A Lattice-Theoretical Approach to Deterministic Parallelism with Shared State

Lindsey Kuper Ryan R. Newton

October 2012*

Abstract

We present a new model for deterministic-by-construction parallel programming that generalizes existing single-assignment models to allow multiple assignments that are monotonically increasing with respect to a user-specified partial order. Our model achieves determinism by using a novel shared data structure with an API that allows only monotonic writes and "threshold" reads that block until a lower bound is reached. We give a proof of determinism for our model, discuss ways to express existing deterministic parallel models using it, and describe how to extend it to support a limited form of nondeterminism that admits failures but never wrong answers.

1 Introduction

Programs written using a deterministic-by-construction model of parallel computation always produce the same observable results, offering programmers the promise of freedom from subtle, hard-to-reproduce nondeterministic bugs. A common theme that emerges in the study of diverse deterministic-by-construction parallel systems, from venerable models like Kahn process networks (KPNs) [15] to modern ones like Intel's Concurrent Collections (CnC) system [7], is that the determinism of the model hinges on some notion of *monotonicity*. In KPNs, for instance, processes communicate over FIFO channels with ever-increasing channel histories, while in CnC, a shared data store of single-assignment variables grows monotonically.

Because state modifications that only add information and never destroy it can be structured to commute with one another (and thereby avoid insidious race conditions), it stands to reason that monotonic data structures play a key role in the design of deterministic-by-construction parallel programming models. Yet there is little in the way of a general theory of monotonic data structures as a basis for deterministic, shared-state concurrency. As a result, models like CnC and KPNs emerge independently, without recognition of their common basis. In this paper we take a step towards a more general theory.

We begin with an example. Consider the program in Figure 1(a), written in a hypothetical programming language with locations, standard get and put operations on locations, and a let par form for parallel evaluation of multiple subexpressions. Depending on whether get l or put l 4 executes first, the value of v might be either 3 or 4. Hence Figure 1(a) is nondeterministic: multiple runs of the program can produce different observable results based on choices made by the scheduler.

A straightforward modification we can make to our hypothetical language to enforce determinism is to require that variables may be written to at most once, resulting in a *single-assignment* language [24]. Such single-assignment variables are sometimes known as *IVars*[1] and are a well-established mechanism for enforcing determinism at the language and library level [8, 26, 7, 18] and even at the hardware level [5]. In a language with IVars, the second call to put in Figure 1(a) would raise an error, and the resulting program, since it would always produce the error, would be deterministic.

IVars enforce determinism by restricting the *writes* that can occur to a variable. However, the single-write restriction can be weakened as long as *reads* are also restricted. In Figure 1(b), we modify get to take an extra argument,

*Revises the previous version dated July 2012.

[1]IVars are so named because they are a special case of *I-structures* [3]—namely, those with only one cell.

(a)

```
let _ = put l 3 in
  let par v = get l
          _ = put l 4
    in v
```

(b)

```
let _ = put l 3 in
  let par v = get l 4
          _ = put l 4
    in v
```

(c)

```
let _ = put l 3 in
  let par v = get l 4
          _ = put l 4
          _ = put l 5
    in v
```

Figure 1: Three example programs: (a) nondeterministic, (b) deterministic with a threshold read, and (c) deterministic with a threshold read that returns the specified threshold value.

representing the *minimum value* that we are interested in reading from v. If the value of l has not yet reached 4 at the time that get l 4 is ready to run, the operation *blocks* until it does, giving put l 4 an opportunity to run first. Assuming (as we do) that the scheduler will eventually decide to run both branches of the let par expression, Figure 1(b) is deterministic and will always evaluate to 4. Moreover, if we had written get l 5 instead of get l 4, the program would be guaranteed to block forever.

Our tweak fixes the specific program in Figure 1(b). But what if multiple subcomputations are writing to l in parallel, all with values greater than or equal to four? Competing puts land us back where we started—Figure 1(c) might evaluate to either 4 or 5 without further restrictions. Therefore we propose a design in which, if a minimum or "threshold" value specified by a get operation has been reached, then the get operation returns *that minimum value*. This get restriction is not as draconian as it may seem; later we will see how the total order in these examples can be relaxed to a partial order, and potentially infinite *sets* of threshold values may be specified. Together, monotonically increasing puts and minimum-value gets yield a *deterministic-by-construction* model, guaranteeing that every program written using the model will behave deterministically.

Our proposed model generalizes IVars to *LVars*, thus named because their states can be represented as elements of a user-specified partially ordered set that forms a *bounded join-semilattice*. This user-specified partially ordered set, which we call a *domain*, determines the semantics of the put and get operations that comprise the interface to LVars. In Figure 1(c), for instance, the domain that determines the semantics of put and get might be the natural numbers ordered by $\leq$. The LVar model is general enough to subsume the IVar model—as well as other deterministic parallel models—because it is parameterized by the choice of domain. For example, a domain of channel histories with a prefix ordering would allow LVars to become FIFO channels that implement a Kahn process network. Different instantiations of the domain result in a family of parallel languages, all of which are deterministic. This family of languages is exactly the class of languages that deal with *asynchronous, data-driven* parallelism [19], which is critical for irregular parallel applications such as graph algorithms.

An example application that uses rich, shared data structures and that processes irregular data is Hindley-Milner type inference. In a parallelized type-inference algorithm, each type variable becomes an LVar, and upward movement in the lattice represents type unification. Another example is the problem of removing duplicates from a list in parallel. One solution is for multiple computations to insert elements into a single, shared *set* data structure, with a domain ordered by subset inclusion.

Monotonically increasing variables naturally lend themselves to a variety of parallel operations on data structures in a way that single-assignment variables do not. For instance, in the duplicate-removal example, the shared set might be represented by a *trie*. Consider then inserting two keys, say, `0111` and `1111`, into the trie from different points in the parallel computation. Supposing that `0` represents "left" and `1` "right", there would seem to be no conflict—the two operations are filling in disjoint parts of the data structure. However, if the trie were implemented with IVars, each operation would need to fill in a *chain* of IVars, populating the tree from the root to the leaf in question. To retain determinism, IVars do not allow testing for emptiness, so there would be no way for one put operation to know if another had already populated the root of the trie. Moreover, if both operations attempted to create a new node and then insert it into the IVar at the root of the trie, then we would cause a violation of the single-assignment rule. This is a limitation of IVars that LVars solve.

Contributions In this paper, we introduce LVars as the building block of a model of deterministic parallelism (Section 2) and use them to define λ_{LVar}, a parallel calculus with shared state based on the call-by-value λ-calculus (Section 3). As our main technical result, we present a proof of determinism for λ_{LVar} (Section 4). A critical aspect of the proof is a frame-rule-like property, expressed by the Independence lemma (Section 4.3), that would *not* hold in a typical language with shared mutable state, but holds in our setting because of the semantics of LVars and their put/get interface. We present evidence that λ_{LVar} is sufficiently expressive to model two paradigms of deterministic parallel computation: shared-state, single-assignment models, exemplified by the Intel Concurrent Collections framework [7] and the `monad-par` Haskell library [18], and data-flow networks, exemplified by Kahn process networks [15] (Section 5). Finally, we describe an extension to the basic λ_{LVar} model: destructive observations, enabling a limited form of nondeterminism that admits failures but not wrong answers (Section 6).

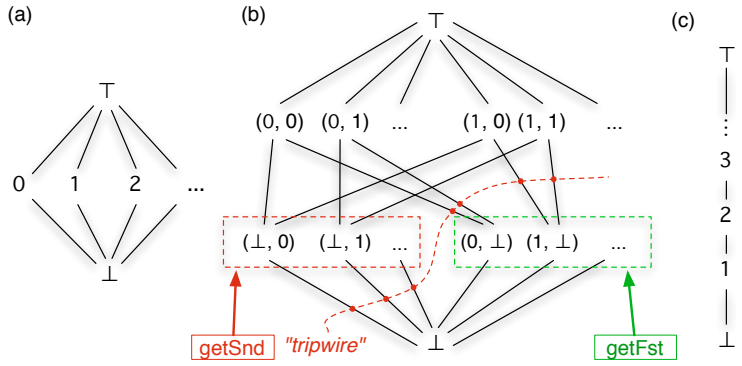

Figure 2: Example domains: (a) IVar containing a natural number; (b) pair of natural-number-valued IVars; (c) $\leq$ ordering. Subfigure (b) is annotated with example threshold sets that would correspond to a blocking read of the first or second element of the pair (see Sections 2.3 and 3.2). Any state transition crossing the "tripwire" for getSnd causes it to unblock and return a result.

2 Domains, Stores, and Determinism

We take as the starting point for our work a call-by-value λ-calculus extended with a *store* and with communication primitives put and get that operate on data in the store. We call this language λ_{LVar}. The class of programs that we are interested in modeling with λ_{LVar} are those with explicit effectful operations on shared data structures, in which subcomputations may communicate with each other via the put and get operations.

In this setting of shared mutable state, the trick that λ_{LVar} employs to maintain determinism is that stores contain *LVars*, which are a generalization of IVars [3]. Whereas IVars are single-assignment variables—either empty or filled with an immutable value—an LVar may have an arbitrary number of states forming a *domain* (or *state space*) D, which is partially ordered by a relation $\sqsubseteq$. An LVar can take on any sequence of states from the domain D, so long as that sequence respects the partial order—that is, updates to the LVar (made via the put operation) are *inflationary* with respect to $\sqsubseteq$. Moreover, the interface presented by the get operation allows only limited observations of the LVar's state. In this section, we discuss how domains and stores work in λ_{LVar} and explain how the semantics of put and get together enforce determinism in λ_{LVar} programs.

2.1 Domains

The definition of λ_{LVar} is parameterized by the choice of a *domain* D: to write concrete λ_{LVar} programs, one must specify the domain that one is interested in working with. Therefore λ_{LVar} is actually a *family* of languages, rather than a single language. Virtually any data structure to which information is added gradually can be represented as a

λ_{LVar} domain, including pairs, arrays, trees, maps, and infinite streams. Figure 2 gives three examples of domains for common data structures.

Formally, a domain D is a *bounded join-semilattice*.[2] In other words:

- D comes equipped with a partial order $\sqsubseteq$;
- every pair of elements in D has a least upper bound (lub) $\sqcup$;
- D has a least element $\bot$ and a greatest element $\top$.

The simplest example of a useful domain is one that represents the state space of a single-assignment variable (an IVar). A natural-number-valued IVar, for instance, would correspond to the domain in Figure 2(a), that is,

$$D = (\{\top, \bot\} \cup \mathbb{N}, \sqsubseteq),$$

where the partial order $\sqsubseteq$ is defined by setting $\bot \sqsubseteq d \sqsubseteq \top$ and $d \sqsubseteq d$ for all $d \in D$. This is a lattice of height three and infinite width, where the naturals are arranged horizontally. After the initial write of some $n \in \mathbb{N}$, any further conflicting writes would push the state of the IVar to $\top$ (an error).

The motivation for requiring domains with the given structure is as follows:

- the least element, $\bot$, is needed to initialize store locations;
- the greatest element, $\top$, is needed to denote "conflicting" updates to store locations;
- the requirement that every two elements must have a lub means that it is always possible to fork a computation into subcomputations that can independently update the store and then join the results by taking the lub of updates to shared locations.

2.2 Stores

During the evaluation of a λ_{LVar} program, a *store* S keeps track of the states of LVars. Each LVar is represented by a binding from a location l, drawn from a set Loc, to its state, which is some element $d \in D$. Although each LVar in a program has its own state, the states of all the LVars are drawn from the same domain D. We can do this with no loss of generality because lattices corresponding to different types of LVars could always be unioned into a single lattice (with shared $\top$ and $\bot$ elements). Alternatively, in a typed formulation of λ_{LVar}, the type of an LVar might determine the domain of its states.

Definition 1. A *store* is either a finite partial mapping $S : Loc \xrightarrow{\text{fin}} (D - \{\top\})$, or the distinguished element $\top_S$.

We use the notation $S[l \mapsto d]$ to denote extending S with a binding from l to d. If $l \in dom(S)$, then $S[l \mapsto d]$ denotes an update to the existing binding for l, rather than an extension. We can also denote a store by explicitly writing out all its bindings, using the notation $[l_1 \mapsto d_1, l_2 \mapsto d_2, \ldots]$. The state space of stores forms a bounded join-semilattice, just as D does. The least element $\bot_S$ is the empty store, and $\top_S$ is the greatest element. It is straightforward to lift the $\sqsubseteq$ and $\sqcup$ operations defined on elements of D to the level of stores:

Definition 2. A store S is *less than or equal to* a store S' (written $S \sqsubseteq_S S'$) iff:

- $S' = \top_S$, or
- $dom(S) \subseteq dom(S')$ and for all $l \in dom(S)$, $S(l) \sqsubseteq S'(l)$.

Definition 3. The *least upper bound (lub)* of two stores S_1 and S_2 (written $S_1 \sqcup_S S_2$) is defined as follows:

- $S_1 \sqcup_S S_2 = \top_S$ iff there exists some $l \in dom(S_1) \cap dom(S_2)$ such that $S_1(l) \sqcup S_2(l) = \top$.

[2] Although we will sometimes abbreviate "bounded join-semilattice" to "lattice" for brevity's sake in the discussion that follows, λ_{LVar} domains do not, in general, satisfy the properties of a lattice.

- Otherwise, $S_1 \sqcup_S S_2$ is the store S such that:
  - $dom(S) = dom(S_1) \cup dom(S_2)$, and
  - For all $l \in dom(S)$:

$$S(l) = \begin{cases} S_1(l) \sqcup S_2(l) & \text{if } l \in dom(S_1) \cap dom(S_2) \\ S_1(l) & \text{if } l \notin dom(S_2) \\ S_2(l) & \text{if } l \notin dom(S_1) \end{cases}$$

By Definition 3, if $d_1 \sqcup d_2 = \top$, then $[l \mapsto d_1] \sqcup_S [l \mapsto d_2] = \top_S$. Notice that a store like $[l \mapsto \top]$ can never arise during the execution of a λ_{LVar} program, because (as we will see in Section 3) an attempted write that would take the state of l to $\top$ would raise an error before the write can occur.

2.3 Communication Primitives

The new, put, and get operations create, write to, and read from LVars, respectively. The interface is similar to that presented by mutable references:

- new extends the store with a binding for a new LVar whose initial state is $\bot$, and returns the location l of that LVar (*i.e.*, a pointer to the LVar).
- put takes a pointer to an LVar and a singleton set containing a new state; it updates the store, merging the current state of the LVar with the new state by taking their lub, and pushes the state of the LVar upward in the lattice. Any update that would take the state of an LVar to $\top$ results in an error.
- get performs a blocking "threshold" read that allows limited observations of the state of an LVar. It takes a pointer to an LVar and a *threshold set* Q, which is a non-empty subset of D that is *pairwise incompatible*, meaning that the lub of any two distinct elements in Q is $\top$. If the LVar's state d in the lattice is *at or above* some $d' \in Q$, the get operation unblocks and returns the singleton set $\{d'\}$. Note that d' is a unique element of Q, for if there is another $d'' \neq d'$ in the threshold set such that $d'' \sqsubseteq d$, it would follow that $d' \sqcup d'' = d \neq \top$, which contradicts the requirement that Q be pairwise incompatible.

The intuition behind get is that it specifies a subset of the lattice that is "horizontal": no two elements in the subset can be above or below one another. Intuitively, each element in the threshold set is an "alarm" that detects the activation of itself or any state above it. One way of visualizing the threshold set for a get operation is as a subset of edges in the lattice that, if crossed, set off the corresponding alarm. Together these edges form a "tripwire". This visualization is pictured in Figure 2(b). The threshold set $\{(\bot, 0), (\bot, 1), ...\}$ (or a subset thereof) would pass the incompatibility test, as would the threshold set $\{(0, \bot), (1, \bot), ...\}$ (or a subset thereof), but a combination of the two would not pass.

Both get and put take and return *sets*. The fact that put takes a singleton set and get returns a singleton set (rather than a value d) may seem awkward; it is merely a way to keep the grammar for values simple, and avoid including set primitives in the language (*e.g.*, for converting d to $\{d\}$).

2.4 Monotonic Store Growth and Determinism

In IVar-based languages, a store can only change in one of two ways: a new binding is added at $\bot$, or a previously $\bot$ binding is permanently updated to a meaningful value. It is therefore straightforward in such languages to define an ordering on stores and establish determinism based on the fact that stores grow monotonically with respect to the ordering. For instance, *Featherweight CnC* [7], a lightweight, single-assignment imperative language that models the CnC system, defines ordering on stores as follows:[3]

Definition 4 (store ordering, *Featherweight CnC*)**.** A store S is *less than or equal to* a store S' (written $S \sqsubseteq_S S'$) iff $dom(S) \subseteq dom(S')$ and for all $l \in dom(S)$, $S(l) = S'(l)$.

[3] In *Featherweight CnC*, the store interface is simpler still: no store location is ever bound to $\bot$. Instead, if $l \notin dom(S)$ then l is defined to be at $\bot$, and a location springs into existence at the time that its permanent value is written.

Given a domain D with elements $d \in D$:

$$
\begin{array}{rrcl}
\text{configurations} & \sigma & ::= & \langle S;\, e\rangle \mid \textbf{error} \\
\text{expressions} & e & ::= & x \mid v \mid e\ e \mid \mathsf{new} \mid \mathsf{put}\ e\ e \mid \mathsf{get}\ e\ e \mid \mathsf{convert}\ e \\
\text{values} & v & ::= & l \mid Q \mid \lambda x.\, e \\
\text{threshold set literals} & Q & ::= & \{d_1, d_2, \ldots, d_n\} \mid \{d \mid pred(d)\} \\
 & & & \text{(where } pred(d) \text{ is computable)} \\
\text{stores} & S & ::= & \top_S \mid [l_1 \mapsto d_1, l_2 \mapsto d_2, \ldots] \\
 & & & \text{(where } d_i \neq \top\text{)}
\end{array}
$$

Figure 3: Syntax for λ_{LVar}.

Our Definition 2 is reminiscent of Definition 4, but Definition 4 requires that $S(l)$ and $S'(l)$ be *equal*, instead of our weaker requirement that $S(l) \sqsubseteq S'(l)$ according to the user-provided partial order $\sqsubseteq$. In λ_{LVar}, stores may grow by updating existing bindings via repeated puts, so Definition 4 would be too strong; for instance, if $\bot \sqsubset d_1 \sqsubseteq d_2$ for distinct $d_1, d_2 \in D$, the relationship $[l \mapsto d_1] \sqsubseteq_S [l \mapsto d_2]$ holds under Definition 2, but would not hold under Definition 4. That is, in λ_{LVar} an LVar could take on the state d_1 followed by d_2, which would not be possible in *Featherweight CnC*. We establish in Section 4 that λ_{LVar} remains *deterministic* despite the relatively weak $\sqsubseteq_S$ relation given in Definition 2. The keys to maintaining determinism are the blocking semantics of the get operation and the fact that it allows only *limited* observations of the state of an LVar.

3 λ_{LVar}: Syntax and Semantics

The syntax and operational semantics of λ_{LVar} appear in Figures 3 and 4, respectively.[4] As we've noted, both the syntax and semantics are parameterized by the domain D. The operational semantics is defined on *configurations* $\langle S;\, e\rangle$ comprising a store and an expression. The *error configuration*, written **error**, is a unique element added to the set of configurations, but we consider $\langle \top_S;\, e\rangle$ to be equal to **error**, for all expressions e. The metavariable σ ranges over configurations.

Figure 4 shows two disjoint sets of reduction rules: those that step to configurations other than **error**, and those that step to **error**. Most of the latter are merely propagating existing errors along. A new **error** can only arise by way of E-PARAPPERR, which represents the joining of two conflicting subcomputations, or by way of the E-PUTVALERR rule, which applies when a put to a location would take its state to $\top$.

The reduction rules E-NEW, E-PUTVAL, and E-GETVAL in Figure 4 respectively express the semantics of the new, put, and get operations described in Section 2.3. The incompatibility property of the threshold set argument to get is enforced in the E-GETVAL rule by the $incomp(Q)$ premise, which requires that the least upper bound of any two distinct elements in Q must be $\top$.[5] The E-PUT-1/E-PUT-2 and E-GET-1/E-GET-2 rules allow for reduction of subexpressions inside put and get expressions until their arguments have been evaluated, at which time the E-PUTVAL (or E-PUTVALERR) and E-GETVAL rules respectively apply. Arguments to put and get are evaluated in arbitrary order, although not simultaneously.[6]

3.1 Fork-Join Parallelism

λ_{LVar} has an explicitly parallel reduction semantics: the E-PARAPP rule in Figure 4 allows simultaneous reduction of the operator and operand in an application expression, so that (eliding stores) the application $e_1\ e_2$ may step to $e_1'\ e_2'$. In the case where one of the subexpressions is already a value or is otherwise unable to step (for instance, if it is a

[4] In addition to the version of λ_{LVar} presented here, we have developed a runnable model of a variant of λ_{LVar} using the PLT Redex semantics engineering toolkit [11]. Our Redex model and test suite are available at `https://github.com/lkuper/lambdaLVar-redex`.

