{x}, 0), (\mathrm{R}, \mathrm{y}, 1), (\mathrm{W}, \mathrm{z}, 1)\} \\
& \mathsf{Rd}: \exists \mathrm{a} \in \{\mathrm{x}, \mathrm{y}\}, \nu, \kappa.\ \mathrm{a} \mapsto \kappa \wedge (\kappa = 1 \wedge \nu \in \{0,1\}) \vee \kappa = \nu = 0 \ @\ \{(\mathrm{R}, \mathrm{a}, \nu)\}
\end{aligned}$$

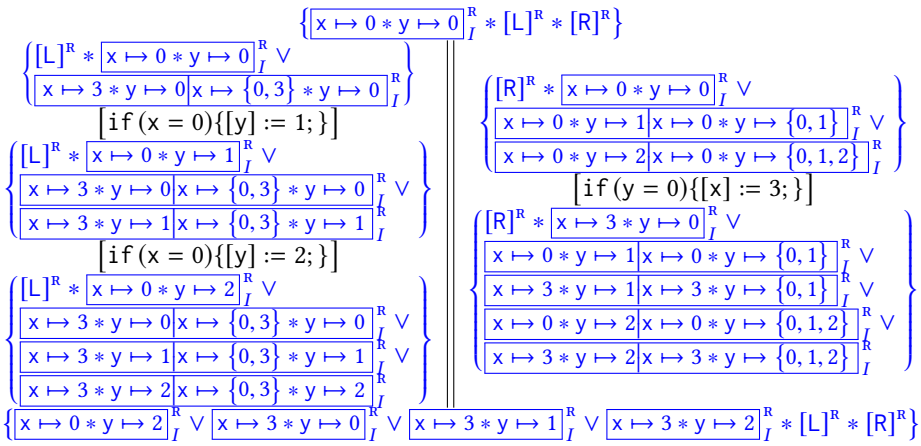


Here the interference is also valid under parallel snapshot isolation. But if we use the purple interference that strictly satisfies snapshot isolation, the post-condition in purple will be eliminated.

5.7 An example for SI

$$\boxed{x \mapsto - * y \mapsto - \mid x \mapsto - * y \mapsto -}^{R}_{I}$$

$$\begin{aligned} I: \quad & L : x \mapsto \{0,3\} * y \mapsto - \mathbin{@} \{(R, x, 0), (W, y, 1)\} \\ & L : x \mapsto \{0,3\} * y \mapsto - \mathbin{@} \{(R, x, 0), (W, y, 2)\} \\ & R : x \mapsto - * y \mapsto \{0,1,2\} \mathbin{@} \{(R, y, 0), (W, x, 3)\} \\ & Rd : \ldots \end{aligned}$$