[5] Although $incomp(Q)$ is given as a premise of the E-GETVAL reduction rule (indicating that it is checked at runtime), in a real implementation the incompatibility condition on threshold sets might be checked statically, eliminating the need for the runtime check. In fact, a real implementation could forego any runtime representation of threshold sets.

[6] It would, however, be straightforward to add to the semantics E-PARPUT and E-PARGET rules analogous to E-PARAPP, should simultaneous evaluation of put and get arguments be desired.

Given a domain D with elements $d \in D$, and a value-conversion function δ:

$incomp(Q) \triangleq \forall\, a, b \in Q.\ (a \neq b \implies a \sqcup b = \top)$

$\boxed{\langle S;\ e\rangle \longhookrightarrow \langle S';\ e'\rangle}$ (where $\langle S';\ e'\rangle \neq$ **error**)

E-REFL
$$\frac{}{\langle S;\ e\rangle \longhookrightarrow \langle S;\ e\rangle}$$

E-PARAPP
$$\frac{\langle S;\ e_1\rangle \longhookrightarrow \langle S_1;\ e_1'\rangle \qquad \langle S;\ e_2\rangle \longhookrightarrow \langle S_2;\ e_2'\rangle \qquad \langle S_1^r;\ e_1'^r\rangle = rename(\langle S_1;\ e_1'\rangle, S_2, S) \qquad S_1^r \sqcup_S S_2 \neq \top_S}{\langle S;\ e_1\ e_2\rangle \longhookrightarrow \langle S_1^r \sqcup_S S_2;\ e_1'^r\ e_2'\rangle}$$

E-PUT-1
$$\frac{\langle S;\ e_1\rangle \longhookrightarrow \langle S_1;\ e_1'\rangle}{\langle S;\ \mathsf{put}\ e_1\ e_2\rangle \longhookrightarrow \langle S_1;\ \mathsf{put}\ e_1'\ e_2\rangle}$$

E-PUT-2
$$\frac{\langle S;\ e_2\rangle \longhookrightarrow \langle S_2;\ e_2'\rangle}{\langle S;\ \mathsf{put}\ e_1\ e_2\rangle \longhookrightarrow \langle S_2;\ \mathsf{put}\ e_1\ e_2'\rangle}$$

E-PUTVAL
$$\frac{S(l) = d_2 \qquad d_1 \in D \qquad d_1 \sqcup d_2 \neq \top}{\langle S;\ \mathsf{put}\ l\ \{d_1\}\rangle \longhookrightarrow \langle S[l \mapsto d_1 \sqcup d_2];\ \{\}\rangle}$$

E-GET-1
$$\frac{\langle S;\ e_1\rangle \longhookrightarrow \langle S_1;\ e_1'\rangle}{\langle S;\ \mathsf{get}\ e_1\ e_2\rangle \longhookrightarrow \langle S_1;\ \mathsf{get}\ e_1'\ e_2\rangle}$$

E-GET-2
$$\frac{\langle S;\ e_2\rangle \longhookrightarrow \langle S_2;\ e_2'\rangle}{\langle S;\ \mathsf{get}\ e_1\ e_2\rangle \longhookrightarrow \langle S_2;\ \mathsf{get}\ e_1\ e_2'\rangle}$$

E-GETVAL
$$\frac{S(l) = d_2 \qquad incomp(Q) \qquad Q \subseteq D \qquad d_1 \in Q \qquad d_1 \sqsubseteq d_2}{\langle S;\ \mathsf{get}\ l\ Q\rangle \longhookrightarrow \langle S;\ \{d_1\}\rangle}$$

E-CONVERT
$$\frac{\langle S;\ e\rangle \longhookrightarrow \langle S';\ e'\rangle}{\langle S;\ \mathsf{convert}\ e\rangle \longhookrightarrow \langle S';\ \mathsf{convert}\ e'\rangle}$$

E-CONVERTVAL
$$\frac{}{\langle S;\ \mathsf{convert}\ v\rangle \longhookrightarrow \langle S;\ \delta(v)\rangle}$$

E-BETA
$$\frac{}{\langle S;\ (\lambda x.\ e)\ v\rangle \longhookrightarrow \langle S;\ e[x := v]\rangle}$$

E-NEW
$$\frac{}{\langle S;\ \mathsf{new}\rangle \longhookrightarrow \langle S[l \mapsto \bot];\ l\rangle}\ (l \notin dom(S))$$

$\boxed{\langle S;\ e\rangle \longhookrightarrow \textbf{error}}$

E-REFLERR
$$\frac{}{\textbf{error} \longhookrightarrow \textbf{error}}$$

E-PARAPPERR
$$\frac{\langle S;\ e_1\rangle \longhookrightarrow \langle S_1;\ e_1'\rangle \qquad \langle S;\ e_2\rangle \longhookrightarrow \langle S_2;\ e_2'\rangle \qquad \langle S_1^r;\ e_1'^r\rangle = rename(\langle S_1;\ e_1'\rangle, S_2, S) \qquad S_1^r \sqcup_S S_2 = \top_S}{\langle S;\ e_1\ e_2\rangle \longhookrightarrow \textbf{error}}$$

E-APPERR-1
$$\frac{\langle S;\ e_1\rangle \longhookrightarrow \textbf{error}}{\langle S;\ e_1\ e_2\rangle \longhookrightarrow \textbf{error}}$$

E-APPERR-2
$$\frac{\langle S;\ e_2\rangle \longhookrightarrow \textbf{error}}{\langle S;\ e_1\ e_2\rangle \longhookrightarrow \textbf{error}}$$

E-PUTERR-1
$$\frac{\langle S;\ e_1\rangle \longhookrightarrow \textbf{error}}{\langle S;\ \mathsf{put}\ e_1\ e_2\rangle \longhookrightarrow \textbf{error}}$$

E-PUTERR-2
$$\frac{\langle S;\ e_2\rangle \longhookrightarrow \textbf{error}}{\langle S;\ \mathsf{put}\ e_1\ e_2\rangle \longhookrightarrow \textbf{error}}$$

E-PUTVALERR
$$\frac{S(l) = d_2 \qquad d_1 \in D \qquad d_1 \sqcup d_2 = \top}{\langle S;\ \mathsf{put}\ l\ \{d_1\}\rangle \longhookrightarrow \textbf{error}}$$

E-GETERR-1
$$\frac{\langle S;\ e_1\rangle \longhookrightarrow \textbf{error}}{\langle S;\ \mathsf{get}\ e_1\ e_2\rangle \longhookrightarrow \textbf{error}}$$

E-GETERR-2
$$\frac{\langle S;\ e_2\rangle \longhookrightarrow \textbf{error}}{\langle S;\ \mathsf{get}\ e_1\ e_2\rangle \longhookrightarrow \textbf{error}}$$

E-CONVERTERR
$$\frac{\langle S;\ e\rangle \longhookrightarrow \textbf{error}}{\langle S;\ \mathsf{convert}\ e\rangle \longhookrightarrow \textbf{error}}$$

Figure 4: An operational semantics for λ_{LVar}.

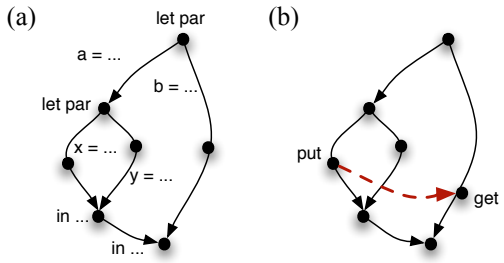

Figure 5: A series-parallel graph induced by basic parallel λ-calculus evaluation (a), vs. a non-series-parallel graph created by put/get communication (b).

blocked get), the reflexive E-REFL rule comes in handy: it allows the E-PARAPP rule to apply nevertheless. When the configuration $\langle S;\ e_1\ e_2\rangle$ takes a step, e_1 and e_2 step as separate subcomputations, each beginning with its own copy of the store S. Each subcomputation can update S independently, and the resulting two stores are combined by taking their least upper bound when the subcomputations rejoin.[7]

Although the semantics admits such parallel reductions, λ_{LVar} is still call-by-value in the sense that arguments to functions must be fully evaluated before function application (β-reduction, modeled by the E-BETA rule) can occur. We can exploit this property to define a syntactic sugar let par for *parallel composition*, which computes two subexpressions e_1 and e_2 in parallel before computing e_3:

$$\begin{array}{l}\textsf{let par } x = e_1 \\ \qquad\quad y = e_2 \\ \textsf{in } e_3\end{array} \qquad \triangleq \qquad ((\lambda x.\,(\lambda y.\,e_3))\ e_1)\ e_2$$

Although e_1 and e_2 are evaluated in parallel, e_3 cannot be evaluated until both e_1 and e_2 are evaluated, because the call-by-value semantics does not allow β-reduction until the operand is fully evaluated, and because it further disallows reduction under a λ-term (sometimes called "full β-reduction"). In the terminology of parallel programming, the above expression executes both a *fork* and a *join*. Indeed, it is common for fork and join to be combined in a single language construct, for example, in languages with parallel tuple expressions such as Manticore [12].

Since let par expresses *fork-join* parallelism, the evaluation of a program comprising nested let par expressions would induce a runtime dependence graph like that pictured in Figure 5(a). In the terminology of parallel algorithms, the λ_{LVar} language (minus put and get) can support any *series-parallel* dependence graph. Adding communication through put and get introduces "lateral" edges between branches of a parallel computation like that shown in Figure 5(b). This adds the ability to construct arbitrary non-series-parallel dependency graphs, just as with *first-class futures* [23].

Conversely, to *sequentially* compose e_1 before e_2 before e_3, we could write the expression $(\lambda x.\,((\lambda y.\,e_3)\ e_2))\ e_1$. Sequential composition is necessary for ordering side-effecting put and get operations on the store. For that reason, full β-reduction would be a poor choice, but parallel call-by-value gives λ_{LVar} both sequential and parallel composition, without introducing additional language forms.

[7] A subtle point that E-PARAPP and E-PARAPPERR must address is location renaming: locations created while e_1 steps must be renamed to avoid name conflicts with locations created while e_2 steps. We discuss the *rename* metafunction as part of a more wide-ranging discussion in Section 4.1.

3.2 Programming with put and get

For our first example of a λ_{LVar} program, we choose our domain to be pairs of natural-number-valued IVars, represented by the lattice shown in Figure 2(b). With D instantiated thusly, we can write the following program:[8]

$$\begin{array}{l}\mathsf{let}\ p = \mathsf{new}\ \mathsf{in}\\ \quad \mathsf{let}\ \_ = \mathsf{put}\ p\ \{(3,4)\}\ \mathsf{in}\\ \quad\quad \mathsf{let}\ v_1 = \mathsf{get}\ p\ \{(\bot, n) \mid n \in \mathbb{N}\}\ \mathsf{in}\\ \quad\quad\quad \ldots v_1 \ldots\end{array} \qquad \text{(Example 1)}$$

This program creates a new LVar p and stores the pair $(3, 4)$ in it. $(3, 4)$ then becomes the *state* of p. The premises of the E-GETVAL reduction rule hold: $S(p) = (3, 4)$; the threshold set $Q = \{(\bot, n) \mid n \in \mathbb{N}\}$ is a pairwise incompatible subset of D; and there exists an element $d_1 \in Q$ such that $d_1 \sqsubseteq (3, 4)$ in the lattice $(D, \sqsubseteq)$. In particular, the pair $(\bot, 4)$ is a member of Q, and $(\bot, 4) \sqsubseteq (3, 4)$ in $(D, \sqsubseteq)$. Therefore, $\mathsf{get}\ p\ \{(\bot, n) \mid n \in \mathbb{N}\}$ returns the singleton set $\{(\bot, 4)\}$, which is a first-class value in λ_{LVar} that can, for example, subsequently be passed to put.

Since threshold sets can be cumbersome to read, we can define some convenient shorthands getFst and getSnd for working with the domain of pairs:

$$\mathsf{getFst}\ p \triangleq \mathsf{get}\ p\ \{(n, \bot) \mid n \in \mathbb{N}\}$$
$$\mathsf{getSnd}\ p \triangleq \mathsf{get}\ p\ \{(\bot, n) \mid n \in \mathbb{N}\}$$

Querying incomplete data structures It is worth noting that getSnd p returns a value even if the first entry of p is not filled in. For example, if the put in the second line of (Example 1) had been $\mathsf{put}\ p\ \{(\bot, 4)\}$, the get expression would still return $\{(\bot, 4)\}$. It is therefore possible to safely query an incomplete data structure—say, an object that is in the process of being initialized by a constructor. However, notice that we *cannot* define a getFstOrSnd function that returns if either entry of a pair is filled in. Doing so would amount to passing all of the boxed elements of the lattice in Figure 2(b) to get as a single threshold set, which would fail the incompatibility criterion.

Blocking reads On the other hand, consider the following:

$$\begin{array}{l}\mathsf{let}\ p = \mathsf{new}\ \mathsf{in}\\ \quad \mathsf{let}\ \_ = \mathsf{put}\ p\ \{(\bot, 4)\}\ \mathsf{in}\\ \quad\quad \mathsf{let\ par}\ v_1 = \mathsf{getFst}\ p\\ \quad\quad\quad\quad\ \ \_\ \ = \mathsf{put}\ p\ \{(3,4)\}\\ \quad\quad\ \mathsf{in}\ \ldots v_1 \ldots\end{array} \qquad \text{(Example 2)}$$

Here getFst can attempt to read from the first entry of p before it has been written to. However, thanks to let par, the getFst operation is being evaluated in parallel with a put operation that will give it a value to read, so getFst simply *blocks* until $\mathsf{put}\ p\ \{(3, 4)\}$ has been evaluated, at which point the evaluation of $\mathsf{getFst}\ p$ can proceed.

In the operational semantics, this blocking behavior corresponds to the last premise of the E-GETVAL rule not being satisfied. In (Example 2), although the threshold set $\{(n, \bot) \mid n \in \mathbb{N}\}$ is incompatible, the E-GETVAL rule cannot apply because there is no state in the threshold set that is lower than the state of p in the lattice—that is, we are trying to get something that isn't yet there! It is only after p's state is updated that the premise is satisfied and the rule applies.

3.3 Converting from Threshold Sets to λ-terms and Back

There are two worlds that λ_{LVar} values may inhabit: the world of threshold sets, and the world of λ-terms. But if these worlds are disjoint—if threshold set values are opaque atoms—certain programs are impossible to write. For example,

[8] For clarity, we will write $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ as a shorthand for $((\lambda x.\, e_2)\ e_1)$.

Frame rule (O'Hearn *et al.*, 2001):

$$\frac{\{p\}\ c\ \{q\}}{\{p * r\}\ c\ \{q * r\}}\ \text{(where no free variable in } r \text{ is modified by } c\text{)}$$

Lemma 3 (Independence), simplified:

$$\frac{\langle S;\ e\rangle \hookrightarrow \langle S';\ e'\rangle}{\langle S \sqcup_S S'';\ e\rangle \hookrightarrow \langle S' \sqcup_S S'';\ e'\rangle}\ (S'' \text{ non-conflicting with } \langle S;\ e\rangle \hookrightarrow \langle S';\ e'\rangle)$$

Figure 6: Comparison of the frame rule with a simplified version of the Independence lemma. The $*$ connective in the frame rule requires that its arguments be disjoint.

implementing single-assignment arrays in λ_{LVar} requires that arbitrary array indices can be computed and converted to threshold sets.

Thus we parameterize our semantics by a *conversion function*, $\delta : v \to v$, to which λ_{LVar} provides an interface through its convert language form. The conversion function can arbitrarily convert between representations of values as threshold sets and representations as λ-terms. It is *optional* in the sense that providing an identity or empty function is acceptable, and leaves λ_{LVar} sensible but less expressive (*i.e.*, threshold sets are still first-class values, but usable only for passing to get and put).[9]

4 Proof of Determinism for λ_{LVar}

Our main technical result is a proof of determinism for the λ_{LVar} language. The complete proofs appear in Appendix A.

4.1 Framing and Renaming

Figure 6 shows a *frame rule*, due to O'Hearn *et al.* [20], which captures the idea that, given a program c with precondition p that holds before it runs and postcondition q that holds afterward, a disjoint condition r that holds before c runs will continue to hold afterward. Moreover, the original postcondition q will continue to hold. For λ_{LVar}, we can state a property that is analogous to the frame rule, but to do so we have to define a notion of *non-conflicting* stores. Given a transition $\langle S;\ e\rangle \hookrightarrow \langle S';\ e'\rangle$, the set $dom(S') - dom(S)$ is the set of names of *new* store bindings created between $\langle S;\ e\rangle$ and $\langle S';\ e'\rangle$. We say that a store S'' is *non-conflicting* with the transition $\langle S;\ e\rangle \hookrightarrow \langle S';\ e'\rangle$ iff $dom(S'')$ does not have any elements in common with $dom(S') - dom(S)$.

Definition 5. A store S'' is *non-conflicting* with the transition $\langle S;\ e\rangle \hookrightarrow \langle S';\ e'\rangle$ iff $(dom(S') - dom(S)) \cap dom(S'') = \emptyset$.

Requiring that a store S'' be non-conflicting with a transition $\langle S;\ e\rangle \hookrightarrow \langle S';\ e'\rangle$ is not as restrictive a requirement as it appears to be at first glance: it is fine for S'' to contain bindings for locations that are bound in S', as long as they are also locations bound in S. In fact, they may even be locations that were *updated* in the transition from $\langle S;\ e\rangle$ to $\langle S';\ e'\rangle$, as long as they were not *created* during it. In other words, given a store S'' that is non-conflicting with $\langle S;\ e\rangle \hookrightarrow \langle S';\ e'\rangle$, it may still be the case that $dom(S'')$ has elements in common with $dom(S)$, and with the subset of $dom(S')$ that is $dom(S)$.

Renaming Recall that when λ_{LVar} programs split into two subcomputations via the E-ParApp rule, the subcomputations' stores are merged (via the lub operation) as they are running. Therefore we need to ensure that the following two properties hold:

[9] A reasonable alternative definition of λ_{LVar} would remove threshold set values entirely and require that threshold set inputs and outputs to get/put be implicitly converted. Yet the language is deterministic even in its more general form—with first-class threshold sets—and we do not want to unduly restrict the language.

1. Location names created before a split still match up with each other after a merge.

2. Location names created by each subcomputation while they are running independently do *not* match up with each other accidentally—*i.e.*, they do not collide.

Property (2) is why it is necessary to *rename* locations in the E-PARAPP (and E-PARAPPERR) rule. This renaming is accomplished by a call to the $rename$ metafunction, which, for each location name l generated during the reduction $\langle S;\ e_1\rangle \longhookrightarrow \langle S_1;\ e_1'\rangle$, generates a name that is not yet used on either side of the split and substitutes that name into $\langle S_1;\ e_1'\rangle$ in place of l.[10] We arbitrarily choose to rename locations created during the reduction of $\langle S;\ e_1\rangle$, but it would work just as well to rename those created during the reduction of $\langle S;\ e_2\rangle$.

Definition 6. The $rename$ metafunction is defined as follows:

$$\begin{array}{lcl} rename(\cdot,\cdot,\cdot) & : & \sigma \times S \times S \to \sigma \\ rename(\langle S';\ e\rangle, S'', S) & \stackrel{\triangle}{=} & \langle S';\ e\rangle[l_1 := l_1'] \ldots [l_n := l_n'] \end{array}$$

where:

- $\{l_1, \ldots, l_n\} = dom(S') - dom(S)$, and
- $\{l_1', \ldots, l_n'\}$ is a set such that $l_i' \notin (dom(S') \cup dom(S''))$ for $i \in [1..n]$.

However, property (1) means that we cannot allow α-renaming of bound locations in a configuration to be done at will. Rather, renaming can only be done safely if it is done in the context of a *transition* from configuration to configuration. Therefore, we define a notion of *safe renaming* with respect to a transition.

Definition 7. A *renaming* of a configuration $\langle S;\ e\rangle$ is the substitution into $\langle S;\ e\rangle$ of location names $l_1', \ldots, l_n'$ for some subset $l_1, \ldots, l_n$ of $dom(S)$.

Definition 8. A *safe renaming of* $\langle S';\ e'\rangle$ *with respect to* $\langle S;\ e\rangle \longhookrightarrow \langle S';\ e'\rangle$ is a renaming of $\langle S';\ e'\rangle$ in which the locations $l_1, \ldots, l_n$ being renamed are the members of the set $dom(S') - dom(S)$, and the names $l_1', \ldots, l_n'$ that are replacing $l_1, \ldots, l_n$ do not appear in $dom(S')$.

If $\langle S'';\ e''\rangle$ is a safe renaming of $\langle S';\ e'\rangle$ with respect to $\langle S;\ e\rangle \longhookrightarrow \langle S';\ e'\rangle$, then S'' is by definition non-conflicting with $\langle S;\ e\rangle \longhookrightarrow \langle S';\ e'\rangle$.

4.2 Renaming Lemmas

With the aforementioned definitions in place, we can establish the following two properties about renaming. Lemma 1 expresses the idea that the names of locations created during a reduction step are arbitrary *within the context of that step*. It says that if a configuration $\langle S;\ e\rangle$ steps to $\langle S';\ e'\rangle$, then $\langle S;\ e\rangle$ can also step to configurations that are safe renamings of $\langle S';\ e'\rangle$ with respect to $\langle S;\ e\rangle \longhookrightarrow \langle S';\ e'\rangle$.

Lemma 1 (Renaming of Locations During a Step)**.** *If* $\langle S;\ e\rangle \longhookrightarrow \langle S';\ e'\rangle$ *(where* $\langle S';\ e'\rangle \neq$ **error***) and* $\{l_1, \ldots, l_n\} = dom(S') - dom(S)$*, then:*

For all sets $\{l_1', \ldots, l_n'\}$ *such that* $l_i' \notin dom(S')$ *for* $i \in [1..n]$*:*

$$\begin{array}{l} \langle S;\ e\rangle \longhookrightarrow \\ \langle S_{oldlocs}[l_1' \mapsto S'(l_1)] \ldots [l_n' \mapsto S'(l_n)];\ e'[l_1 := l_1'] \ldots [l_n := l_n']\rangle \\ (\neq \textbf{error}), \end{array}$$

where $S_{oldlocs}$ *is defined as follows:* $dom(S_{oldlocs}) = dom(S)$*, and for all* $l \in dom(S_{oldlocs})$*,* $S_{oldlocs}(l) = S'(l)$*.*

[10] Since λ_{LVar} locations are drawn from a distinguished set Loc, they cannot occur in the user's domain D—that is, locations in λ_{LVar} may not contain pointers to other locations. Likewise, λ-bound variables in e are never location names. Therefore, substitutions like the one in Definition 6 will not capture bound occurrences of location names.

Proof. See Appendix, Section A.1. □

Finally, Lemma 2 says that in the circumstances where we use the $rename$ metafunction, the renaming it performs meets the specification set by Lemma 1.

Lemma 2 (Safety of $rename$)**.** *If* $\langle S;\, e\rangle \hookrightarrow \langle S';\, e'\rangle$ *(where* $\langle S';\, e'\rangle \neq$ **error***) and* $S'' \neq \top_S$*, then:*
$\langle S;\, e\rangle \hookrightarrow rename(\langle S';\, e'\rangle, S'', S)$.

Proof. See Appendix, Section A.2. □

4.3 Supporting Lemmas

Lemmas 3, 4, and 5 express three key properties that we need for establishing determinism. Lemma 3 expresses a *local reasoning* property: it says that if a transition steps from $\langle S;\, e\rangle$ to $\langle S';\, e'\rangle$, then the configuration $\langle S \sqcup_S S'';\, e\rangle$, where S'' is some other store (*e.g.*, one from another subcomputation), will step to $\langle S' \sqcup_S S'';\, e'\rangle$. The only restrictions on S'' are that $S' \sqcup_S S''$ cannot be $\top_S$, and that S'' must be non-conflicting with the original transition $\langle S;\, e\rangle \hookrightarrow \langle S';\, e'\rangle$. Like the frame rule, the Independence lemma allows us to "frame in" a larger store around e and still finish the transition with e', with the non-conflicting requirement ruling out name conflicts caused by allocation.

Lemma 4 handles the case where $S' \sqcup_S S'' = \top_S$ and ensures that in that case, $\langle S \sqcup_S S'';\, e\rangle$ steps to **error**. In either case, whether the transition results in $\langle S' \sqcup_S S'';\, e'\rangle$ or in **error**, we know that it will never result in a configuration containing some other $e'' \neq e'$. Finally, Lemma 5 says that if a configuration $\langle S;\, e\rangle$ steps to **error**, then evaluating e in some larger store will also result in **error**.

Lemma 3 (Independence)**.** *If* $\langle S;\, e\rangle \hookrightarrow \langle S';\, e'\rangle$ *(where* $\langle S';\, e'\rangle \neq$ **error***), then for all* S'' *such that* S'' *is non-conflicting with* $\langle S;\, e\rangle \hookrightarrow \langle S';\, e'\rangle$ *and* $S' \sqcup_S S'' \neq \top_S$*:*
$\langle S \sqcup_S S'';\, e\rangle \hookrightarrow \langle S' \sqcup_S S'';\, e'\rangle$.

Proof. See Appendix, Section A.3. □

Lemma 4 (Clash)**.** *If* $\langle S;\, e\rangle \hookrightarrow \langle S';\, e'\rangle$ *(where* $\langle S';\, e'\rangle \neq$ **error***), then for all* S'' *such that* S'' *is non-conflicting with* $\langle S;\, e\rangle \hookrightarrow \langle S';\, e'\rangle$ *and* $S' \sqcup_S S'' = \top_S$*:*
$\langle S \sqcup_S S'';\, e\rangle \hookrightarrow$ **error**.

Proof. See Appendix, Section A.4. □

Lemma 5 (Error Preservation)**.** *If* $\langle S;\, e\rangle \hookrightarrow$ **error** *and* $S \sqsubseteq_S S'$*, then* $\langle S';\, e\rangle \hookrightarrow$ **error**.

Proof. See Appendix, Section A.5. □

4.4 Diamond Lemma

Lemma 6 does the heavy lifting of our determinism proof: it establishes the *diamond property* (or *Church-Rosser property* [4]), which says that if a configuration steps to two different configurations, there exists a single third configuration to which those configurations both step.

Lemma 6 (Diamond)**.** *If* $\sigma \hookrightarrow \sigma_a$ *and* $\sigma \hookrightarrow \sigma_b$*, then there exists* σ_c *such that either:*

- $\sigma_a \hookrightarrow \sigma_c$ *and* $\sigma_b \hookrightarrow \sigma_c$*, or*
- *there exists a safe renaming* σ'_b *of* σ_b *with respect to* $\sigma \hookrightarrow \sigma_b$*, such that* $\sigma_a \hookrightarrow \sigma_c$ *and* $\sigma'_b \hookrightarrow \sigma_c$*.*

Proof. See Appendix, Section A.6. □

We can readily restate Lemma 6 as Corollary 1:

Corollary 1 (Strong Local Confluence)**.** *If* $\sigma \hookrightarrow \sigma'$ *and* $\sigma \hookrightarrow \sigma''$*, then there exist* σ_c, i, j *such that* $\sigma' \hookrightarrow^i \sigma_c$ *and* $\sigma'' \hookrightarrow^j \sigma_c$ *and* $i \leq 1$ *and* $j \leq 1$*.*

Proof. Choose $i = j = 1$. The proof follows immediately from Lemma 6. □

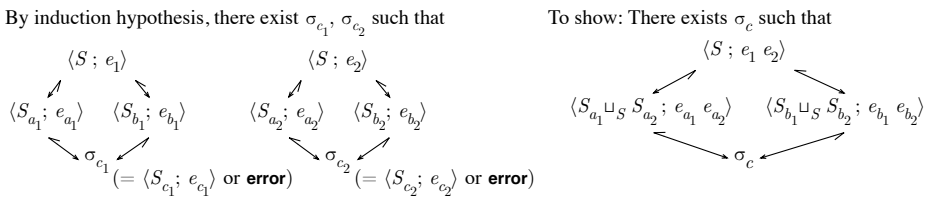

Figure 7: Diagram of the subcase of Lemma 6 in which the E-PARAPP rule is the last rule in the derivation of both $\sigma \longrightarrow \sigma_a$ and $\sigma \longrightarrow \sigma_b$. We are required to show that, if the configuration $\langle S;\, e_1\ e_2\rangle$ steps by E-PARAPP to two different configurations, $\langle S_{a_1} \sqcup_S S_{a_2};\, e_{a_1}\ e_{a_2}\rangle$ and $\langle S_{b_1} \sqcup_S S_{b_2};\, e_{b_1}\ e_{b_2}\rangle$, they both step to some third configuration σ_c.

4.5 Confluence Lemmas and Determinism

With Lemma 6 in place, we can straightforwardly generalize its result to multiple steps, by induction on the number of steps, as Lemmas 7, 8, and 9 show. [11]

Lemma 7 (Strong One-Sided Confluence)**.** *If $\sigma \longrightarrow \sigma'$ and $\sigma \longrightarrow^m \sigma''$, where $1 \leq m$, then there exist σ_c, i, j such that $\sigma' \longrightarrow^i \sigma_c$ and $\sigma'' \longrightarrow^j \sigma_c$ and $i \leq m$ and $j \leq 1$.*

Proof. See Appendix, Section A.7. □

Lemma 8 (Strong Confluence)**.** *If $\sigma \longrightarrow^n \sigma'$ and $\sigma \longrightarrow^m \sigma''$, where $1 \leq n$ and $1 \leq m$, then there exist σ_c, i, j such that $\sigma' \longrightarrow^i \sigma_c$ and $\sigma'' \longrightarrow^j \sigma_c$ and $i \leq m$ and $j \leq n$.*

Proof. See Appendix, Section A.8. □

Lemma 9 (Confluence)**.** *If $\sigma \longrightarrow^* \sigma'$ and $\sigma \longrightarrow^* \sigma''$, then there exists σ_c such that $\sigma' \longrightarrow^* \sigma_c$ and $\sigma'' \longrightarrow^* \sigma_c$.*

Proof. Strong Confluence (Lemma 8) implies Confluence. □

Theorem 1 (Determinism)**.** *If $\sigma \longrightarrow^* \sigma'$ and $\sigma \longrightarrow^* \sigma''$, and neither σ' nor σ'' can take a step except by* E-REFL *or* E-REFLERR*, then $\sigma' = \sigma''$.*

Proof. We have from Lemma 9 that there exists σ_c such that $\sigma' \longrightarrow^* \sigma_c$ and $\sigma'' \longrightarrow^* \sigma_c$. Since σ' and σ'' can only step to themselves, we must have $\sigma' = \sigma_c$ and $\sigma'' = \sigma_c$, hence $\sigma' = \sigma''$. □

5 Modeling Other Deterministic Parallel Models

In this section, we present evidence that the λ_{LVar} programming model is general enough to subsume two rather different families of deterministic-by-construction parallel computation models. The first category is single-assignment models, from which we'll take Intel's Concurrent Collections framework [7] and Haskell's `monad-par` library [18] as two examples. The second is data-flow networks, specifically Kahn process networks (KPNs) [15]. In Section 7, we discuss additional models that are related to, but not directly modeled by, λ_{LVar}.

[11] Lemmas 7, 8, and 9 are nearly identical to the corresponding lemmas in the proof of determinism for *Featherweight CnC* given by Budimlić *et al.*[7]. We also reuse Budimlić *et al.*'s naming conventions for Lemmas 3 through 6, but the statements and proofs of those lemmas differ considerably in our setting.

5.1 Concurrent Collections

In Section 2.4, we mentioned the *Featherweight CnC* language and its monotonically growing memory store. *Featherweight CnC* is a simplified model of the Concurrent Collections (CnC) [7] language for composing graphs of "steps", more commonly known as *actors*, which are implemented separately in a general-purpose language (C++, Java, Haskell, or Python). To begin execution, a subset of steps are invoked at startup time. Each step, when executed, may perform puts and gets on global, shared data collections (tables of IVars), as well as send messages to invoke other steps. The steps themselves are stateless, except for the information they store externally in the aforementioned tables.

The role of monotonicity has been understood, at least informally, in the design of CnC. However, this has not—until now—led to a treatment of shared data collections as general as λ_{LVar}. λ_{LVar} subsumes CnC in the following sense. If the language used to express CnC steps is the call-by-value λ-calculus, then CnC programs can be translated to λ_{LVar}; each step would become a function definition, generated in the following way:

- Each step function takes a single argument (its message, or in CnC terminology, its *tag*) and returns $\{\}$—our *unit*, the empty threshold set—being executed for effect only.
- All invocations of other steps (message sends) within the step body, are aggregated at the end of the function and performed inside a let par. This is the sole source of parallelism. The aggregation can be accomplished either statically, by a program transformation that moves sends, or by dynamic buffering of the outgoing sends.
- The rest of the body of a step is translated directly: puts on data collections become λ_{LVar} puts; gets become become λ_{LVar} gets.

The following skeleton shows the form of a program converted by the above method. It first defines steps, then launches the initial batch of "messages", and finally reads whatever result is desired.

```
let step1 = λmsg. get ... ; put ... ;
                  let par _ = step1 ...
                          _ = step2 ...
                          _ = step2 ...
                  in {}
in let step2 = ...
in let data1 = new      - - global data collections
in let par _ = step1 33    - - invoke initial steps
         _ = step2 44
in convert (get data1 key)    - - retrieve final result
```

Somewhat surprisingly, the CnC programming model is *not* implementable in a parallel call-by-value λ-calculus extended only with IVars. In fact, it was this observation that began the inquiry leading to the development of λ_{LVar}. The reason is that CnC provides globally scoped, extensible *tables* of IVars, not just IVars alone. While a λ-calculus augmented with IVars could model shared global IVars, or even fixed collections of IVars, it is, to our knowledge, impossible to create a mutable, extensible table data structure with IVars alone.

Finally, if there were not already a determinism result for CnC (which is previous work by the second author and others [7]), one could bootstrap determinism by proving that every valid step in a CnC semantics maps onto one or more evaluation steps for the translated version under the λ_{LVar} semantics; that is, the λ_{LVar} encoding simulates all possible executions of the CnC program, and since it yields a single answer, so does the CnC program.

5.2 The `monad-par` Haskell library

The `monad-par` package for Haskell [18] provides a parallel deterministic programming model with an explicit fork operation together with first-class IVars. `monad-par` uses explicit sequencing via a monad, together with Haskell's

lazy evaluation. To translate `monad-par` programs to λ_{LVar}, evaluation order can be addressed using standard techniques, and λ_{LVar} can model `monad-par`'s fork operation with let par, using the method in Section 3.1. But because `monad-par` has no *join* operations (IVar gets being the only synchronization mechanism), it would be necessary to use continuation-passing style in the translation. If the original `monad-par` program forks a child computation and returns, the translated program must invoke both the fork and its continuation within a let par expression.

Another wrinkle for translation of `monad-par` programs into λ_{LVar} is that while `monad-par` IVars may contain other IVars, LVars cannot contain LVars. This problem can be overcome by using a type-directed translation in which ach IVar is represented by the wide, height-three lattice shown in Figure 2(a), and multiple IVars are modeled by product lattices. For example, a location of type `IVar (IVar Int, IVar Char)` in `monad-par` would correspond to a lattice similar to that pictured in Figure 2(b). Chaining IVar type constructors, *e.g.*, `IVar (IVar (...))`, would simply add additional empty states, repeatedly lifting the domain with a new $\bot$. All these types create larger state spaces, but do not pose a fundamental barrier to encoding `monad-par` IVars as LVars.

Although λ_{LVar} is a calculus rather than a practical programming language, the exercise of modeling `monad-par` in λ_{LVar} suggests practical extensions to `monad-par`. For example, additional data structures beyond IVars could be provided (*e.g.*, maps or tries), using the λ_{LVar} translation to ensure determinism is retained.

5.3 Kahn Process Networks

Data-flow models have been a topic of theoretical [15] and practical [14] study for decades. In particular, Kahn's 1974 paper crystallized the contemporary work on data-flow with a denotational account of *Kahn process networks* (KPNs)—a deterministic model in which a network of processes communicates through single-reader, single-writer FIFO channels with non-blocking writes and blocking reads. Because λ_{LVar} is general enough to subsume KPNs, it represents a step towards bringing the body of work on data-flow into the broader context of functional and single-assignment languages.

To map KPNs into λ_{LVar}, we represent FIFOs as ordered sequences of values, monotonically growing on one end (*i.e.*, channel *histories*). In fact, the original work on KPNs [15] used exactly this representation (and the complete partial order based on it) to establish determinism. However, to our knowledge neither KPNs nor any other data-flow model has generalized the data structures used for communication beyond FIFOs to include other monotonically-growing structures (*e.g.*, maps).

An LVar representing a FIFO has a state encoding all elements sent on that FIFO to date. We represent sequences as sets of (*index, value*) associations with subset inclusion as the order $\sqsubseteq$. For example, $\{(0, a), (1, b)\}$ encodes a two-element sequence. This makes it convenient to write threshold sets such as $\{(0, n) \mid n \in \mathbb{N}\}$, which will match any state encoding a channel history with a natural number value in position 0.

In this encoding, the producers and consumers using a FIFO must explicitly keep track of what position they read and write, *i.e.*, the "cursor". This contrasts with an imperative formulation, where advancing the cursor is a side effect of "popping" the FIFO. A proper encoding of FIFO behavior writes and reads consecutive positions only.[12]

But what of the deterministic processes themselves? In Kahn's original work, they are treated as functions on channel histories without any internal structure. In a λ_{LVar} formulation of KPNs, they take the form of recursive functions that carry their state (and cursor positions) as arguments. In Figure 8, we use self-application to enable recursion, and we express a stream filter `filterDups` that prunes out all duplicate consecutive numbers from a stream.

Figure 8 assumes quite a bit in the way of syntactic sugar, although nothing non-standard. Church numerals would be needed to encode natural numbers, as well as a standard encoding of booleans. Because the encoding of Figure 8 will only work for finite executions, the cnt argument tells `filterDups` how many input elements to process. The fourth argument, lst, tracks the previously observed element on the input stream, that is, the *state* of the stream transducer. The second and third arguments to `filterDups` are the cursors that track positions in both the input and output streams. The convert function is necessary for *computing* threshold sets based on the values of cursors.

This technique is sufficient for encoding arbitrary KPN programs into λ_{LVar}. It is by no means a natural expression of this concept, especially due to the fact that the input and output stream cursors must be tracked explicitly. However,

[12] In the λ_{LVar} abstraction we don't address concrete representations or storage requirements for LVar states and threshold sets. In a practical implementation, one would expect that already-consumed FIFO elements would be garbage-collected, which in turn *requires* strict enforcement of consecutively increasing access only.

```
let filterDups = λf i1 i2 lst cnt.
    let next = get inp (convert i1)
        i2' = if (lst = next) then i2
                  else put outp (convert (i2, next)); (i2 + 1)
    in if (cnt = 0) then {}
       else f f (i1 + 1) i2' next (cnt − 1)
in filterDups filterDups 0 ...
where convert i = {{(i, n)} | n ∈ ℕ}
      convert (i, n) = {{(i, n)}}
```

Figure 8: Process an input stream, removing consecutive duplicates. inp and $outp$ are channels, globally bound elsewhere.

with additional infrastructure for tracking stream cursors (and other state) by means of a state monad, the program given in Figure 8 could become significantly more idiomatic.

6 Safe, Limited Nondeterminism

In practice, a major problem with nondeterministic programs is that they can *silently* go wrong. Most parallel programming models are *unsafe* in this sense, but we may classify a nondeterministic language as *safe* if all occurrences of nondeterminism—that is, execution paths that would yield an incorrect answer—are caught and reported as errors. This notion of *safe nondeterminism* is analogous to the concept of type safety: type-safe programs can throw exceptions, but they will not "go wrong". We find that there are various extensions[13] to a deterministic language make it safely nondeterministic. Here, we will look at one such extension: *exact but destructive observations*.