$\{[x \mapsto 0 * y \mapsto 0]^R_I * [L]^R * [R]^R\}$
$\{[L]^R * [x \mapsto 0 * y \mapsto 0]^R_I \vee [x \mapsto 3 * y \mapsto 0 | x \mapsto \{0,3\} * y \mapsto 0]^R_I\}$
$[\text{if}\,(x = 0)\{[y] := 1;\}]$
$\{[L]^R * [x \mapsto 0 * y \mapsto 1]^R_I \vee [x \mapsto 3 * y \mapsto 0 | x \mapsto \{0,3\} * y \mapsto 0]^R_I \vee [x \mapsto 3 * y \mapsto 1 | x \mapsto \{0,3\} * y \mapsto 1]^R_I\}$
$[\text{if}\,(x = 0)\{[y] := 2;\}]$
$\{[L]^R * [x \mapsto 0 * y \mapsto 2]^R_I \vee [x \mapsto 3 * y \mapsto 0 | x \mapsto \{0,3\} * y \mapsto 0]^R_I \vee [x \mapsto 3 * y \mapsto 1 | x \mapsto \{0,3\} * y \mapsto 1]^R_I \vee [x \mapsto 3 * y \mapsto 2 | x \mapsto \{0,3\} * y \mapsto 2]^R_I\}$
$\{[R]^R * [x \mapsto 0 * y \mapsto 0]^R_I \vee [x \mapsto 0 * y \mapsto 1 | x \mapsto 0 * y \mapsto \{0,1\}]^R_I \vee [x \mapsto 0 * y \mapsto 2 | x \mapsto 0 * y \mapsto \{0,1,2\}]^R_I\}$
$[\text{if}\,(y = 0)\{[x] := 3;\}]$
$\{[R]^R * [x \mapsto 3 * y \mapsto 0]^R_I \vee [x \mapsto 0 * y \mapsto 1 | x \mapsto 0 * y \mapsto \{0,1\}]^R_I \vee [x \mapsto 3 * y \mapsto 1 | x \mapsto 3 * y \mapsto \{0,1\}]^R_I \vee [x \mapsto 0 * y \mapsto 2 | x \mapsto 0 * y \mapsto \{0,1,2\}]^R_I \vee [x \mapsto 3 * y \mapsto 2 | x \mapsto 3 * y \mapsto \{0,1,2\}]^R_I\}$
$\{[x \mapsto 0 * y \mapsto 2]^R_I \vee [x \mapsto 3 * y \mapsto 0]^R_I \vee [x \mapsto 3 * y \mapsto 1]^R_I \vee [x \mapsto 3 * y \mapsto 2]^R_I * [L]^R * [R]^R\}$

REFERENCES

The notation $f[k \mapsto v]$ on a function f means the result by replacing or extending the element associated with k to v. The notation $l[i \mapsto k]$ on a list l means the result by replacing the *i-th* element to k. The $+\!\!+$ denotes list concatenation.

$$\begin{aligned}
\mathsf{updateMKVS}\,(kv, u, t, \mathbf{1}_{\mathcal{F}}) &\triangleq kv \\
\mathsf{updateMKVS}\,\big(kv, u, t, \mathcal{F} \uplus \{(\mathtt{R}, k, -)\}\big) &\triangleq \mathtt{let}\ (n, t', \mathcal{T}) = kv(k, u(k)) \\
&\quad\ \mathtt{and}\ kv' = kv\big[k \mapsto kv(k)\big[u(k) \mapsto (n, t', \mathcal{T} \uplus \{t\})\big]\big] \\
&\quad\ \mathtt{in}\ \mathsf{updateMKVS}\,(kv', u, t, \mathcal{F}) \\
\mathsf{updateMKVS}\,\big(kv, u, t, \mathcal{F} \uplus \{(\mathtt{W}, k, n)\}\big) &\triangleq \mathtt{let}\ kv' = kv[k \mapsto (kv(k) +\!\!+ [(n, t, \emptyset)])] \\
&\quad\ \mathtt{in}\ \mathsf{updateMKVS}\,(kv', u, t, \mathcal{F}) \\
\mathsf{updateView}\,(kv, u, \mathbf{1}_{\mathcal{F}}) &\triangleq u \\
\mathsf{updateView}\,\big(kv, u, \mathcal{F} \uplus \{(\mathtt{R}, k, -)\}\big) &\triangleq \mathsf{updateView}\,(kv, u, \mathcal{F}) \\
\mathsf{updateView}\,\big(kv, u, \mathcal{F} \uplus \{(\mathtt{W}, k, -)\}\big) &\triangleq \mathsf{updateView}\,(kv, u[k \mapsto (|kv(k)| - 1)], \mathcal{F}) \\
\mathsf{fresh}\,(kv) &\triangleq \{t \mid t \in \textsc{TransID} \wedge \forall k, i.\, t \neq \mathsf{Write}(kv(k, i)) \wedge t \notin \mathsf{Reads}(\mathcal{H}^{\mathsf{r}}(k, i))\}
\end{aligned}$$