We take as our motivating example the shared, increment-only counter of Figure 2(c), and begin with the observation that when the state of a shared counter has come to rest—when no more increments will occur—then its final value is a deterministic function of program inputs, and is therefore safe to read directly. The problem is determining *when* an LVar has come to rest. However, *if* the value of an LVar is indeed at rest, then we do no harm to it by corrupting its state in such a way that further increments will lead to an error. We can accomplish this by adding an extra state, called *probation*, to the domain D. The lattice defined by the relation $\sqsubseteq$ is extended thus:

$$\begin{aligned} &\mathit{probation} \sqsubseteq \top \\ &\forall d \in D.\ d \not\sqsubseteq \mathit{probation} \end{aligned}$$

[13] While not recognized explicitly by the authors as such, a recent extension to CnC for memory management incidentally fell into this category [21].

```
let cnt = new in
  let sum = new in
    let par p1 = (bump3 sum; bump1 cnt)
            p2 = (bump4 sum; bump1 cnt)
            p3 = (bump5 sum; bump1 cnt)
            r  = (get cnt 3; consume sum)
      in ...r...
```

Figure 9: A deterministic program that makes destructive observations.

We then propose a new operation, consume, that takes a pointer to an LVar l, updates the store, setting l's state to *probation*, and returns a singleton set containing the *exact* previous state of l, rather than a lower bound on that state. The idea is to ensure that, after a consume, any further operations on l will go awry: put operations will attempt to move the state of l to $\top$, which will cause the system to step to **error**.

Figure 9 shows an example program that uses consume to perform an asynchronous sum reduction over a known number of inputs. In such a reduction, data dependencies alone determine when the reduction is complete, rather than control constructs such as parallel loops and barriers. In Figure 9 we use semicolon as sugar for sequential composition: for example, $e_1; e_2$ rather than let $\_ = e_1$ in e_2. We also assume a new syntactic sugar in the form of a bump operation that takes a pointer to an LVar and increments it by one, with $\mathsf{bump}_n\ l$ as an additional shorthand for n consecutive bumps to l. The get cnt 3 before the call to consume serves as a synchronization mechanism, ensuring that all increments are complete before the value is read. Three writers and one reader execute in parallel, and only when all writers complete does the reader return the sum, which in this case will be $3 + 4 + 5 = 12$.

The good news is that the program of Figure 9 is correct and deterministic; it will always return the same value in any execution. However, the consume primitive in general admits safe nondeterminism, meaning that, while all runs of the program will terminate with the same value *if* they terminate without error, some runs of the program may terminate in **error**, in spite of other runs completing successfully. To see how an error might occur, imagine an alternate version of the program of Figure 9 in which get cnt 3 is replaced by get cnt 2. This version would have insufficient synchronization. The program could run correctly many times—if the bumps happen to complete before the consume operation executes—and yet step to **error** on the thousandth run. Yet, with safe nondeterminism, it is possible to catch and respond to this error, for example by rerunning in a debug mode that is guaranteed to find a valid execution if it exists, or by using a *data-race detector* which will reproduce all races in the execution in question. We have implemented example interpreters and a race-detector for λ_{LVar}, available at `http://github.com/rrnewton/lambdapar_interps`.

6.1 Syntactic Sugar for Counting

Strictly speaking, if we directly use the lattice of Figure 2(c), the bump operation would not be possible. Therefore, rather than use the domain in Figure 2(c) directly, we can simulate it using a power-set lattice over an arbitrary alphabet of symbols $\{a, b, c, \ldots\}$, ordered by subset inclusion. LVars occupying such a lattice encode natural numbers using the cardinality of the subset.[14] Thus, a blocking get operation that unblocks when the count reaches, say, 3 would take a threshold set enumerating all the three-element subsets of the alphabet.

With this encoding, incrementing a shared variable l requires put l $\{\alpha\}$, where $\alpha \in \{a, b, c, \ldots\}$ and α has not previously been used. Thus, without any additional support, a hypothetical programmer would be responsible for creating a unique α for each parallel contribution to the counter. There are well-known techniques, however, for generating a unique (but schedule-invariant and deterministic) identifier for a given point in a parallel execution. One solution is to reify the position of an operation inside a tree (or DAG) of parallel evaluations. The Cilk Plus parallel programming language refers to this notion as the operation's *pedigree* and uses it to seed a deterministic parallel random number generator [17].

With this encoding, we can implement an expression unique, which, when evaluated, returns a singleton threshold set containing a single unique element of the alphabet: $\{\alpha\}$. With the unique syntax, we can write programs like the following, in which two parallel threads increment the same counter:

```
let sum = new in
  let par p1 = (put sum unique; put sum unique)
          p2 = (put sum unique)
      in ...
```
(Example 3)

In this case, the p_1 and p_2 "threads" will together increment the sum by three. Notice that consecutive increments performed by p_2 are not atomic. With unique in place, bump l desugars to put l unique. The unique construct could be implemented by a whole-program transformation over a sugared λ_{LVar} expression. Figure 10 shows one possible

[14] Of course, just as with an encoding like Church numerals, this encoding would never be used by a realistic implementation.

$$
\begin{aligned}
[\![\mathsf{unique}]\!] &= \lambda p.\ \mathsf{convert}\ p \\
[\![v]\!] &= \lambda p.\ v \\
[\![Q]\!] &= \lambda p.\ Q \\
[\![\lambda v.\ e]\!] &= \lambda p.\ \lambda v.\ [\![e]\!] \\
[\![\mathsf{new}]\!] &= \lambda p.\ new \\
[\![e1\ e2]\!] &= \lambda p.\ (([\![e1]\!]\ L{:}p)\ ([\![e2]\!]\ R{:}p)\ J{:}p) \\
[\![\mathsf{put}\ a\ b]\!] &= \lambda p.\ \mathsf{put}\ ([\![a]\!]\ L{:}p)\ ([\![b]\!]\ R{:}p) \\
[\![\mathsf{get}\ a\ b]\!] &= \lambda p.\ \mathsf{get}\ ([\![a]\!]\ L{:}p)\ ([\![b]\!]\ R{:}p) \\
[\![\mathsf{convert}\ e]\!] &= \lambda p.\ \mathsf{convert}\ ([\![e]\!]\ p)
\end{aligned}
$$

Figure 10: Rewrite rules for desugaring the unique construct within λ_{LVar} programs. Here we use "L:", "R:", "J:" to *cons* onto the front of a list that represents a path within a fork/join DAG. The symbols mean, respectively, "left branch", "right branch", or "after the join" of the two branches. This requires a λ-calculus encoding of lists, as well as a definition of convert that is an injective function from these list values onto the domain D.

implementation. It creates a tree that tracks the dynamic evaluation of applications, and shows some similarity to a continuation-passing style transformation [10].

7 Related Work

Work on deterministic parallel programming models is long-standing. In addition to the single-assignment and KPN models already discussed, here we consider a few recent contributions to the literature.

Deterministic Parallel Java (DPJ) DPJ [6] is a deterministic language consisting of a system of annotations for Java code. A sophisticated region-based type system ensures that a mutable region of the heap is, essentially, passed linearly to an exclusive writer. While a linear type system or region system like that of DPJ could be used to enforce single assignment statically, accommodating λ_{LVar}'s semantics would involve parameterizing the type system by the user-specified domain—a direction of inquiry that we leave for future work.

DPJ also provides a way to unsafely assert that operations commute with one another (using the `commutesWith` form) to enable concurrent mutation. However, DPJ does not provide direct support for modeling message-passing (*e.g.*, KPNs) or asynchronous communication within parallel regions. Finally, a key difference between the λ_{LVar} model and DPJ is that λ_{LVar} retains determinism by restricting *what* can be read or written, rather than by restricting *who* can read or write.

Concurrent Revisions The Concurrent Revisions (CR) [16] programming model uses isolation types to distinguish regions of the heap shared by multiple mutators. Rather than enforcing exclusive access, CR clones a copy of the state for each mutator, using a deterministic policy for resolving conflicts in local copies. The management of shared variables in CR is tightly coupled to a fork-join control structure, and the implementation of these variables is similar to reduction variables in other languages (*e.g.*, Cilk *hyperobjects* [13]). CR charts an important new area in the deterministic-parallelism design space, but one that differs significantly from λ_{LVar}. CR could be used to model similar types of data structures—if versioned variables used least upper bound as their merge function for conflicts—but effects would only become visible at the end of parallel regions, rather than λ_{LVar}'s asynchronous communication within parallel regions.

Bloom and Bloom^L In the distributed systems literature, *eventually consistent* systems [25] leverage the idea of monotonicity to guarantee that, for instance, nodes in a distributed database eventually agree. The Bloom language for

distributed database programming [1] guarantees eventual consistency for distributed data collections that are updated monotonically. The initial formulation of Bloom [2] had a notion of monotonicity based on *set containment*, analogous to the store ordering for single-assignment languages given in Definition 4. However, recent work by Conway *et al.* [9] generalizes Bloom to a more flexible lattice-parameterized system, Bloom^L, in a manner analogous to our generalization from IVars to LVars. Bloom^L comes with a library of built-in lattice types and also allows for users to implement their own lattice types as Ruby classes. Although Conway *et al.* do not give a proof of eventual consistency for Bloom^L, our determinism result for λ_{LVar} suggests that their generalization is indeed safe. Moreover, although the goals of Bloom differ from those of λ_{LVar}, we believe that Bloom^L bodes well for programmers' willingness to use lattice-based data structures like LVars, and lattice-parameterized languages based on them, to address real-world programming challenges.

Quantum programming The λ_{LVar} semantics is reminiscent of the semantics of quantum programming languages that extend a conventional λ-calculus with a store that maintains the quantum state. Because of quantum parallelism, the quantum state can be accessed by many threads in parallel, but only through a restricted interface. As a concrete example, the language designed by Selinger and Valiron [22] allows only the following operations on quantum data: (1) "appending" to the current data using the tensor product; (2) performing a unitary operation that must, by definition, act linearly and uniformly on the data; and (3) selecting a set of orthogonal subspaces and performing a measurement that projects the quantum state onto one of the subspaces. These operations correspond roughly to λ_{LVar}'s new, put, and get. Quantum mechanics may serve as a source of inspiration when designing operations like consume that introduce limited nondeterminism.

8 Conclusion

As single-assignment languages and Kahn process networks demonstrate, monotonicity serves as the foundation of deterministic parallelism. Taking monotonicity as a starting point, our work generalizes single assignment to monotonic multiple assignment parameterized by a user-specified lattice. By combining monotonic writes with threshold reads, we get a shared-state parallel programming model that generalizes and unifies an entire class of monotonic languages suitable for asynchronous, data-driven applications. Our model is provably deterministic, and further provides a foundation for exploration of limited nondeterminism. Future work will investigate implementation strategies, formally establish the relationship between λ_{LVar} and other deterministic parallel models, and prove the more limited guarantees provided by $\lambda_{\text{LVar}} +$ consume.

Acknowledgments

We thank Amr Sabry for his feedback and tireless assistance through all stages of our work. This research was funded in part by NSF grant CCF-1218375.

References

[1] Bloom homepage. URL `http://bloom-lang.net`.

[2] P. Alvaro, N. Conway, J. Hellerstein, and W. R. Marczak. Consistency analysis in Bloom: a CALM and collected approach. In *CIDR*, 2011.

[3] Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: data structures for parallel computing. *ACM Trans. Program. Lang. Syst.*, 11, October 1989.

[4] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[5] D. A. Bader and K. Madduri. Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2. ICPP. IEEE Computer Society, 2006.

[6] R. L. Bocchino, Jr. et al. Safe nondeterminism in a deterministic-by-default parallel language. In *POPL*, 2011.

[7] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, and S. Taşirlar. Concurrent Collections. *Sci. Program.*, 18, August 2010. URL `http://dl.acm.org/citation.cfm?id=1938482.1938486`.

[8] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 21(3), 2007.

[9] N. Conway, W. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier. Logic and lattices for distributed programming. In *SOCC*, 2012.

[10] O. Danvy and A. Filinski. Representing control: a study of the CPS transformation, 1992.

[11] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 1st edition, 2009.

[12] M. Fluet, M. Rainey, J. Reppy, and A. Shaw. Implicitly-threaded parallelism in Manticore. In *ICFP*, 2008.

[13] M. Frigo et al. Reducers and other Cilk++ hyperobjects. In *SPAA*, 2009.

[14] J. Hicks, D. Chiou, B. S. Ang, and Arvind. Performance studies of Id on the Monsoon dataflow system. *J. Parallel Distrib. Comput.*, 1993.

[15] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*. North Holland, Amsterdam, Aug 1974.

[16] D. Leijen, M. Fahndrich, and S. Burckhardt. Prettier concurrency: purely functional concurrent revisions. In *Haskell*, 2011.

[17] C. E. Leiserson, T. B. Schardl, and J. Sukha. Deterministic parallel random-number generation for dynamic-multithreading platforms. In *PPoPP*, 2012.

[18] S. Marlow, R. Newton, and S. Peyton Jones. A monad for deterministic parallelism. In *Haskell*, 2011.

[19] R. Newton et al. Deterministic reductions in an asynchronous parallel language. In *Workshop on Determinism and Correctness in Parallel Programming (WoDet)*, 2011.

[20] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In L. Fribourg, editor, *CSL*, volume 2142 of *Lecture Notes in Computer Science*. Springer, 2001.

[21] D. Sbirlea, K. Knobe, and V. Sarkar. Folding of tagged single assignment values for memory-efficient parallelism. In C. Kaklamanis, T. S. Papatheodorou, and P. G. Spirakis, editors, *Euro-Par*, volume 7484 of *Lecture Notes in Computer Science*, pages 601–613. Springer, 2012. ISBN 978-3-642-32819-0.

[22] P. Selinger and B. Valiron. A lambda calculus for quantum computation with classical control. *Mathematical Structures in Computer Science*, 16(3), 2006.

[23] D. Spoonhower, G. E. Blelloch, P. B. Gibbons, and R. Harper. Beyond nested parallelism: tight bounds on work-stealing overheads for parallel futures. In *SPAA*, 2009.

[24] L. G. Tesler and H. J. Enea. A language design for concurrent processes. In *AFIPS*, 1968 (Spring).

[25] W. Vogels. Eventually consistent. *Commun. ACM*, 52(1), Jan. 2009. doi: 10.1145/1435417.1435432.

[26] K. B. Wheeler, R. C. Murphy, and D. Thain. Qthreads: An API for programming with millions of lightweight threads, 2008.

A Proof of Determinism

Definition 9. Two stores S and S' are *equal* iff:

1. $S = \top_S$ and $S' = \top_S$, or
2. $dom(S) = dom(S')$ and for all $l \in dom(S)$, $S(l) = S'(l)$.

A.1 Renaming of Locations During a Step

Lemma 1 (Renaming of Locations During a Step)**.** *If* $\langle S;\, e\rangle \hookrightarrow \langle S';\, e'\rangle$ *(where* $\langle S';\, e'\rangle \neq$ **error***) and* $\{l_1, \ldots, l_n\} = dom(S') - dom(S)$*, then:*

For all sets $\{l'_1, \ldots, l'_n\}$ *such that* $l'_i \notin dom(S')$ *for* $i \in [1..n]$*:*

$$\begin{array}{l}\langle S;\, e\rangle \hookrightarrow \\ \langle S_{oldlocs}[l'_1 \mapsto S'(l_1)] \ldots [l'_n \mapsto S'(l_n)];\, e'[l_1 := l'_1] \ldots [l_n := l'_n]\rangle \\ (\neq \textbf{error}),\end{array}$$

where $S_{oldlocs}$ *is defined as follows:* $dom(S_{oldlocs}) = dom(S)$*, and for all* $l \in dom(S_{oldlocs})$*,* $S_{oldlocs}(l) = S'(l)$*.*

Proof. By induction on the derivation of $\langle S;\, e\rangle \hookrightarrow \langle S';\, e'\rangle$, by cases on the last rule in the derivation. Since $\langle S';\, e'\rangle \neq$ **error**, we only need to consider rules that step to non-**error** configurations. In cases where $dom(S') - dom(S) = \emptyset$, then the only possible set $\{l'_1, \ldots, l'_n\}$ is also $\emptyset$, so in such cases we need only show that $\langle S;\, e\rangle \hookrightarrow \langle S_{oldlocs};\, e'\rangle$.

A.1.1 E-REFL

- E-REFL:

  Given: $\langle S;\, e\rangle \hookrightarrow \langle S;\, e\rangle$.

  To show: $\langle S;\, e\rangle \hookrightarrow \langle S_{oldlocs};\, e\rangle$, where $S_{oldlocs}$ is defined as follows: $dom(S_{oldlocs}) = dom(S)$, and for all $l \in dom(S_{oldlocs})$, $S_{oldlocs}(l) = S(l)$.

  Since $dom(S_{oldlocs}) = dom(S)$ and since for all $l \in dom(S_{oldlocs})$, $S_{oldlocs}(l) = S(l)$, we have by Definition 9 that $S_{oldlocs} = S$, so the case is immediate by E-REFL.

A.1.2 E-PARAPP

- E-PARAPP:

  (NB: For simplicity, we elide renaming of $\langle S_1;\, e'_1\rangle$ in this case, and assume without loss of generality that location names created during the transition $\langle S;\, e_1\rangle \hookrightarrow \langle S_1;\, e'_1\rangle$ are distinct from those created during the transition $\langle S;\, e_2\rangle \hookrightarrow \langle S_2;\, e'_2\rangle$.)

  Given: $\langle S;\, e_1\ e_2\rangle \hookrightarrow \langle S_1 \sqcup_S S_2;\, e'_1\ e'_2\rangle$ and $\{l_1, \ldots, l_n\} = dom(S_1 \sqcup_S S_2) - dom(S)$.

  To show: For all sets $\{l'_1, \ldots, l'_n\}$ such that $l'_i \notin dom(S_1 \sqcup_S S_2)$ for $i \in [1..n]$,

  $$\langle S;\, e_1\ e_2\rangle \hookrightarrow \langle S_{oldlocs}[l'_1 \mapsto (S_1 \sqcup_S S_2)(l_1)] \ldots [l'_n \mapsto (S_1 \sqcup_S S_2)(l_n)];\, (e'_1\ e'_2)[l_1 := l'_1] \ldots [l_n := l'_n]\rangle,$$

  where $S_{oldlocs}$ is defined as follows: $dom(S_{oldlocs}) = dom(S)$, and for all $l \in dom(S_{oldlocs})$, $S_{oldlocs}(l) = (S_1 \sqcup_S S_2)(l)$.

  Consider arbitrary $\{l'_1, \ldots, l'_n\}$ such that $l'_i \notin dom(S_1 \sqcup_S S_2)$ for $i \in [1..n]$.

  From the first two premises of E-PARAPP, we have that $\langle S;\, e_1\rangle \hookrightarrow \langle S_1;\, e'_1\rangle$ and $\langle S;\, e_2\rangle \hookrightarrow \langle S_2;\, e'_2\rangle$.

Since we assume that location names created during $\langle S;\ e_1\rangle \longrightarrow \langle S_1;\ e_1'\rangle$ are distinct from those created during $\langle S;\ e_2\rangle \longrightarrow \langle S_2;\ e_2'\rangle$, and since $\{l_1, \ldots, l_n\} = dom(S_1 \sqcup_S S_2) - dom(S)$, then we have that $dom(S_1) - dom(S) = \{l_1, \ldots, l_k\}$ and $dom(S_2) - dom(S) = \{l_{k+1}, \ldots, l_n\}$ for some k such that $\{l_1, \ldots, l_k\} \cap \{l_{k+1}, \ldots, l_n\} = \emptyset$ and $\{l_1, \ldots, l_k\} \uplus \{l_{k+1}, \ldots, l_n\} = \{l_1, \ldots, l_n\}$.

Then, by IH, we have the following two facts:

1. For all sets $\{l_1', \ldots, l_k'\}$ such that $l_i' \notin dom(S_1)$ for $i \in [1..k]$:

   $$\langle S;\ e_1\rangle \longrightarrow \langle S_{\mathit{oldlocs}1}[l_1' \mapsto S_1(l_1)] \ldots [l_k' \mapsto S_1(l_k)];\ e_1'[l_1 := l_1'] \ldots [l_k := l_k']\rangle \neq \textbf{error},$$

   where $S_{\mathit{oldlocs}1}$ is defined as follows: $dom(S_{\mathit{oldlocs}1}) = dom(S)$, and for all $l \in dom(S_{\mathit{oldlocs}1})$, $S_{\mathit{oldlocs}1}(l) = S_1(l)$.
2. For all sets $\{l_{k+1}', \ldots, l_n'\}$ such that $l_i' \notin dom(S_2)$ for $i \in [k{+}1..n]$:

   $$\langle S;\ e_2\rangle \longrightarrow \langle S_{\mathit{oldlocs}2}[l_{k+1}' \mapsto S_2(l_{k+1})] \ldots [l_n' \mapsto S_2(l_n)];\ e_2'[l_{k+1} := l_{k+1}'] \ldots [l_n := l_n']\rangle \neq \textbf{error},$$

   where $S_{\mathit{oldlocs}2}$ is defined as follows: $dom(S_{\mathit{oldlocs}2}) = dom(S)$, and for all $l \in dom(S_{\mathit{oldlocs}2})$, $S_{\mathit{oldlocs}2}(l) = S_2(l)$.

Instantiate facts (1) and (2) with $\{l_1', \ldots, l_k'\}$ and $\{l_{k+1}', \ldots, l_n'\}$, respectively, where $\{l_1', \ldots, l_k'\} \cap \{l_{k+1}', \ldots, l_n'\} = \emptyset$ and $\{l_1', \ldots, l_k'\} \uplus \{l_{k+1}', \ldots, l_n'\} = \{l_1', \ldots, l_n'\}$.

Note that since $l_i' \notin dom(S_1 \sqcup_S S_2)$ for $i \in [1..n]$, it is also the case that $l_i' \notin dom(S_1)$ for $i \in [1..k]$ and that $l_i' \notin dom(S_2)$ for $i \in [k{+}1..n]$. Therefore, we have that:

1. $\langle S;\ e_1\rangle \longrightarrow \langle S_{\mathit{oldlocs}1}[l_1' \mapsto S_1(l_1)] \ldots [l_k' \mapsto S_1(l_k)];\ e_1'[l_1 := l_1'] \ldots [l_k := l_k']\rangle \neq \textbf{error}$, where $S_{\mathit{oldlocs}1}$ is defined as follows: $dom(S_{\mathit{oldlocs}1}) = dom(S)$, and for all $l \in dom(S_{\mathit{oldlocs}1})$, $S_{\mathit{oldlocs}1}(l) = S_1(l)$.
2. $\langle S;\ e_2\rangle \longrightarrow \langle S_{\mathit{oldlocs}2}[l_{k+1}' \mapsto S_2(l_{k+1})] \ldots [l_n' \mapsto S_2(l_n)];\ e_2'[l_{k+1} := l_{k+1}'] \ldots [l_n := l_n']\rangle \neq \textbf{error}$, where $S_{\mathit{oldlocs}2}$ is defined as follows: $dom(S_{\mathit{oldlocs}2}) = dom(S)$, and for all $l \in dom(S_{\mathit{oldlocs}2})$, $S_{\mathit{oldlocs}2}(l) = S_2(l)$.

Since

$$\langle S_{\mathit{oldlocs}1}[l_1' \mapsto S_1(l_1)] \ldots [l_k' \mapsto S_1(l_k)];\ e_1'[l_1 := l_1'] \ldots [l_k := l_k']\rangle \neq \textbf{error}$$

and

$$\langle S_{\mathit{oldlocs}2}[l_{k+1}' \mapsto S_2(l_{k+1})] \ldots [l_n' \mapsto S_2(l_n)];\ e_2'[l_{k+1} := l_{k+1}'] \ldots [l_n := l_n']\rangle \neq \textbf{error},$$

we have that

$$S_{\mathit{oldlocs}1}[l_1' \mapsto S_1(l_1)] \ldots [l_k' \mapsto S_1(l_k)] \neq \top_S$$

and

$$S_{\mathit{oldlocs}2}[l_{k+1}' \mapsto S_2(l_{k+1})] \ldots [l_n' \mapsto S_2(l_n)] \neq \top_S.$$

Further, since $S_1 \sqcup_S S_2 \neq \top_S$ (from the third premise of E-PARAPP) and since $\{l_1', \ldots, l_k'\} \cap \{l_{k+1}', \ldots, l_n'\} = \emptyset$, we have that

$$S_{\mathit{oldlocs}1}[l_1' \mapsto S_1(l_1)] \ldots [l_k' \mapsto S_1(l_k)] \sqcup_S S_{\mathit{oldlocs}2}[l_{k+1}' \mapsto S_2(l_{k+1})] \ldots [l_n' \mapsto S_2(l_n)] \neq \top_S.$$

Therefore, by E-PARAPP, we have that $\langle S;\ e_1\ e_2\rangle$ steps to

$$\langle S_{\mathit{oldlocs}1}[l_1' \mapsto S_1(l_1)] \ldots [l_k' \mapsto S_1(l_k)] \sqcup_S S_{\mathit{oldlocs}2}[l_{k+1}' \mapsto S_2(l_{k+1})] \ldots [l_n' \mapsto S_2(l_n)];\\ e_1'[l_1 := l_1'] \ldots [l_k := l_k']\ e_2'[l_{k+1} := l_{k+1}'] \ldots [l_n := l_n']\rangle.$$

It remains to show that the above configuration is equivalent to

$$\langle S_{\mathit{oldlocs}}[l_1' \mapsto (S_1 \sqcup_S S_2)(l_1)] \ldots [l_n' \mapsto (S_1 \sqcup_S S_2)(l_n)];\ (e_1'\ e_2')[l_1 := l_1'] \ldots [l_n := l_n']\rangle,$$

which we show as follows.

First, since $dom(S_{oldlocs1}) = dom(S_{oldlocs2}) = dom(S)$, we have that:

$$\begin{array}{rl} & (S_{oldlocs1}[l'_1 \mapsto S_1(l_1)] \ldots [l'_k \mapsto S_1(l_k)]) \sqcup_S (S_{oldlocs2}[l'_{k+1} \mapsto S_2(l_{k+1})] \ldots [l'_n \mapsto S_2(l_n)]) \\ = & (S_{oldlocs1} \sqcup_S S_{oldlocs2})[l'_1 \mapsto S_1(l_1)] \ldots [l'_k \mapsto S_1(l_k)][l'_{k+1} \mapsto S_2(l_{k+1})] \ldots [l'_n \mapsto S_2(l_n)]. \end{array}$$

Note that $dom(S_{oldlocs1} \sqcup_S S_{oldlocs2}) = dom(S)$. Therefore $dom(S_{oldlocs1} \sqcup_S S_{oldlocs2}) = dom(S_{oldlocs})$. Further, by Definition 3 we have that for all $l \in dom(S_{oldlocs1} \sqcup_S S_{oldlocs2})$, $(S_{oldlocs1} \sqcup_S S_{oldlocs2})(l) = S_{oldlocs1}(l) \sqcup S_{oldlocs2}(l) = S_1(l) \sqcup S_2(l) = (S_1 \sqcup_S S_2)(l) = S_{oldlocs}(l)$. Therefore, by Definition 9, we have that $S_{oldlocs1} \sqcup_S S_{oldlocs2} = S_{oldlocs}$. Continuing from above, then, we have that

$$\begin{array}{rl} & (S_{oldlocs1} \sqcup_S S_{oldlocs2})[l'_1 \mapsto S_1(l_1)] \ldots [l'_k \mapsto S_1(l_k)][l'_{k+1} \mapsto S_2(l_{k+1})] \ldots [l'_n \mapsto S_2(l_n)] \\ = & S_{oldlocs}[l'_1 \mapsto S_1(l_1)] \ldots [l'_k \mapsto S_1(l_k)][l'_{k+1} \mapsto S_2(l_{k+1})] \ldots [l'_n \mapsto S_2(l_n)]. \end{array}$$

Next, since $\{l_1, \ldots, l_k\} \cap \{l_{k+1}, \ldots, l_n\} = \emptyset$, we have that $l_i \notin dom(S_2)$ for $i \in [1..k]$ and $l_i \notin dom(S_1)$ for $i \in [k{+}1..n]$. Therefore $S_1(l_i) = (S_1 \sqcup_S S_2)(l_i)$ for $i \in [1..k]$ and $S_2(l_i) = (S_1 \sqcup_S S_2)(l_i)$ for $i \in [k{+}1..n]$, and so we have

$$\begin{array}{rl} & S_{oldlocs}[l'_1 \mapsto S_1(l_1)] \ldots [l'_k \mapsto S_1(l_k)][l'_{k+1} \mapsto S_2(l_{k+1})] \ldots [l'_n \mapsto S_2(l_n)] \\ = & S_{oldlocs}[l'_1 \mapsto (S_1 \sqcup_S S_2)(l_1)] \ldots [l'_k \mapsto (S_1 \sqcup_S S_2)(l_k)][l'_{k+1} \mapsto (S_1 \sqcup_S S_2)(l_{k+1})] \ldots [l'_n \mapsto (S_1 \sqcup_S S_2)(l_n)] \\ = & S_{oldlocs}[l'_1 \mapsto (S_1 \sqcup_S S_2)(l_1)] \ldots [l'_n \mapsto (S_1 \sqcup_S S_2)(l_n)]. \end{array}$$

Finally, we need to show that $(e'_1\ e'_2)[l_1 := l'_1] \ldots [l_n := l'_n]$ is equivalent to

$$e'_1[l_1 := l'_1] \ldots [l_k := l'_k]\ e'_2[l_{k+1} := l'_{k+1}] \ldots [l_n := l'_n].$$

Here, note that $l_{k+1}, \ldots, l_n$ cannot occur in e'_1 and $l_1, \ldots, l_k$ cannot occur in e'_2. Therefore the above expression is equivalent to

$$e'_1[l_1 := l'_1] \ldots [l_n := l'_n]\ e'_2[l_1 := l'_1] \ldots [l_n := l'_n],$$

which is equivalent to $(e'_1\ e'_2)[l_1 := l'_1] \ldots [l_n := l'_n]$. Therefore we have that

$$\langle S;\ e_1\ e_2\rangle \longhookrightarrow \langle S_{oldlocs}[l'_1 \mapsto (S_1 \sqcup_S S_2)(l_1)] \ldots [l'_n \mapsto (S_1 \sqcup_S S_2)(l_n)];\ (e'_1\ e'_2)[l_1 := l'_1] \ldots [l_n := l'_n]\rangle,$$

as we were required to show.

A.1.3 E-PUT-1

- E-PUT-1:

  Given: $\langle S;\ \mathsf{put}\ e_1\ e_2\rangle \longhookrightarrow \langle S_1;\ \mathsf{put}\ e'_1\ e_2\rangle$ and $\{l_1, \ldots, l_n\} = dom(S_1) - dom(S)$.

  To show: For all sets $\{l'_1, \ldots, l'_n\}$ such that $l'_i \notin dom(S_1)$ for $i \in [1..n]$,

  $$\langle S;\ \mathsf{put}\ e_1\ e_2\rangle \longhookrightarrow \langle S_{oldlocs}[l'_1 \mapsto S_1(l_1)] \ldots [l'_n \mapsto S_1(l_n)];\ (\mathsf{put}\ e'_1\ e_2)[l_1 := l'_1] \ldots [l_n := l'_n]\rangle,$$

  where $S_{oldlocs}$ is defined as follows: $dom(S_{oldlocs}) = dom(S)$, and for all $l \in dom(S_{oldlocs})$, $S_{oldlocs}(l) = S_1(l)$.

  Consider arbitrary $\{l'_1, \ldots, l'_n\}$ such that $l'_i \notin dom(S_1)$ for $i \in [1..n]$.

  From the premise of E-PUT-1, we have that $\langle S;\ e_1\rangle \longhookrightarrow \langle S_1;\ e'_1\rangle$. By IH we have that

  $$\langle S;\ e_1\rangle \longhookrightarrow \langle S_{oldlocs}[l'_1 \mapsto S_1(l_1)] \ldots [l'_n \mapsto S_1(l_n)];\ e'_1[l_1 := l'_1] \ldots [l_n := l'_n]\rangle.$$

  Therefore, by E-PUT-1 we have that:

  $$\langle S;\ \mathsf{put}\ e_1\ e_2\rangle \longhookrightarrow \langle S_{oldlocs}[l'_1 \mapsto S_1(l_1)] \ldots [l'_n \mapsto S_1(l_n)];\ \mathsf{put}\ e'_1[l_1 := l'_1] \ldots [l_n := l'_n]\ e_2\rangle.$$

Note that $l_1, \ldots l_n$ do not occur in e_2, for if some l_i occurred in e_2, then we would have $l_i \in dom(S)$, which contradicts $\{l_1, \ldots, l_n\} = dom(S_1) - dom(S)$. Therefore $e_2 = e_2[l_1 := l'_1] \ldots [l_n := l'_n]$, and so we have:

$$\langle S;\ \mathsf{put}\ e_1\ e_2\rangle \longhookrightarrow \langle S_{oldlocs}[l'_1 \mapsto S_1(l_1)] \ldots [l'_n \mapsto S_1(l_n)];\ \mathsf{put}\ e'_1[l_1 := l'_1] \ldots [l_n := l'_n]\ e_2[l_1 := l'_1] \ldots [l_n := l'_n]\rangle,$$

which is equivalent to

$$\langle S;\ \mathsf{put}\ e_1\ e_2\rangle \longhookrightarrow \langle S_{oldlocs}[l'_1 \mapsto S_1(l_1)] \ldots [l'_n \mapsto S_1(l_n)];\ (\mathsf{put}\ e'_1\ e_2)[l_1 := l'_1] \ldots [l_n := l'_n]\rangle,$$

as we were required to show.

A.1.4 E-PUT-2

- E-PUT-2:

  Given: $\langle S;\ \mathsf{put}\ e_1\ e_2\rangle \longhookrightarrow \langle S_2;\ \mathsf{put}\ e_1\ e'_2\rangle$ and $\{l_1, \ldots, l_n\} = dom(S_2) - dom(S)$.

  To show: For all sets $\{l'_1, \ldots, l'_n\}$ such that $l'_i \notin dom(S_2)$ for $i \in [1..n]$,

  $$\langle S;\ \mathsf{put}\ e_1\ e_2\rangle \longhookrightarrow \langle S_{oldlocs}[l'_1 \mapsto S_2(l_1)] \ldots [l'_n \mapsto S_2(l_n)];\ (\mathsf{put}\ e_1\ e'_2)[l_1 := l'_1] \ldots [l_n := l'_n]\rangle,$$

  where $S_{oldlocs}$ is defined as follows: $dom(S_{oldlocs}) = dom(S)$, and for all $l \in dom(S_{oldlocs})$, $S_{oldlocs}(l) = S_2(l)$.

  Consider arbitrary $\{l'_1, \ldots, l'_n\}$ such that $l'_i \notin dom(S_2)$ for $i \in [1..n]$.

  From the premise of E-PUT-2, we have that $\langle S;\ e_2\rangle \longhookrightarrow \langle S_2;\ e'_2\rangle$. By IH we have that

  $$\langle S;\ e_2\rangle \longhookrightarrow \langle S_{oldlocs}[l'_1 \mapsto S_2(l_1)] \ldots [l'_n \mapsto S_2(l_n)];\ e'_2[l_1 := l'_1] \ldots [l_n := l'_n]\rangle.$$

  Therefore, by E-PUT-2 we have that:

  $$\langle S;\ \mathsf{put}\ e_1\ e_2\rangle \longhookrightarrow \langle S_{oldlocs}[l'_1 \mapsto S_2(l_1)] \ldots [l'_n \mapsto S_2(l_n)];\ \mathsf{put}\ e_1\ e'_2[l_1 := l'_1] \ldots [l_n := l'_n]\rangle.$$

  Note that $l_1, \ldots l_n$ do not occur in e_1, for if some l_i occurred in e_1, then we would have $l_i \in dom(S)$, which contradicts $\{l_1, \ldots, l_n\} = dom(S_2) - dom(S)$. Therefore $e_1 = e_1[l_1 := l'_1] \ldots [l_n := l'_n]$, and so we have:

  $$\langle S;\ \mathsf{put}\ e_1\ e_2\rangle \longhookrightarrow \langle S_{oldlocs}[l'_1 \mapsto S_2(l_1)] \ldots [l'_n \mapsto S_2(l_n)];\ \mathsf{put}\ e_1[l_1 := l'_1] \ldots [l_n := l'_n]\ e'_2[l_1 := l'_1] \ldots [l_n := l'_n]\rangle,$$

  which is equivalent to

  $$\langle S;\ \mathsf{put}\ e_1\ e_2\rangle \longhookrightarrow \langle S_{oldlocs}[l'_1 \mapsto S_2(l_1)] \ldots [l'_n \mapsto S_2(l_n)];\ (\mathsf{put}\ e_1\ e'_2)[l_1 := l'_1] \ldots [l_n := l'_n]\rangle,$$

  as we were required to show.

A.1.5 E-PUTVAL

- E-PUTVAL:

  Given: $\langle S;\ \mathsf{put}\ l\ \{d_1\}\rangle \longhookrightarrow \langle S[l \mapsto d_1 \sqcup d_2];\ \{\}\rangle$ (note that no new locations are created during this transition, since we already have $l \in dom(S)$ from the $S(l) = d_1$ premise of E-PUTVAL).

  To show: $\langle S;\ \mathsf{put}\ l\ \{d_1\}\rangle \longhookrightarrow \langle S_{oldlocs};\ \{\}\rangle$, where $S_{oldlocs}$ is defined as follows: $dom(S_{oldlocs}) = dom(S)$, and for all $l' \in dom(S_{oldlocs})$, $S_{oldlocs}(l') = (S[l \mapsto d_1 \sqcup d_2])(l')$.

  Since $dom(S_{oldlocs}) = dom(S) = dom(S[l \mapsto d_1 \sqcup d_2])$ and since for all $l' \in dom(S_{oldlocs})$, $S_{oldlocs}(l') = (S[l \mapsto d_1 \sqcup d_2])(l')$, we have by Definition 9 that $S_{oldlocs} = S[l \mapsto d_1 \sqcup d_2]$, so the case is immediate by E-PUTVAL.

A.1.6 E-GET-1

- Case E-GET-1: Analogous to E-PUT-1.

A.1.7 E-GET-2

- Case E-GET-2: Analogous to E-PUT-2.

A.1.8 E-GETVAL

- E-GETVAL:

  Given: $\langle S;\ \mathsf{get}\ l\ Q\rangle \longhookrightarrow \langle S;\ \{d_1\}\rangle$.

  To show: $\langle S;\ \mathsf{get}\ l\ Q\rangle \longhookrightarrow \langle S_{oldlocs};\ \{d_1\}\rangle$, where $S_{oldlocs}$ is defined as follows: $dom(S_{oldlocs}) = dom(S)$, and for all $l' \in dom(S_{oldlocs})$, $S_{oldlocs}(l') = S(l')$.

  Since $dom(S_{oldlocs}) = dom(S)$ and since for all $l' \in dom(S_{oldlocs})$, $S_{oldlocs}(l') = S(l')$, we have by Definition 9 that $S_{oldlocs} = S$, so the case is immediate by E-GETVAL.

A.1.9 E-CONVERT

- E-CONVERT:

  Given: $\langle S;\ \mathsf{convert}\ e\rangle \longhookrightarrow \langle S';\ \mathsf{convert}\ e'\rangle$ and $\{l_1, \ldots, l_n\} = dom(S') - dom(S)$.

  To show: For all sets $\{l'_1, \ldots, l'_n\}$ such that $l'_i \notin dom(S')$ for $i \in [1..n]$,

  $$\langle S;\ \mathsf{convert}\ e\rangle \longhookrightarrow \langle S_{oldlocs}[l'_1 \mapsto S'(l_1)] \ldots [l'_n \mapsto S'(l_n)];\ (\mathsf{convert}\ e')[l_1 := l'_1] \ldots [l_n := l'_n]\rangle,$$

  where $S_{oldlocs}$ is defined as follows: $dom(S_{oldlocs}) = dom(S)$, and for all $l \in dom(S_{oldlocs})$, $S_{oldlocs}(l) = S'(l)$.

  Consider arbitrary $\{l'_1, \ldots, l'_n\}$ such that $l'_i \notin dom(S')$ for $i \in [1..n]$.

  From the premise of E-CONVERT, we have that $\langle S;\ e\rangle \longhookrightarrow \langle S';\ e'\rangle$. By IH we have that

  $$\langle S;\ e\rangle \longhookrightarrow \langle S_{oldlocs}[l'_1 \mapsto S'(l_1)] \ldots [l'_n \mapsto S'(l_n)];\ e'[l_1 := l'_1] \ldots [l_n := l'_n]\rangle.$$

  Therefore, by E-CONVERT we have that:

  $$\langle S;\ \mathsf{convert}\ e\rangle \longhookrightarrow \langle S_{oldlocs}[l'_1 \mapsto S'(l_1)] \ldots [l'_n \mapsto S'(l_n)];\ \mathsf{convert}\ e'[l_1 := l'_1] \ldots [l_n := l'_n]\rangle,$$

  which is equivalent to

  $$\langle S;\ \mathsf{convert}\ e\rangle \longhookrightarrow \langle S_{oldlocs}[l'_1 \mapsto S'(l_1)] \ldots [l'_n \mapsto S'(l_n)];\ (\mathsf{convert}\ e')[l_1 := l'_1] \ldots [l_n := l'_n]\rangle,$$

  as we were required to show.