The function snapshot is defined in Theorem 2.4 and operational semantics for transactions $\rightsquigarrow$ in Fig. 2. Given the set of executions tests ET (Theorem 2.9), stacks Stacks (Theorem 2.6), multi-version key-value stores MKVSs (Theorem 2.1) and views Views (Theorem 2.2), the *operational semantics for commands*:

$$\rightarrow : ((\text{MKVSs} \times \textsc{Stacks} \times \textsc{Views}) \times \textsc{Cmd}) \times \text{ET} \times ((\text{MKVSs} \times \textsc{Stacks} \times \textsc{Views}) \times \textsc{Cmd})$$

PCommit

$$\frac{u'' \geq u \qquad t \in \mathsf{fresh}\,(kv) \qquad ss = \mathsf{snapshot}(kv, u'') \qquad (s, ss, \mathbf{1}_{\mathcal{F}}), \mathtt{T} \rightsquigarrow^{*} (s', ss', \mathcal{F}), \mathtt{skip} \\ kv' = \mathsf{updateMKVS}\,(kv, u'', t, \mathcal{F}) \qquad u' \geq \mathsf{updateView}\,(kv', u'', \mathcal{F}) \qquad (kv, u) \triangleright_{\text{ET}} \mathcal{F} : u'}{(kv, s, u), \big[\mathtt{T}\big] \xrightarrow{\text{ET}} (kv', s', u'), \mathtt{skip}}$$

PAssign

$$\frac{v = [\![\mathtt{E}]\!]_s}{(kv, s, u), \mathtt{x} := \mathtt{E} \xrightarrow{\text{ET}} (kv, s[\mathtt{x} \mapsto v], u), \mathtt{skip}}$$

PAssume

$$\frac{[\![\mathtt{E}]\!]_\sigma \neq 0}{(kv, s, u), \mathtt{assume(E)} \xrightarrow{\text{ET}} (kv, s, u), \mathtt{skip}}$$

PChoice

$$\frac{i \in \{1, 2\}}{(kv, s, u), \mathtt{C}_1 + \mathtt{C}_2 \xrightarrow{\text{ET}} (kv, s, u), \mathtt{C}_i}$$

PIter

$$\frac{}{(kv, s, u), \mathtt{C}^* \xrightarrow{\text{ET}} (kv, s, u), \mathtt{skip} + (\mathtt{C}\,;\mathtt{C}^*)}$$

PSeqSkip

$$\frac{}{(kv, s, u), \mathtt{skip}\,;\mathtt{C} \xrightarrow{\text{ET}} (kv, s, u), \mathtt{C}}$$

PSeq

$$\frac{(kv, s, u), \mathtt{C}_1 \xrightarrow{\text{ET}} (kv, s', u'), \mathtt{C}_1'}{(kv, s, u), \mathtt{C}_1\,;\mathtt{C}_2 \xrightarrow{\text{ET}} (kv, s', u'), \mathtt{C}_1'\,;\mathtt{C}_2}$$

The thread environment is a partial function from thread identifiers to pairs of stacks and views $Env \in \textsc{EnvEnv} \triangleq \textsc{ClientID} \xrightarrow{fin} \textsc{Stacks} \times \textsc{Views}$. Given the set of execution tests ET (Theorem 2.9) and key-value stores MKVSs (Theorem 2.1), the *semantics for programs*:

$$\Rightarrow_{\text{ET}} : (\text{MKVSs} \times \textsc{EnvEnv} \times \textsc{Prog}) \times \text{ET} \times (\text{MKVSs} \times \textsc{EnvEnv} \times \textsc{Prog})$$

PSingleThread

$$\frac{(s, u) = Env(\mathrm{cl}) \qquad \mathtt{C} = \mathtt{P}(\mathrm{cl}) \qquad (kv, s, u), \mathtt{C}, \xrightarrow{\text{ET}} (s', kv', u'), \mathtt{C}'}{(kv, Env, \mathtt{P}) \Rightarrow_{\text{ET}} (kv', Env[\mathrm{cl} \mapsto (s', u')], \mathtt{P}[\mathrm{cl} \mapsto \mathtt{C}'])}$$