A.1.10 E-CONVERTVAL

- E-CONVERTVAL:

  Given: $\langle S;\ \mathsf{convert}\ v\rangle \longhookrightarrow \langle S;\ \delta(v)\rangle$.

  To show: $\langle S;\ \mathsf{convert}\ v\rangle \longhookrightarrow \langle S_{oldlocs};\ \delta(v)\rangle$, where $S_{oldlocs}$ is defined as follows: $dom(S_{oldlocs}) = dom(S)$, and for all $l \in dom(S_{oldlocs})$, $S_{oldlocs}(l) = S(l)$.

  Since $dom(S_{oldlocs}) = dom(S)$ and since for all $l \in dom(S_{oldlocs})$, $S_{oldlocs}(l) = S(l)$, we have by Definition 9 that $S_{oldlocs} = S$, so the case is immediate by E-CONVERTVAL.

A.1.11 E-BETA

- E-BETA:

  Given: $\langle S;\ (\lambda x.\, e)\ v\rangle \longrightarrow \langle S;\ e[x := v]\rangle$.

  To show: $\langle S;\ (\lambda x.\, e)\ v\rangle \longrightarrow \langle S_{oldlocs};\ e[x := v]\rangle$, where $S_{oldlocs}$ is defined as follows: $dom(S_{oldlocs}) = dom(S)$, and for all $l \in dom(S_{oldlocs})$, $S_{oldlocs}(l) = S(l)$.

  Since $dom(S_{oldlocs}) = dom(S)$ and since for all $l \in dom(S_{oldlocs})$, $S_{oldlocs}(l) = S(l)$, we have by Definition 9 that $S_{oldlocs} = S$, so the case is immediate by E-BETA.

A.1.12 E-NEW

- E-NEW:

  Given: $\langle S;\ \mathsf{new}\rangle \longrightarrow \langle S[l \mapsto \bot];\ l\rangle$.

  To show: For all $l' \notin dom(S[l \mapsto \bot])$,

  $$\langle S;\ \mathsf{new}\rangle \longrightarrow \langle S_{oldlocs}[l' \mapsto \bot];\ l'\rangle,$$

  where $S_{oldlocs}$ is defined as follows: $dom(S_{oldlocs}) = dom(S)$, and for all $l'' \in dom(S_{oldlocs})$, $S_{oldlocs}(l'') = (S[l \mapsto \bot])(l'')$.

  We have from the definition of $S_{oldlocs}$ that $dom(S_{oldlocs}) = dom(S)$. Then, since the transition $\langle S;\ \mathsf{new}\rangle \longrightarrow \langle S[l \mapsto \bot];\ l\rangle$ does not update any existing bindings (since $l \notin dom(S)$ from the side condition of E-NEW), $S_{oldlocs}(l'') = S(l'')$ for all $l'' \in dom(S)$. So, by Definition 9, $S_{oldlocs} = S$.

  Therefore, we have only to show that $\langle S;\ \mathsf{new}\rangle \longrightarrow \langle S[l' \mapsto \bot];\ l'\rangle$, which is immediate by E-NEW since $l' \notin dom(S)$, which follows from $l' \notin dom(S[l \mapsto \bot])$.

□

A.2 Safety of *rename*

Lemma 1 characterizes the circumstances under which location renamings are safe. In the context of a transition $\langle S;\ e\rangle \longrightarrow \langle S';\ e'\rangle$, it characterizes the set of safe renamings of S' as those that can be expressed as a store $S_{oldlocs}$ (whose domain is equal to the domain of S, but whose codomain may differ from that of S because of updates to existing bindings), extended with bindings from each new location name to the value bound by the corresponding location name in S'.

The *rename* metafunction, on the other hand, is defined *algorithmically*: it takes a configuration $\langle S';\ e'\rangle$ and stores S'' and S as arguments and performs capture-avoiding substitution of new location names for the corresponding old ones in $\langle S';\ e'\rangle$, where the names to be replaced and the names they are to be replaced with are chosen based on S'' and S. In and of itself, the *rename* metafunction does nothing to ensure that the renaming it performs is "safe"—it is up to the caller to use it correctly. Lemma 2 shows that in the circumstances where we use *rename*—namely, the circumstances where a configuration $\langle S;\ e\rangle$ has stepped to $\langle S';\ e'\rangle$ and there exists a third store $S'' \neq \top_S$—then the renaming $rename(\langle S';\ e'\rangle, S'', S)$ meets the specification that Lemma 1 sets.

Lemma 2 (Safety of *rename*)**.** *If* $\langle S;\ e\rangle \longrightarrow \langle S';\ e'\rangle$ *(where* $\langle S';\ e'\rangle \neq$ **error***) and* $S'' \neq \top_S$*, then:*
$\langle S;\ e\rangle \longrightarrow rename(\langle S';\ e'\rangle, S'', S)$.

Proof. From the definition of *rename*, we have that:

$$\begin{aligned} rename(\langle S';\ e'\rangle, S'', S) &= \langle S';\ e'\rangle[l_1 := l'_1] \ldots [l_n := l'_n] \\ &= \langle S'[l_1 := l'_1] \ldots [l_n := l'_n];\ e'[l_1 := l'_1] \ldots [l_n := l'_n]\rangle, \end{aligned}$$

where:

- $\{l_1, \ldots, l_n\} = dom(S') - dom(S)$, and
- $\{l'_1, \ldots, l'_n\}$ is a set such that $l'_i \notin (dom(S') \cup dom(S''))$ for $i \in [1..n]$.

Therefore we need to show that $\langle S;\, e\rangle \longhookrightarrow \langle S'[l_1 := l'_1] \ldots [l_n := l'_n];\, e'[l_1 := l'_1] \ldots [l_n := l'_n]\rangle$, with $\{l_1, \ldots, l_n\}$ and $\{l'_1, \ldots, l'_n\}$ defined as above.

Applying Lemma 1 to $\langle S;\, e\rangle \longhookrightarrow \langle S';\, e'\rangle$ and $\{l_1, \ldots, l_n\}$, we have that for all sets $\{l'_1, \ldots, l'_n\}$ such that $l'_i \notin dom(S')$ for $i \in [1..n]$:

$$\begin{array}{l}\langle S;\, e\rangle \longhookrightarrow \\ \langle S_{oldlocs}[l'_1 \mapsto S'(l_1)] \ldots [l'_n \mapsto S'(l_n)];\, e'[l_1 := l'_1] \ldots [l_n := l'_n]\rangle,\end{array}$$

where $S_{oldlocs}$ is defined as follows: $dom(S_{oldlocs}) = dom(S)$, and for all $l \in dom(S_{oldlocs})$, $S_{oldlocs}(l) = S'(l)$.

Instantiate that result with $\{l'_1, \ldots, l'_n\}$. Note that since $l'_i \notin (dom(S') \cup dom(S''))$ for $i \in [1..n]$, we have that $l'_i \notin dom(S')$ for $i \in [1..n]$. Therefore we have that

$$\langle S;\, e\rangle \longhookrightarrow \langle S_{oldlocs}[l'_1 \mapsto S'(l_1)] \ldots [l'_n \mapsto S'(l_n)];\, e'[l_1 := l'_1] \ldots [l_n := l'_n]\rangle.$$

Since our goal is to show that

$$\langle S;\, e\rangle \longhookrightarrow \langle S'[l_1 := l'_1] \ldots [l_n := l'_n];\, e'[l_1 := l'_1] \ldots [l_n := l'_n]\rangle,$$

all that remains is to show that $S'[l_1 := l'_1] \ldots [l_n := l'_n]$ and $S_{oldlocs}[l'_1 \mapsto S'(l_1)] \ldots [l'_n \mapsto S'(l_n)]$ are equal.

By Definition 9, we have to show that:

- $dom(S'[l_1 := l'_1] \ldots [l_n := l'_n]) = dom(S_{oldlocs}[l'_1 \mapsto S'(l_1)] \ldots [l'_n \mapsto S'(l_n)])$, and
- for all $l'' \in dom(S'[l_1 := l'_1] \ldots [l_n := l'_n])$,

$$(S'[l_1 := l'_1] \ldots [l_n := l'_n])(l'') = (S_{oldlocs}[l'_1 \mapsto S'(l_1)] \ldots [l'_n \mapsto S'(l_n)])(l'').$$

For the first conjunct, $dom(S_{oldlocs}) = dom(S)$ by definition, so

$$\begin{array}{rl} dom(S_{oldlocs}[l'_1 \mapsto S'(l_1)] \ldots [l'_n \mapsto S'(l_n)]) & = dom(S) \cup \{l'_1, \ldots, l'_n\} \\ & = dom(S) \cup \{l_1, \ldots, l_n\}[l_1 := l'_1] \ldots [l_n := l'_n] \\ & = dom(S) \cup (dom(S') - dom(S))[l_1 := l'_1] \ldots [l_n := l'_n] \\ & = dom(S) \cup (dom(S')[l_1 := l'_1] \ldots [l_n := l'_n] - dom(S)) \\ & \quad \text{(since } l_i \notin dom(S)\text{)} \\ & = dom(S) \cup dom(S')[l_1 := l'_1] \ldots [l_n := l'_n] \\ & = dom(S) \cup dom(S'[l_1 := l'_1] \ldots [l_n := l'_n]) \\ & = dom(S'[l_1 := l'_1] \ldots [l_n := l'_n]) \\ & \quad \text{(since } dom(S) \subseteq dom(S') \text{ and } l_i \notin dom(S)\text{)}. \end{array}$$

For the second conjunct, there are two possibilities for l'':

- $l'' \in dom(S)$:

$$\begin{array}{rl} (S'[l_1 := l'_1] \ldots [l_n := l'_n])(l'') & = S'(l'') \\ & \quad \text{(since } l_i \notin dom(S)\text{)} \\ & = S_{oldlocs}(l'') \\ & \quad \text{(since } S_{oldlocs}(l) = S'(l) \text{ for all } l \in dom(S_{oldlocs}) = dom(S)\text{)} \\ & = (S_{oldlocs}[l'_1 \mapsto S'(l_1)] \ldots [l'_n \mapsto S'(l_n)])(l'') \\ & \quad \text{(since additional bindings are irrelevant to the lookup of } l''\text{)}. \end{array}$$

- $l'' \in \{l'_1, \ldots, l'_n\}$:

$$\begin{aligned}(S'[l_1 := l'_1] \ldots [l_n := l'_n])(l'') &= ([l'_1 \mapsto S'(l_1) \ldots l'_n \mapsto S(l_n)])(l'') \\ &= (S_{oldlocs}[l'_1 \mapsto S'(l_1)] \ldots [l'_n \mapsto S'(l_n)])(l'') \\ &\quad \text{(since additional bindings are irrelevant to the lookup of } l'').\end{aligned}$$

Therefore $S'[l_1 := l'_1] \ldots [l_n := l'_n]$ and $S_{oldlocs}[l'_1 \mapsto S'(l_1)] \ldots [l'_n \mapsto S'(l_n)]$ are equal. Since both their stores and expressions are equal, then, we have that

$$rename(\langle S'; e' \rangle, S'', S) = \langle S_{oldlocs}[l'_1 \mapsto S'(l_1)] \ldots [l'_n \mapsto S'(l_n)]\,;\, e'[l_1 := l'_1] \ldots [l_n := l'_n] \rangle,$$

as we were required to show. □

A.3 Independence

Lemma 3 (Independence)**.** *If $\langle S; e\rangle \longrightarrow \langle S'; e'\rangle$ (where $\langle S'; e'\rangle \neq$ **error**), then for all S'' such that S'' is non-conflicting with $\langle S; e\rangle \longrightarrow \langle S'; e'\rangle$ and $S' \sqcup_S S'' \neq \top_S$:*
$\langle S \sqcup_S S''; e\rangle \longrightarrow \langle S' \sqcup_S S''; e'\rangle$.

Proof. Consider arbitrary S'' such that S'' is non-conflicting with $\langle S; e\rangle \longrightarrow \langle S'; e'\rangle$ and $S' \sqcup_S S'' \neq \top_S$. To show: $\langle S \sqcup_S S''; e\rangle \longrightarrow \langle S' \sqcup_S S''; e'\rangle$.

The proof is by induction on the derivation of $\langle S; e\rangle \longrightarrow \langle S'; e'\rangle$, by cases on the last rule in the derivation. Since $\langle S'; e'\rangle \neq$ **error**, we only need to consider rules that step to non-**error** configurations. The requirement that S'' is non-conflicting with $\langle S; e\rangle \longrightarrow \langle S'; e'\rangle$ is only needed in the E-NEW case.

- Case E-REFL:

  Given: $\langle S; e\rangle \longrightarrow \langle S; e\rangle$, and $S \sqcup_S S'' \neq \top_S$.

  To show: $\langle S \sqcup_S S''; e\rangle \longrightarrow \langle S \sqcup_S S''; e\rangle$.

  The proof is immediate by E-REFL.

- Case E-PARAPP:

  Given: $\langle S; e_1\ e_2\rangle \longrightarrow \langle S_1^r \sqcup_S S_2; e_1'^r\ e_2'\rangle$, and $(S_1^r \sqcup_S S_2) \sqcup_S S'' \neq \top_S$.

  To show: $\langle S \sqcup_S S''; e_1\ e_2\rangle \longrightarrow \langle (S_1^r \sqcup_S S_2) \sqcup_S S''; e_1'^r\ e_2'\rangle$.

  From the premises of E-PARAPP, we have that $\langle S; e_1\rangle \longrightarrow \langle S_1; e_1'\rangle$, $\langle S; e_2\rangle \longrightarrow \langle S_2; e_2'\rangle$, and $\langle S_1^r; e_1'^r\rangle = rename(\langle S_1; e_1'\rangle, S_2, S)$.

  Since $(S_1^r \sqcup_S S_2) \sqcup_S S'' \neq \top_S$, we have that $S_2 \neq \top_S$. Therefore, since $\langle S; e_1\rangle \longrightarrow \langle S_1; e_1'\rangle$, by Lemma 2, we have that $\langle S; e_1\rangle \longrightarrow rename(\langle S_1; e_1'\rangle, S_2, S)$. Since $\langle S_1^r; e_1'^r\rangle = rename(\langle S_1; e_1'\rangle, S_2, S)$, we have that $\langle S; e_1\rangle \longrightarrow \langle S_1^r; e_1'^r\rangle$.

  Since $(S_1^r \sqcup_S S_2) \sqcup_S S'' \neq \top_S$, we know that $S_1^r \sqcup_S S'' \neq \top_S$ and $S_2 \sqcup_S S'' \neq \top_S$.

  Therefore, by IH, we have that $\langle S \sqcup_S S''; e_1\rangle \longrightarrow \langle S_1^r \sqcup_S S''; e_1'^r\rangle$ and that $\langle S \sqcup_S S''; e_2\rangle \longrightarrow \langle S_2 \sqcup_S S''; e_2'\rangle$.

  Since $(S_1^r \sqcup_S S_2) \sqcup_S S'' \neq \top_S$, we have that $(S_1^r \sqcup_S S'') \sqcup_S (S_2 \sqcup_S S'') \neq \top_S$.

  Therefore, by E-PARAPP we have that $\langle S \sqcup_S S''; e_1\ e_2\rangle \longrightarrow \langle (S_1^r \sqcup_S S'') \sqcup_S (S_2 \sqcup_S S''); e_1'^r\ e_2'\rangle$.

  Since $(S_1^r \sqcup_S S'') \sqcup_S (S_2 \sqcup_S S'')$ is equal to $(S_1^r \sqcup_S S_2) \sqcup_S S''$, we have that $\langle S \sqcup_S S''; e_1\ e_2\rangle \longrightarrow \langle (S_1^r \sqcup_S S_2) \sqcup_S S''; e_1'^r\ e_2'\rangle$, as required.

- Case E-PUT-1:

  Given: $\langle S; \mathsf{put}\ e_1\ e_2\rangle \longrightarrow \langle S_1; \mathsf{put}\ e_1'\ e_2\rangle$, and $S_1 \sqcup_S S'' \neq \top_S$.

  To show: $\langle S \sqcup_S S''; \mathsf{put}\ e_1\ e_2\rangle \longrightarrow \langle S_1 \sqcup_S S''; \mathsf{put}\ e_1'\ e_2\rangle$.

  From the premise of E-PUT-1, we have that $\langle S; e_1\rangle \longrightarrow \langle S_1; e_1'\rangle$. Since $S_1 \sqcup_S S'' \neq \top_S$, by IH we have that $\langle S \sqcup_S S''; e_1\rangle \longrightarrow \langle S_1 \sqcup_S S''; e_1'\rangle$.

  Therefore, by E-PUT-1 we have that $\langle S \sqcup_S S''; \mathsf{put}\ e_1\ e_2\rangle \longrightarrow \langle S_1 \sqcup_S S''; \mathsf{put}\ e_1'\ e_2\rangle$, as required.

- Case E-PUT-2:

  Given: $\langle S;\ \mathsf{put}\ e_1\ e_2\rangle \longrightarrow \langle S_2;\ \mathsf{put}\ e_1\ e_2'\rangle$, and $S_2 \sqcup_S S'' \neq \top_S$.

  To show: $\langle S \sqcup_S S'';\ \mathsf{put}\ e_1\ e_2\rangle \longrightarrow \langle S_2 \sqcup_S S'';\ \mathsf{put}\ e_1\ e_2'\rangle$.

  From the premise of E-PUT-2, we have that $\langle S;\ e_2\rangle \longrightarrow \langle S_2;\ e_2'\rangle$. Since $S_2 \sqcup_S S'' \neq \top_S$, by IH we have that $\langle S \sqcup_S S'';\ e_2\rangle \longrightarrow \langle S_2 \sqcup_S S'';\ e_2'\rangle$.

  Therefore, by E-PUT-2 we have that $\langle S \sqcup_S S'';\ \mathsf{put}\ e_1\ e_2\rangle \longrightarrow \langle S_2 \sqcup_S S'';\ \mathsf{put}\ e_1\ e_2'\rangle$, as required.

- Case E-GET-1: Analogous to E-PUT-1.

- Case E-GET-2: Analogous to E-PUT-2.

- Case E-CONVERT:

  Given: $\langle S;\ \mathsf{convert}\ e\rangle \longrightarrow \langle S';\ \mathsf{convert}\ e'\rangle$, and $S' \sqcup_S S'' \neq \top_S$.

  To show: $\langle S \sqcup_S S'';\ \mathsf{convert}\ e\rangle \longrightarrow \langle S' \sqcup_S S'';\ \mathsf{convert}\ e'\rangle$.

  From the premise of E-CONVERT, we have that $\langle S;\ e\rangle \longrightarrow \langle S';\ e'\rangle$. Since $S' \sqcup_S S'' \neq \top_S$, by IH we have that $\langle S \sqcup_S S'';\ e\rangle \longrightarrow \langle S' \sqcup_S S'';\ e'\rangle$.

  Therefore, by E-CONVERT we have that $\langle S \sqcup_S S'';\ \mathsf{convert}\ e\rangle \longrightarrow \langle S' \sqcup_S S'';\ \mathsf{convert}\ e'\rangle$, as required.

- Case E-BETA:

  Given: $\langle S;\ (\lambda x.\, e)\ v\rangle \longrightarrow \langle S;\ e[x := v]\rangle$, and $S \sqcup_S S'' \neq \top_S$.

  To show: $\langle S \sqcup_S S'';\ (\lambda x.\, e)\ v\rangle \longrightarrow \langle S \sqcup_S S'';\ e[x := v]\rangle$.

  Immediate by E-BETA.

- Case E-NEW:

  Given: $\langle S;\ \mathsf{new}\rangle \longrightarrow \langle S[l \mapsto \bot];\ l\rangle$ (where $l \notin dom(S)$), S'' is non-conflicting with $\langle S;\ \mathsf{new}\rangle \longrightarrow \langle S[l \mapsto \bot];\ l\rangle$, and $S[l \mapsto \bot] \sqcup_S S'' \neq \top$.

  To show: $\langle S \sqcup_S S'';\ \mathsf{new}\rangle \longrightarrow \langle S[l \mapsto \bot] \sqcup_S S'';\ l\rangle$.

  By E-NEW, we have that $\langle S \sqcup_S S'';\ \mathsf{new}\rangle \longrightarrow \langle (S \sqcup_S S'')[l' \mapsto \bot];\ l'\rangle$, where $l' \notin dom(S \sqcup_S S'')$. One of the following two possibilities must hold:

  - $l' = l$.

    In this case, we immediately have that $\langle S \sqcup_S S'';\ \mathsf{new}\rangle \longrightarrow \langle (S \sqcup_S S'')[l \mapsto \bot];\ l\rangle$.

  - $l' \neq l$.

    In this case, we apply Lemma 1 to $\langle S \sqcup_S S'';\ \mathsf{new}\rangle \longrightarrow \langle (S \sqcup_S S'')[l' \mapsto \bot];\ l'\rangle$ and $\{l'\}$. Therefore, for all l'' such that $l'' \notin dom((S \sqcup_S S'')[l' \mapsto \bot])$,

    $$\begin{array}{rcl} \langle S \sqcup_S S'';\ \mathsf{new}\rangle & \longrightarrow & \langle S_{oldlocs}[l'' \mapsto ((S \sqcup_S S'')[l' \mapsto \bot])(l')];\ l'[l' := l'']\rangle \\ & = & \langle S_{oldlocs}[l'' \mapsto \bot];\ l''\rangle, \end{array}$$

    where $S_{oldlocs}$ is defined as follows: $dom(S_{oldlocs}) = dom(S \sqcup_S S'')$, and for all $l \in dom(S_{oldlocs})$, $S_{oldlocs}(l) = ((S \sqcup_S S'')[l' \mapsto \bot])(l)$.
    Note that since $l' \notin dom(S \sqcup_S S'')$, $S_{oldlocs}(l) = (S \sqcup_S S'')(l)$ for all $l \in dom(S_{oldlocs})$. Therefore, since $dom(S_{oldlocs}) = dom(S \sqcup_S S'')$ and $S_{oldlocs}(l) = (S \sqcup_S S'')(l)$ for all $l \in dom(S_{oldlocs})$, the conditions of Definition 9 are satisfied, and $S_{oldlocs} = S \sqcup_S S''$.
    Therefore, we have that for all l'' such that $l'' \notin dom((S \sqcup_S S'')[l' \mapsto \bot])$,

    $$\langle S \sqcup_S S'';\ \mathsf{new}\rangle \longrightarrow \langle (S \sqcup_S S'')[l'' \mapsto \bot];\ l''\rangle.$$

    Instantiate the above with l. Since S'' is non-conflicting with $\langle S;\ \mathsf{new}\rangle \longrightarrow \langle S[l \mapsto \bot];\ l\rangle$, we know that $l \notin dom(S'')$, and we have from the side condition of E-NEW that $l \notin dom(S)$. Therefore $l \notin dom(S \sqcup_S S'')$, and since $l \neq l'$, we have that $l \notin dom((S \sqcup_S S'')[l' \mapsto \bot])$. Therefore, $\langle S \sqcup_S S'';\ \mathsf{new}\rangle \longrightarrow \langle (S \sqcup_S S'')[l \mapsto \bot];\ l\rangle$.

So, regardless of whether $l' = l$ or $l' \neq l$, we can conclude $\langle S \sqcup_S S''; \mathsf{new}\rangle \hookrightarrow \langle (S \sqcup_S S'')[l \mapsto \bot]; l\rangle$. Then, since S'' is non-conflicting with $\langle S; \mathsf{new}\rangle \hookrightarrow \langle S[l \mapsto \bot]; l\rangle$, we have that $l \notin dom(S'')$, and we have from the side condition of E-NEW that $l \notin dom(S)$. Therefore, we have:

$$\begin{aligned}(S \sqcup_S S'')[l \mapsto \bot] &= S[l \mapsto \bot] \sqcup_S S''[l \mapsto \bot] \\ &= S \sqcup_S [l \mapsto \bot] \sqcup_S S'' \sqcup_S [l \mapsto \bot] \\ &= S \sqcup_S [l \mapsto \bot] \sqcup_S S'' \\ &= S[l \mapsto \bot] \sqcup_S S''.\end{aligned}$$

Therefore $\langle S \sqcup_S S''; \mathsf{new}\rangle \hookrightarrow \langle S[l \mapsto \bot] \sqcup_S S''; l\rangle$, as we were required to show.

- Case E-PUTVAL:

  Given: $\langle S; \mathsf{put}\ l\ \{d_1\}\rangle \hookrightarrow \langle S[l \mapsto d_1 \sqcup d_2]; \{\}\rangle$, and $S[l \mapsto d_1 \sqcup d_2] \sqcup_S S'' \neq \top_S$.

  To show: $\langle S \sqcup_S S''; \mathsf{put}\ l\ \{d_1\}\rangle \hookrightarrow \langle S[l \mapsto d_1 \sqcup d_2] \sqcup_S S''; \{\}\rangle$.

  We have two cases:

  - $l \notin dom(S'')$.
    In this case, since $S(l) = d_2$ (from the premises of E-PUTVAL), we know that $(S \sqcup_S S'')(l) = d_2$. Therefore, by E-PUTVAL, $\langle S \sqcup_S S''; \mathsf{put}\ l\ \{d_1\}\rangle \hookrightarrow \langle S[l \mapsto d_1 \sqcup d_2] \sqcup_S S''; \{\}\rangle$, as we were required to show.
  - $l \in dom(S'')$.
    Since $S(l) = d_2$ (from the premises of E-PUTVAL), we know that $(S \sqcup_S S'')(l) = d_2'$, where $d_2 \sqsubseteq d_2'$.
    We show that $d_1 \sqcup d_2' \neq \top$, as follows:
    * Since $S[l \mapsto d_1 \sqcup d_2] \sqcup_S S'' \neq \top_S$, we know that $(S[l \mapsto d_1 \sqcup d_2])(l) \sqcup S''(l) \neq \top$.
    * Therefore, we have:

$$\begin{aligned}\top &\neq (S[l \mapsto d_1 \sqcup d_2])(l) \sqcup S''(l) \\ &= d_1 \sqcup d_2 \sqcup S''(l) && \text{(since } (S[l \mapsto d_1 \sqcup d_2])(l) = d_1 \sqcup d_2\text{)} \\ &= d_1 \sqcup S(l) \sqcup S''(l) && \text{(since } S(l) = d_2\text{)} \\ &= S(l) \sqcup S''(l) \sqcup d_1 \\ &= d_1 \sqcup (S \sqcup_S S'')(l) \\ &= d_1 \sqcup d_2'\end{aligned}$$

    Since $(S \sqcup_S S'')(l) = d_2'$ and $d_1 \sqcup d_2' \neq \top$, by E-PUTVAL we have that

$$\langle S \sqcup_S S''; \mathsf{put}\ l\ \{d_1\}\rangle \hookrightarrow \langle (S \sqcup_S S'')[l \mapsto d_1 \sqcup d_2']; \{\}\rangle.$$

    It remains to show that $(S \sqcup_S S'')[l \mapsto d_1 \sqcup d_2']$ is equal to $S[l \mapsto d_1 \sqcup d_2] \sqcup_S S''$.
    By Definition 9, to show that the stores are equal, we have two requirements to satisfy:
    * $dom((S \sqcup_S S'')[l \mapsto d_1 \sqcup d_2']) = dom(S[l \mapsto d_1 \sqcup d_2] \sqcup_S S'')$, and
    * for all l', $((S \sqcup_S S'')[l \mapsto d_1 \sqcup d_2'])(l') = (S[l \mapsto d_1 \sqcup d_2] \sqcup_S S'')(l')$.

    The first requirement follows from the observation that

$$\begin{aligned}dom((S \sqcup_S S'')[l \mapsto d_1 \sqcup d_2']) &= dom(S \sqcup_S S'') \cup \{l\} \\ &= dom(S) \cup \{l\} \cup dom(S'') \\ &= dom(S[l \mapsto d_1 \sqcup d_2]) \cup dom(S'') \\ &= dom(S[l \mapsto d_1 \sqcup d_2] \sqcup_S dom(S'')).\end{aligned}$$

    For the second requirement, we have two cases to consider:

* $l' \neq l$: In this case, bindings for l are irrelevant, so

$$\begin{aligned}((S \sqcup_S S'')[l \mapsto d_1 \sqcup d_2'])(l') &= (S \sqcup_S S'')(l') \\ &= (S[l \mapsto d_1 \sqcup d_2] \sqcup_S S'')(l'),\end{aligned}$$

  as required.
* $l' = l$: In this case, we have $((S \sqcup_S S'')[l \mapsto d_1 \sqcup d_2'])(l) = d_1 \sqcup d_2'$.
  We show that $(S[l \mapsto d_1 \sqcup d_2] \sqcup_S S'')(l)$ is also equal to $d_1 \sqcup d_2'$, as follows:

$$\begin{aligned}(S[l \mapsto d_1 \sqcup d_2] \sqcup_S S'')(l) &= (S[l \mapsto d_1 \sqcup d_2])(l) \sqcup S''(l) \\ &= d_1 \sqcup d_2 \sqcup S''(l) \\ &= d_1 \sqcup S(l) \sqcup S''(l) \\ &= d_1 \sqcup S(l) \sqcup S''(l) \\ &= d_1 \sqcup (S \sqcup_S S'')(l) \\ &= d_1 \sqcup d_2'\end{aligned}$$

Therefore we have that $\langle S \sqcup_S S''; \mathsf{put}\ l\ \{d_1\}\rangle \longhookrightarrow \langle S[l \mapsto d_1 \sqcup d_2] \sqcup_S S''; \{\}\rangle$, as required.

- Case E-GETVAL:

  Given: $\langle S; \mathsf{get}\ l\ Q\rangle \longhookrightarrow \langle S; \{d_1\}\rangle$, and $S \sqcup_S S'' \neq \top_S$.

  To show: $\langle S \sqcup_S S''; \mathsf{get}\ l\ Q\rangle \longhookrightarrow \langle S \sqcup_S S''; \{d_1\}\rangle$.

  Since $S(l) = d_2$ (from the premises of E-GETVAL), we know that $(S \sqcup_S S'')(l) = d_2'$, where $d_2 \sqsubseteq d_2'$.

  From the premises of E-GETVAL, we also have that $d_1 \in Q$ and that $d_1 \sqsubseteq d_2$. Since $d_2 \sqsubseteq d_2'$, we have that $d_1 \sqsubseteq d_2'$. Therefore, by E-GETVAL, we have that $\langle S \sqcup_S S''; \mathsf{get}\ l\ Q\rangle \longhookrightarrow \langle S \sqcup_S S''; \{d_1\}\rangle$, as we were required to show.

  (Intuitively, $\mathsf{get}\ l\ Q$ is asking if the value of $S(l)$ is at least the value of d_1. Once that is true, it will remain so under increasing S, since the value of $S(l)$ can only increase as S increases.)

- Case E-CONVERTVAL:

  Given: $\langle S; \mathsf{convert}\ Q\rangle \longhookrightarrow \langle S; \delta(Q)\rangle$, and $S \sqcup_S S'' \neq \top_S$.

  To show: $\langle S \sqcup_S S''; \mathsf{convert}\ Q\rangle \longhookrightarrow \langle S \sqcup_S S''; \delta(Q)\rangle$.

  Immediate by E-CONVERTVAL.

□

A.4 Clash

Lemma 4 (Clash)**.** *If $\langle S; e\rangle \longhookrightarrow \langle S'; e'\rangle$ (where $\langle S'; e'\rangle \neq$ **error**), then for all S'' such that S'' is non-conflicting with $\langle S; e\rangle \longhookrightarrow \langle S'; e'\rangle$ and $S' \sqcup_S S'' = \top_S$:*
$\langle S \sqcup_S S''; e\rangle \longhookrightarrow$ **error**.

Proof. Consider arbitrary S'' such that S'' is non-conflicting with $\langle S; e\rangle \longhookrightarrow \langle S'; e'\rangle$ and $S' \sqcup_S S'' \neq \top_S$. To show: $\langle S \sqcup_S S''; e\rangle \longhookrightarrow$ **error**.

The proof is by induction on the derivation of $\langle S; e\rangle \longhookrightarrow \langle S'; e'\rangle$, by cases on the last rule in the derivation. Since $\langle S'; e'\rangle \neq$ **error**, we only need to consider rules that step to non-**error** configurations.

- Case E-REFL:

  Given: $\langle S; e\rangle \longhookrightarrow \langle S; e\rangle$ and $S \sqcup_S S'' = \top_S$.

  To show: $\langle S \sqcup_S S''; e\rangle \longhookrightarrow$ **error**.

  Immediate by E-REFLERR since $\langle \top_S; e\rangle =$ **error**.

- Case E-PARAPP:

  Given: $\langle S;\ e_1\ e_2\rangle \longrightarrow \langle S_1^r \sqcup_S S_2;\ e_1'^r\ e_2'\rangle$, S'' is non-conflicting with $\langle S;\ e_1\ e_2\rangle \longrightarrow \langle S_1^r \sqcup_S S_2;\ e_1'^r\ e_2'\rangle$, and $(S_1^r \sqcup_S S_2) \sqcup_S S'' = \top_S$.

  To show: $\langle S \sqcup_S S'';\ e_1\ e_2\rangle \longrightarrow$ **error**.

  From the premises of E-PARAPP, we have that $\langle S;\ e_1\rangle \longrightarrow \langle S_1;\ e_1'\rangle$, $\langle S;\ e_2\rangle \longrightarrow \langle S_2;\ e_2'\rangle$, and $\langle S_1^r;\ e_1'^r\rangle = rename(\langle S_1;\ e_1'\rangle, S_2, S)$.

  At least one of the following situations must occur:

  - $S_1 \sqcup_S S'' = \top_S$. Then, by IH, $\langle S \sqcup_S S'';\ e_1\rangle \longrightarrow$ **error**. Therefore, by E-APPERR-1, we have that $\langle S \sqcup_S S'';\ e_1\ e_2\rangle \longrightarrow$ **error**, as required.
  - $S_2 \sqcup_S S'' = \top_S$. Then, by IH, $\langle S \sqcup_S S'';\ e_2\rangle \longrightarrow$ **error**. Therefore, by E-APPERR-2, we have that $\langle S \sqcup_S S'';\ e_1\ e_2\rangle \longrightarrow$ **error**, as required.
  - $S_1 \sqcup_S S'' \neq \top_S$ and $S_2 \sqcup_S S'' \neq \top_S$.

    In this case, since $S_2 \sqcup_S S'' \neq \top_S$, we have that $S_2 \neq \top_S$. Therefore, since $\langle S;\ e_1\rangle \longrightarrow \langle S_1;\ e_1'\rangle$, by Lemma 2, we have that $\langle S;\ e_1\rangle \longrightarrow rename(\langle S_1;\ e_1'\rangle, S_2, S)$. Since $\langle S_1^r;\ e_1'^r\rangle = rename(\langle S_1;\ e_1'\rangle, S_2, S)$, we have that $\langle S;\ e_1\rangle \longrightarrow \langle S_1^r;\ e_1'^r\rangle$.

    We have from premises that S'' is non-conflicting with $\langle S;\ e_1\ e_2\rangle \longrightarrow \langle S_1^r \sqcup_S S_2;\ e_1'^r\ e_2'\rangle$, so, by Definition 5, $(dom(S_1^r \sqcup_S S_2) - dom(S)) \cap dom(S'') = \emptyset$. Therefore $(dom(S_1^r) - dom(S)) \cap dom(S'') = \emptyset$, and so S'' is non-conflicting with $\langle S;\ e_1\rangle \longrightarrow \langle S_1^r;\ e_1'^r\rangle$. Likewise, $(dom(S_2) - dom(S)) \cap dom(S'') = \emptyset$, and so S'' is non-conflicting with $\langle S;\ e_2\rangle \longrightarrow \langle S_2;\ e_2'\rangle$.

    Therefore, by Lemma 3, we have that $\langle S \sqcup_S S'';\ e_1\rangle \longrightarrow \langle S_1^r \sqcup_S S'';\ e_1'^r\rangle$ and that $\langle S \sqcup_S S'';\ e_2\rangle \longrightarrow \langle S_2 \sqcup_S S'';\ e_2'\rangle$.

    Since $(S_1^r \sqcup_S S_2) \sqcup_S S'' = \top_S$, we have that $(S_1^r \sqcup_S S'') \sqcup_S (S_2 \sqcup_S S'') = \top_S$. Therefore, by E-PARAPPERR, we have that $\langle S \sqcup_S S'';\ e_1\ e_2\rangle \longrightarrow$ **error**, as required.

- Case E-PUT-1:

  Given: $\langle S;\ \mathsf{put}\ e_1\ e_2\rangle \longrightarrow \langle S_1;\ \mathsf{put}\ e_1'\ e_2\rangle$, and $S_1 \sqcup_S S'' = \top_S$.

  To show: $\langle S \sqcup_S S'';\ \mathsf{put}\ e_1\ e_2\rangle \longrightarrow$ **error**.

  From the premise of E-PUT-1, we have that $\langle S;\ e_1\rangle \longrightarrow \langle S_1;\ e_1'\rangle$.

  Since $S_1 \sqcup_S S'' = \top_S$, by IH, we have that $\langle S \sqcup_S S'';\ e_1\rangle \longrightarrow$ **error**.

  Therefore, by E-PUTERR-1 we have that $\langle S \sqcup_S S'';\ \mathsf{put}\ e_1\ e_2\rangle \longrightarrow$ **error**, as required.

- Case E-PUT-2:

  Given: $\langle S;\ \mathsf{put}\ e_1\ e_2\rangle \longrightarrow \langle S_2;\ \mathsf{put}\ e_1\ e_2'\rangle$, and $S_2 \sqcup_S S'' = \top_S$.

  To show: $\langle S \sqcup_S S'';\ \mathsf{put}\ e_1\ e_2\rangle \longrightarrow$ **error**.

  From the premise of E-PUT-2, we have that $\langle S;\ e_2\rangle \longrightarrow \langle S_2;\ e_2'\rangle$.

  Since $S_2 \sqcup_S S'' = \top_S$, by IH, we have that $\langle S \sqcup_S S'';\ e_2\rangle \longrightarrow$ **error**.

  Therefore, by E-PUTERR-2 we have that $\langle S \sqcup_S S'';\ \mathsf{put}\ e_1\ e_2\rangle \longrightarrow$ **error**, as required.

- Case E-GET-1: Analogous to E-PUT-1.

- Case E-GET-2: Analogous to E-PUT-2.

- Case E-CONVERT:

  Given: $\langle S;\ \mathsf{convert}\ e\rangle \longrightarrow \langle S';\ \mathsf{convert}\ e'\rangle$ and $S' \sqcup_S S'' = \top_S$.

  To show: $\langle S \sqcup_S S'';\ \mathsf{convert}\ e\rangle \longrightarrow$ **error**.

  From the premise of E-CONVERT, we have that $\langle S;\ e\rangle \longrightarrow \langle S';\ e'\rangle$.

Since $S' \sqcup_S S'' = \top_S$, by IH, we have that $\langle S \sqcup_S S''; e\rangle \hookrightarrow$ **error**.

Therefore, by E-CONVERTERR we have that $\langle S \sqcup_S S''; \mathsf{convert}\ e\rangle \hookrightarrow$ **error**, as required.

- Case E-BETA:

  Given: $\langle S; (\lambda x.\, e)\ v\rangle \hookrightarrow \langle S; e[x := v]\rangle$ and $S \sqcup_S S'' = \top_S$.

  To show: $\langle S \sqcup_S S''; (\lambda x.\, e)\ v\rangle \hookrightarrow$ **error**.

  Immediate by E-REFLERR since $\langle \top_S; (\lambda x.\, e)\ v\rangle =$ **error**.

- Case E-NEW:

  Given: $\langle S; \mathsf{new}\rangle \hookrightarrow \langle S[l \mapsto \bot]; l\rangle$ (where $l \notin dom(S)$), S'' is non-conflicting with $\langle S; \mathsf{new}\rangle \hookrightarrow \langle S[l \mapsto \bot]; l\rangle$, and $S[l \mapsto \bot] \sqcup_S S'' = \top_S$.

  To show: $\langle S \sqcup_S S''; \mathsf{new}\rangle \hookrightarrow$ **error**.

  By E-NEW, $\langle S \sqcup_S S''; \mathsf{new}\rangle \hookrightarrow \langle (S \sqcup_S S'')[l' \mapsto \bot]; l'\rangle$, where $l' \notin dom(S \sqcup_S S'')$. One of the following two possibilities must hold:

  - $l' = l$.

    In this case, we immediately have that $\langle S \sqcup_S S''; \mathsf{new}\rangle \hookrightarrow \langle (S \sqcup_S S'')[l \mapsto \bot]; l\rangle$.