Fig. 3. operational semantics for threads and programs

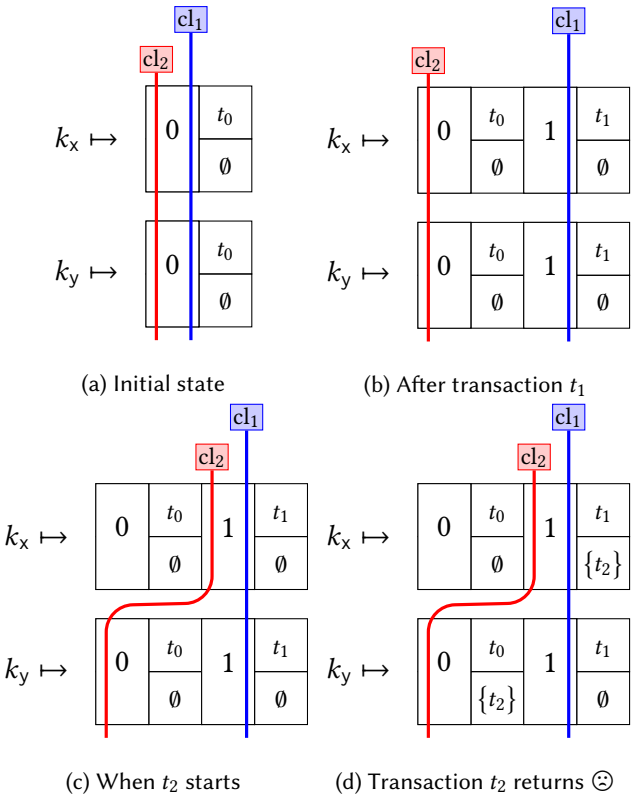


Fig. 4. Graphical Representation of configurations obtained through the execution of P_1.

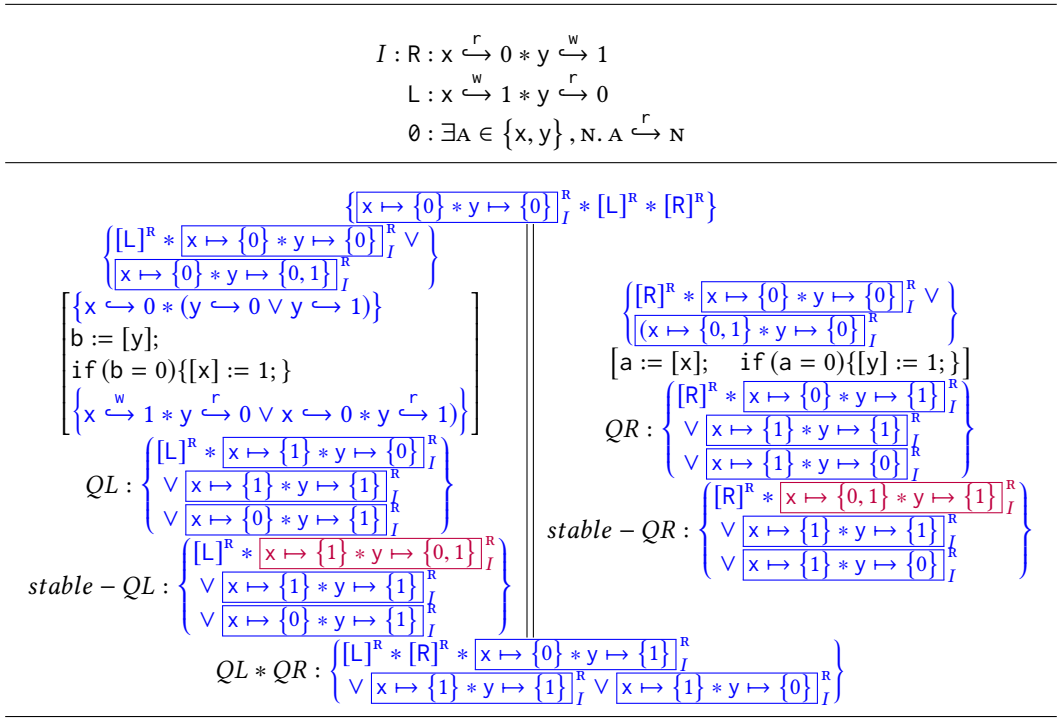


Fig. 5. write skew under snapshot isolation

SX: Font for E

$$\textsc{TRLookup}\ \frac{\mathtt{x} \notin \mathtt{fv}(\mathtt{E}) \qquad p \xrightarrow{\mathtt{R}(\mathtt{E},\mathtt{E})} q}{\vdash_I \{p\}\ \mathtt{x} := [\mathtt{E}]\ \{\mathtt{x} \doteq \mathtt{E} * q[^{\mathtt{E}}/_{\mathtt{x}}]\}} \qquad \textsc{TRMutate}\ \frac{p \xrightarrow{\mathtt{W}(\mathtt{E}_1,\mathtt{E}_2)} q}{\vdash_I \{p\}\ [\mathtt{E}_1] := \mathtt{E}_2\ \{q\}} \qquad \textsc{TRAss}\ \frac{\mathtt{x} \notin \mathtt{fv}(\mathtt{E})}{\vdash_I \{\mathtt{x} \doteq \mathtt{E}\}\ \mathtt{x} := \mathtt{E}\ \{\mathtt{x} \doteq \mathtt{E}[^{\mathtt{E}}/_{\mathtt{x}}]\}}$$

$$\textsc{TRAssume}\ \frac{}{\vdash_I \{\mathtt{E} \dot{\neq} 0\}\ \mathtt{assume(E)}\ \{\mathtt{E} \dot{\neq} 0\}} \qquad \textsc{TRChoice}\ \frac{\vdash_I \{p\}\ \mathtt{T}_1\ \{q\} \qquad \vdash_I \{p\}\ \mathtt{T}_2\ \{q\}}{\vdash_I \{p\}\ \mathtt{T}_1 + \mathtt{T}_2\ \{q\}}$$

$$\textsc{TRSeq}\ \frac{\vdash_I \{p\}\ \mathtt{T}_1\ \{r\} \qquad \vdash_I \{r\}\ \mathtt{T}_2\ \{q\}}{\vdash_I \{p\}\ \mathtt{T}_1\,;\mathtt{T}_2\ \{q\}} \qquad \textsc{TRIter}\ \frac{\vdash_I \{p\}\ \mathtt{T}\ \{p\}}{\vdash_I \{p\}\ \mathtt{T}^*\ \{p\}} \qquad \textsc{TRFrame}\ \frac{\vdash_I \{p\}\ \mathtt{T}\ \{q\}}{\vdash_I \{p * r\}\ \mathtt{T}\ \{q * r\}}$$

$$\textsc{TRSkip}\ \frac{}{\vdash_I \{\mathtt{emp}\}\ \mathtt{skip}\ \{\mathtt{emp}\}}$$

$$\mathtt{E} \hookrightarrow \mathtt{E}' \xrightarrow{\mathtt{R}(\mathtt{E},\mathtt{E}')} \mathtt{E} \overset{\mathtt{r}}{\hookrightarrow} \mathtt{E}' \qquad \mathtt{E} \hookrightarrow \mathtt{E}' \xrightarrow{\mathtt{W}(\mathtt{E},\mathtt{E}'')} \mathtt{E} \overset{\mathtt{w}}{\hookrightarrow} \mathtt{E}''$$

$$\mathtt{E} \overset{\mathtt{r}}{\hookrightarrow} \mathtt{E}' \xrightarrow{\mathtt{R}(\mathtt{E},\mathtt{E}')} \mathtt{E} \overset{\mathtt{r}}{\hookrightarrow} \mathtt{E}' \qquad \mathtt{E} \overset{\mathtt{r}}{\hookrightarrow} \mathtt{E}' \xrightarrow{\mathtt{W}(\mathtt{E},\mathtt{E}'')} \mathtt{E} \overset{\mathtt{rw}}{\hookrightarrow} (\mathtt{E}', \mathtt{E}'')$$

$$\mathtt{E} \overset{\mathtt{w}}{\hookrightarrow} \mathtt{E}' \xrightarrow{\mathtt{R}(\mathtt{E},\mathtt{E}')} \mathtt{E} \overset{\mathtt{w}}{\hookrightarrow} \mathtt{E}' \qquad \mathtt{E} \overset{\mathtt{w}}{\hookrightarrow} \mathtt{E}' \xrightarrow{\mathtt{W}(\mathtt{E},\mathtt{E}'')} \mathtt{E} \overset{\mathtt{w}}{\hookrightarrow} \mathtt{E}''$$

$$\mathtt{E} \overset{\mathtt{rw}}{\hookrightarrow} (\mathtt{E}'', \mathtt{E}') \xrightarrow{\mathtt{R}(\mathtt{E},\mathtt{E}')} \mathtt{E} \overset{\mathtt{rw}}{\hookrightarrow} (\mathtt{E}'', \mathtt{E}') \qquad \mathtt{E} \overset{\mathtt{rw}}{\hookrightarrow} (\mathtt{E}'', \mathtt{E}') \xrightarrow{\mathtt{W}(\mathtt{E},\mathtt{E}''')} \mathtt{E} \overset{\mathtt{rw}}{\hookrightarrow} (\mathtt{E}'', \mathtt{E}''')$$

Fig. 6. The rules for transactions

$$\textsc{PRCommit}\ \frac{\vdash_I \{p\}\ \mathsf{T}\ \{q\} \qquad \mathsf{ET} \vdash P \Rrightarrow^{\{p\}\{q\}} Q \qquad \mathsf{stable}(P, \mathsf{ET}) \qquad \mathsf{stable}(Q, \mathsf{ET})}{\vdash_g \{P\}\ [\mathsf{T}]\ \{Q\}}$$

$$\textsc{TRPar}\ \frac{\vdash_g \{P_1\}\ \mathsf{C}_1\ \{Q_1\} \qquad \vdash_g \{P_2\}\ \mathsf{C}_2\ \{Q_2\} \qquad \mathsf{stable}(P_1, \mathsf{ET}) \qquad \mathsf{stable}(P_2, \mathsf{ET})}{\vdash_g \{P_1 * P_2\}\ \mathsf{C}_1 \parallel \mathsf{C}_2\ \{Q_1 * Q_2\}}$$

$$\textsc{PRAss}\ \frac{\mathsf{x} \notin \mathsf{fv}(\mathsf{E})}{\vdash_g \{\mathsf{x} \doteq \mathsf{E}\}\ \mathsf{x} := \mathsf{E}\ \{\mathsf{x} \doteq \mathsf{E}[^{\mathsf{E}}/_{\mathsf{x}}]\}}$$

$$\textsc{PRAssume}\ \frac{}{\vdash_g \{\mathsf{E} \dot{\neq} 0\}\ \mathsf{assume}(\mathsf{E})\ \{\mathsf{E} \dot{\neq} 0\}}$$

$$\textsc{PRChoice}\ \frac{\vdash_g \{P\}\ \mathsf{C}_1\ \{Q\} \qquad \vdash_g \{P\}\ \mathsf{C}_2\ \{Q\}}{\vdash_g \{P\}\ \mathsf{C}_1 + \mathsf{C}_2\ \{Q\}}$$

$$\textsc{TRSeq}\ \frac{\vdash_g \{P\}\ \mathsf{C}_1\ \{R\} \qquad \vdash_g \{R\}\ \mathsf{C}_2\ \{Q\}}{\vdash_g \{P\}\ \mathsf{C}_1 ; \mathsf{C}_2\ \{Q\}}$$

$$\textsc{TRIter}\ \frac{\vdash_g \{P\}\ \mathsf{C}\ \{P\}}{\vdash_g \{P\}\ \mathsf{C}^*\ \{P\}}$$

$$\textsc{TRFrame}\ \frac{\vdash_g \{P\}\ \mathsf{C}\ \{Q\} \qquad \mathsf{stable}(R, \mathsf{ET})}{\vdash_g \{P * R\}\ \mathsf{C}\ \{Q * R\}}$$

$$\begin{aligned}\mathsf{ET} \vdash P \Rrightarrow^{\{p\}\{q\}} Q \triangleq\ & \forall w, kv, u, \iota, s.\ w \in [\![P]\!]_{\iota,s} \wedge (kv, u) \in \lfloor\!| w |\!\rfloor \implies \\ & \exists ss.\ ss = \mathsf{snapshot}(kv, u) \wedge (ss, \mathbf{1}_{\mathcal{T}}) \in [\![p]\!]_{\iota,s} \\ & \quad \wedge \forall s', t, \mathcal{F}, kv', u'.\ \exists w'.\ t \in \mathsf{fresh}(kv) \wedge (-, \mathcal{F}) \in [\![q]\!]_{\iota,s'} \\ & \quad\quad \wedge kv' = \mathsf{UpdateMKVS}(kv, u, t, \mathcal{F}) \wedge u' \geq \mathsf{UpdateView}(kv, u, \mathcal{F}) \\ & \quad\quad \wedge (w, w') \in \mathcal{G} \wedge (kv, u) \triangleright_{\mathsf{ET}} \mathcal{F} : u' \wedge (kv', u') \in \lfloor\!| w' |\!\rfloor \wedge w' \in [\![Q]\!]_{\iota,s'}\end{aligned}$$

Fig. 7. The rules for programs

$$\frac{}{\vdash_f \{\mathsf{E} \mapsto \mathsf{E}\}\ \mathsf{E} \hookrightarrow \mathsf{E}\ \{\mathsf{E} \mapsto \mathsf{E}\}}\ \textsc{FInit}$$

$$\frac{}{\vdash_f \{\mathsf{E} \mapsto \mathsf{E}\}\ \mathsf{E} \overset{\mathsf{r}}{\hookrightarrow} \mathsf{E}\ \{\mathsf{E} \mapsto \mathsf{E}\}}\ \textsc{FRead}$$

$$\frac{}{\vdash_f \{\mathsf{E} \mapsto \mathsf{E}\}\ \mathsf{E} \overset{\mathsf{w}}{\hookrightarrow} \mathsf{E}'\ \{\mathsf{E} \mapsto \mathsf{E}'\}}\ \textsc{FWrite}$$

$$\frac{}{\vdash_f \{\mathsf{E} \mapsto \mathsf{E}\}\ \mathsf{E} \overset{\mathsf{rw}}{\hookrightarrow} (\mathsf{E}, \mathsf{E}')\ \{\mathsf{E} \mapsto \mathsf{E}'\}}\ \textsc{FReWrt}$$

$$\frac{\vdash_f \{\bar{p}_1\}\ \bar{f}_1\ \{\bar{q}_1\} \qquad \vdash_f \{\bar{p}_2\}\ \bar{f}_2\ \{\bar{q}_2\}}{\vdash_f \{\bar{p}_1 * \bar{p}_2\}\ \bar{f}_1 * \bar{f}_2\ \{\bar{q}_1 * \bar{q}_2\}}\ \textsc{FFrame}$$

Fig. 8. Syntactic rule for UpdateMKVS and UpdateView functions