  - $l' \neq l$.

    In this case, we apply Lemma 1 to $\langle S \sqcup_S S''; \mathsf{new}\rangle \hookrightarrow \langle (S \sqcup_S S'')[l' \mapsto \bot]; l'\rangle$ and $\{l'\}$. Therefore, for all l'' such that $l'' \notin dom((S \sqcup_S S'')[l' \mapsto \bot])$,

    $$\begin{array}{rcl} \langle S \sqcup_S S''; \mathsf{new}\rangle & \hookrightarrow & \langle S_{oldlocs}[l'' \mapsto ((S \sqcup_S S'')[l' \mapsto \bot])(l')]; l'[l' := l'']\rangle \\ & = & \langle S_{oldlocs}[l'' \mapsto \bot]; l''\rangle, \end{array}$$

    where $S_{oldlocs}$ is defined as follows: $dom(S_{oldlocs}) = dom(S \sqcup_S S'')$, and for all $l \in dom(S_{oldlocs})$, $S_{oldlocs}(l) = ((S \sqcup_S S'')[l' \mapsto \bot])(l)$.

    Note that since $l' \notin dom(S \sqcup_S S'')$, $S_{oldlocs}(l) = (S \sqcup_S S'')(l)$ for all $l \in dom(S_{oldlocs})$. Therefore, since $dom(S_{oldlocs}) = dom(S \sqcup_S S'')$ and $S_{oldlocs}(l) = (S \sqcup_S S'')(l)$ for all $l \in dom(S_{oldlocs})$, the conditions of Definition 9 are satisfied, and $S_{oldlocs} = S \sqcup_S S''$.

    Therefore, we have that for all l'' such that $l'' \notin dom((S \sqcup_S S'')[l' \mapsto \bot])$,

    $$\langle S \sqcup_S S''; \mathsf{new}\rangle \hookrightarrow \langle (S \sqcup_S S'')[l'' \mapsto \bot]; l''\rangle.$$

    Instantiate the above with l. Since S'' is non-conflicting with $\langle S; \mathsf{new}\rangle \hookrightarrow \langle S[l \mapsto \bot]; l\rangle$, we know that $l \notin dom(S'')$, and we have from the side condition of E-NEW that $l \notin dom(S)$. Therefore $l \notin dom(S \sqcup_S S'')$, and since $l \neq l'$, we have that $l \notin dom((S \sqcup_S S'')[l' \mapsto \bot])$. Therefore, $\langle S \sqcup_S S''; \mathsf{new}\rangle \hookrightarrow \langle (S \sqcup_S S'')[l \mapsto \bot]; l\rangle$.

  So, regardless of whether $l' = l$ or $l' \neq l$, we can conclude $\langle S \sqcup_S S''; \mathsf{new}\rangle \hookrightarrow \langle (S \sqcup_S S'')[l \mapsto \bot]; l\rangle$. Then, since S'' is non-conflicting with $\langle S; \mathsf{new}\rangle \hookrightarrow \langle S[l \mapsto \bot]; l\rangle$, we have that $l \notin dom(S'')$, and we have from the side condition of E-NEW that $l \notin dom(S)$. Therefore, we have:

  $$\begin{aligned} (S \sqcup_S S'')[l \mapsto \bot] &= S[l \mapsto \bot] \sqcup_S S''[l \mapsto \bot] \\ &= S \sqcup_S [l \mapsto \bot] \sqcup_S S'' \sqcup_S [l \mapsto \bot] \\ &= S \sqcup_S [l \mapsto \bot] \sqcup_S S'' \\ &= S[l \mapsto \bot] \sqcup_S S'' \\ &= \top_S. \end{aligned}$$

  Therefore, since $\langle \top_S; l\rangle =$ **error**, we have that $\langle S \sqcup_S S''; \mathsf{new}\rangle \hookrightarrow$ **error**, as we were required to show.

- Case E-PUTVAL:

  Given: $\langle S;\ \mathsf{put}\ l\ \{d_1\}\rangle \longrightarrow \langle S[l \mapsto d_1 \sqcup d_2];\ \{\}\rangle$ and $S[l \mapsto d_1 \sqcup d_2] \sqcup_S S'' = \top_S$.

  To show: $\langle S \sqcup_S S'';\ \mathsf{put}\ l\ \{d_1\}\rangle \longrightarrow \textbf{error}$.

  One of the following must be the case:

  - $S \sqcup_S S'' = \top_S$. In this case, the proof is immediate by E-REFLERR, since $\langle \top_S;\ \mathsf{put}\ l\ \{d_1\}\rangle = \textbf{error}$.
  - $S \sqcup_S S'' \neq \top_S$. In this case, we proceed as follows:
    Since $S(l) = d_2$ (from the premises of E-PUTVAL), we know that $(S \sqcup_S S'')(l) = d_2'$, where $d_2 \sqsubseteq d_2'$.
    We show that $d_1 \sqcup d_2' = \top$, as follows:
    - Since $S[l \mapsto d_1 \sqcup d_2] \sqcup_S S'' = \top_S$, we know that there exists some $l' \in dom(S[l \mapsto d_1 \sqcup d_2]) \cap dom(S'')$ such that $(S[l \mapsto d_1 \sqcup d_2])(l') \sqcup S''(l') = \top$.
    - If $l' \neq l$, then $(S[l \mapsto d_1 \sqcup d_2])(l')$ would be equal to $S(l')$, because the binding for l would be irrelevant. We would then have $(S[l \mapsto d_1 \sqcup d_2])(l') \sqcup S''(l') = S(l') \sqcup S''(l') = \top$, a contradiction since $S \sqcup_S S'' \neq \top_S$. Therefore it must be the case that $l' = l$, so we have that $(S[l \mapsto d_1 \sqcup d_2])(l) \sqcup S''(l) = \top$.
    - Therefore, we have:

$$
\begin{aligned}
\top &= (S[l \mapsto d_1 \sqcup d_2])(l) \sqcup S''(l) \\
&= d_1 \sqcup d_2 \sqcup S''(l) && \text{(since } (S[l \mapsto d_1 \sqcup d_2])(l) = d_1 \sqcup d_2\text{)} \\
&= d_1 \sqcup S(l) \sqcup S''(l) && \text{(since } S(l) = d_2\text{)} \\
&= d_1 \sqcup S(l) \sqcup S''(l) \\
&= d_1 \sqcup (S \sqcup_S S'')(l) \\
&= d_1 \sqcup d_2'
\end{aligned}
$$

    Therefore, since $(S \sqcup_S S'')(l) = d_2'$ and $d_1 \sqcup d_2' = \top$, by E-PUTVALERR we have that $\langle S \sqcup_S S'';\ \mathsf{put}\ l\ \{d_1\}\rangle \longrightarrow \textbf{error}$, as required.

- Case E-GETVAL:

  Given: $\langle S;\ \mathsf{get}\ l\ Q\rangle \longrightarrow \langle S;\ \{d_1\}\rangle$ and $S \sqcup_S S'' = \top_S$.

  To show: $\langle S \sqcup_S S'';\ \mathsf{get}\ l\ Q\rangle \longrightarrow \textbf{error}$.

  Immediate by E-REFLERR since $\langle \top_S;\ \mathsf{get}\ l\ Q\rangle = \textbf{error}$.

- Case E-CONVERTVAL:

  Given: $\langle S;\ \mathsf{convert}\ Q\rangle \longrightarrow \langle S;\ \delta(Q)\rangle$ and $S \sqcup_S S'' = \top_S$.

  To show: $\langle S \sqcup_S S'';\ \mathsf{convert}\ Q\rangle \longrightarrow \textbf{error}$.

  Immediate by E-REFLERR since $\langle \top_S;\ \mathsf{convert}\ Q\rangle = \textbf{error}$.

□

A.5 Error Preservation

Lemma 5 (Error Preservation)**.** *If* $\langle S;\ e\rangle \longrightarrow \textbf{error}$ *and* $S \sqsubseteq_S S'$*, then* $\langle S';\ e\rangle \longrightarrow \textbf{error}$.

Proof. Let $S \sqsubseteq_S S'$ and proceed by induction on the derivation of $\langle S;\ e\rangle \longrightarrow \textbf{error}$. We only need to consider the reduction rules that step to **error**.

- Case E-APPERR-1:

  Given: $\langle S;\ e_1\ e_2\rangle \hookrightarrow$ **error**.

  To show: $\langle S';\ e_1\ e_2\rangle \hookrightarrow$ **error**.

  From the premise of E-APPERR-1 we have that $\langle S;\ e_1\rangle \hookrightarrow$ **error**. Since $S \sqsubseteq_S S'$, we have by the induction hypothesis that $\langle S';\ e_1\rangle \hookrightarrow$ **error**. Therefore, by E-APPERR-1, we have that $\langle S';\ e_1\ e_2\rangle \hookrightarrow$ **error**, as required.

- Case E-APPERR-2: Analogous to E-APPERR-1.

- Case E-PARAPPERR:

  (NB: For simplicity, we elide renaming throughout this case and assume that configurations can be renamed to meet non-conflicting requirements.)

  Given: $\langle S;\ e_1\ e_2\rangle \hookrightarrow$ **error**.

  To show: $\langle S';\ e_1\ e_2\rangle \hookrightarrow$ **error**.

  From the premises of E-PARAPPERR, we have that:

  - $\langle S;\ e_1\rangle \hookrightarrow \langle S_1;\ e_1'\rangle$,
  - $\langle S;\ e_2\rangle \hookrightarrow \langle S_2;\ e_2'\rangle$, and
  - $S_1 \sqcup_S S_2 = \top_S$.

  At least one of the following situations must occur:

  - $S_1 \sqcup_S S' = \top_S$.
    In this case, since $\langle S;\ e_1\rangle \hookrightarrow \langle S_1;\ e_1'\rangle$, and $S_1 \sqcup_S S' = \top_S$, we have from Lemma 4 that $\langle S \sqcup_S S';\ e_1\rangle \hookrightarrow$ **error**. Since $S \sqsubseteq_S S'$, $S \sqcup_S S' = S'$, so we have that $\langle S';\ e_1\rangle \hookrightarrow$ **error**. Therefore, by E-APPERR-1, we have that $\langle S';\ e_1\ e_2\rangle \hookrightarrow$ **error**, as required.
  - $S_2 \sqcup_S S' = \top_S$.
    In this case, since $\langle S;\ e_2\rangle \hookrightarrow \langle S_2;\ e_2'\rangle$, and $S_2 \sqcup_S S' = \top_S$, we have from Lemma 4 that $\langle S \sqcup_S S';\ e_2\rangle \hookrightarrow$ **error**. Since $S \sqsubseteq_S S'$, $S \sqcup_S S' = S'$, so we have that $\langle S';\ e_2\rangle \hookrightarrow$ **error**. Therefore, by E-APPERR-2, we have that $\langle S';\ e_1\ e_2\rangle \hookrightarrow$ **error**, as required.
  - $S_1 \sqcup_S S' \neq \top_S$ and $S_2 \sqcup_S S' \neq \top_S$.
    In this case, since $\langle S;\ e_1\rangle \hookrightarrow \langle S_1;\ e_1'\rangle$ and $S_1 \sqcup_S S' \neq \top_S$, we have from Lemma 3 that $\langle S \sqcup_S S';\ e_1\rangle \hookrightarrow \langle S_1 \sqcup_S S';\ e_1'\rangle$. Likewise, since $\langle S;\ e_2\rangle \hookrightarrow \langle S_2;\ e_2'\rangle$ and $S_2 \sqcup_S S' \neq \top_S$, we have from Lemma 3 that $\langle S \sqcup_S S';\ e_2\rangle \hookrightarrow \langle S_2 \sqcup_S S';\ e_2'\rangle$.
    Since $S \sqsubseteq_S S'$, $S \sqcup_S S' = S'$, so we have that $\langle S';\ e_1\rangle \hookrightarrow \langle S_1 \sqcup_S S';\ e_1'\rangle$ and $\langle S';\ e_2\rangle \hookrightarrow \langle S_2 \sqcup_S S';\ e_2'\rangle$.
    Since $S_1 \sqcup_S S_2 = \top_S$, we have that $S_1 \sqcup_S S' \sqcup_S S_2 \sqcup_S S' = \top_S$. Therefore, by E-PARAPPERR, we have that $\langle S';\ e_1\ e_2\rangle \hookrightarrow$ **error**, as desired.

- Case E-PUTERR-1: Analogous to E-APPERR-1.

- Case E-PUTERR-2: Analogous to E-APPERR-1.

- Case E-GETERR-1: Analogous to E-APPERR-1.

- Case E-GETERR-2: Analogous to E-APPERR-1.

- Case E-CONVERTERR:

  Given: $\langle S;\ \mathsf{convert}\ e\rangle \hookrightarrow$ **error**.

  To show: $\langle S';\ \mathsf{convert}\ e\rangle \hookrightarrow$ **error**.

From the premise of E-CONVERTERR we have that $\langle S;\ e\rangle \hookrightarrow \textbf{error}$. Since $S \sqsubseteq_S S'$, we have by the induction hypothesis that $\langle S';\ e\rangle \hookrightarrow \textbf{error}$. Therefore, by E-CONVERTERR, we have that $\langle S';\ \mathsf{convert}\ e\rangle \hookrightarrow \textbf{error}$, as required.

- Case E-PUTVALERR:

  Given: $\langle S;\ \mathsf{put}\ l\ \{d_1\}\rangle \hookrightarrow \textbf{error}$.

  To show: $\langle S';\ \mathsf{put}\ l\ \{d_1\}\rangle \hookrightarrow \textbf{error}$.

  Since $S(l) = d_2$ (from the first premise of E-PUTVALERR), we know that $S'(l) = d_2'$, where $d_2 \sqsubseteq d_2'$. Since $d_1 \sqcup d_2 = \top$, we have that $d_1 \sqcup d_2' = \top$. Therefore, by E-PUTVALERR, $\langle S';\ \mathsf{put}\ l\ \{d_1\}\rangle \hookrightarrow \textbf{error}$, as required.

□

A.6 Diamond

Lemma 6 (Diamond)**.** *If $\sigma \hookrightarrow \sigma_a$ and $\sigma \hookrightarrow \sigma_b$, then there exists σ_c such that either:*

- *$\sigma_a \hookrightarrow \sigma_c$ and $\sigma_b \hookrightarrow \sigma_c$, or*
- *there exists a safe renaming σ_b' of σ_b with respect to $\sigma \hookrightarrow \sigma_b$, such that $\sigma_a \hookrightarrow \sigma_c$ and $\sigma_b' \hookrightarrow \sigma_c$.*

Proof. By induction on the derivation of $\sigma \hookrightarrow \sigma_a$, by cases on the last rule in the derivation. For all cases except the E-NEW case, we prove the first disjunct; in the E-NEW case, we prove the second disjunct.

Where necessary, we use a "left/right" naming convention for subcases of the proof. For instance, the subcase E-PARAPP/E-REFL is the case where the last rule in the derivation of $\sigma \hookrightarrow \sigma_a$ (the "left" side of the diamond) is E-PARAPP and the last rule in the derivation of $\sigma \hookrightarrow \sigma_b$ (the "right" side of the diamond) is E-REFL.

A.6.1 E-REFL

- E-REFL: $\sigma = \langle S;\ e\rangle$, and $\sigma_a = \langle S;\ e\rangle$.

  Given:

  - $\langle S;\ e\rangle \hookrightarrow \langle S;\ e\rangle$, and
  - $\langle S;\ e\rangle \hookrightarrow \sigma_b$.

  To show: There exists σ_c such that

  - $\langle S;\ e\rangle \hookrightarrow \sigma_c$, and
  - $\sigma_b \hookrightarrow \sigma_c$.

  For all subcases E-REFL/*, choose $\sigma_c = \sigma_b$.

  To show:

  - $\langle S;\ e\rangle \hookrightarrow \sigma_b$, which is immediate from our assumptions, above, and
  - $\sigma_b \hookrightarrow \sigma_b$, which follows from either E-REFL or E-REFLERR.

A.6.2 E-PARAPP

- E-PARAPP: $\sigma = \langle S;\ e_1\ e_2\rangle$, and $\sigma_a = \langle S_1 \sqcup_S S_2;\ e_1'\ e_2'\rangle$.

  (NB: For simplicity, we elide renaming throughout this case and assume that configurations can be renamed to meet non-conflicting requirements.)

  Given:

– $\langle S;\ e_1\ e_2\rangle \hookrightarrow \langle S_1 \sqcup_S S_2;\ e_1'\ e_2'\rangle$, and
– $\langle S;\ e_1\ e_2\rangle \hookrightarrow \sigma_b$.

To show: There exists σ_c such that

– $\langle S_1 \sqcup_S S_2;\ e_1'\ e_2'\rangle \hookrightarrow \sigma_c$, and
– $\sigma_b \hookrightarrow \sigma_c$.

From the premises of E-ParApp, we have the following facts:

– $\langle S;\ e_1\rangle \hookrightarrow \langle S_1;\ e_1'\rangle$;
– $\langle S;\ e_2\rangle \hookrightarrow \langle S_2;\ e_2'\rangle$; and
– $S_1 \sqcup_S S_2 \neq \top_S$.

We proceed by subcases, on the last rule in the derivation of $\langle S;\ e_1\ e_2\rangle \hookrightarrow \sigma_b$. By the operational semantics, there are six possibilities: E-ParApp/E-Refl, E-ParApp/E-ParApp, E-ParApp/E-Beta, E-ParApp/E-AppErr-1, E-ParApp/E-AppErr-2, and E-ParApp/E-ParAppErr.

– E-ParApp/E-Refl:
  Analogous to the E-Refl/E-ParApp case, with σ_a and σ_b reversed.
– E-ParApp/E-ParApp:
  In this case, we have the following facts:
  * $\sigma_b = \langle S_{b_1} \sqcup_S S_{b_2};\ e_{b_1}\ e_{b_2}\rangle$,
  * $\langle S;\ e_1\rangle \hookrightarrow \langle S_{b_1};\ e_{b_1}\rangle$,
  * $\langle S;\ e_2\rangle \hookrightarrow \langle S_{b_2};\ e_{b_2}\rangle$, and
  * $S_{b_1} \sqcup_S S_{b_2} \neq \top_S$.

  Since $\langle S;\ e_1\rangle \hookrightarrow \langle S_1;\ e_1'\rangle$ and $\langle S;\ e_1\rangle \hookrightarrow \langle S_{b_1};\ e_{b_1}\rangle$ (from above), we have by IH that there exists σ_{c_1} such that $\langle S_1;\ e_1'\rangle \hookrightarrow \sigma_{c_1}$ and $\langle S_{b_1};\ e_{b_1}\rangle \hookrightarrow \sigma_{c_1}$. Either σ_{c_1} is **error**, or it is some non-**error** configuration $\langle S_{c_1};\ e_{c_1}\rangle$.
  Similarly, since $\langle S;\ e_2\rangle \hookrightarrow \langle S_2;\ e_2'\rangle$ and $\langle S;\ e_2\rangle \hookrightarrow \langle S_{b_2};\ e_{b_2}\rangle$, we have by IH that there exists σ_{c_2} such that $\langle S_2;\ e_2'\rangle \hookrightarrow \sigma_{c_2}$ and $\langle S_{b_2};\ e_{b_2}\rangle \hookrightarrow \sigma_{c_2}$. Either σ_{c_2} is **error**, or it is some non-**error** configuration $\langle S_{c_2};\ e_{c_2}\rangle$.
  We're required to show that there exists σ_c such that
  * $\langle S_1 \sqcup_S S_2;\ e_1'\ e_2'\rangle \hookrightarrow \sigma_c$, and
  * $\langle S_{b_1} \sqcup_S S_{b_2};\ e_{b_1}\ e_{b_2}\rangle \hookrightarrow \sigma_c$.

  We consider possibilities 1, 2, and 3, at least one of which must hold. We will show that in 1, 2, 3a, 3b, and 3c, $\sigma_c =$ **error**, and in 3d, $\sigma_c = \langle S_{c_1} \sqcup_S S_{c_2};\ e_{c_1}\ e_{c_2}\rangle$.

  1. $\sigma_{c_1} =$ **error**.
     Then, since $\langle S_1;\ e_1'\rangle \hookrightarrow$ **error**, we have by E-AppErr-1 that $\langle S_1;\ e_1'\ e_2'\rangle \hookrightarrow$ **error**. Then, by Lemma 5, $\langle S_1 \sqcup_S S_2;\ e_1'\ e_2'\rangle \hookrightarrow$ **error**. Likewise, since $\langle S_{b_1};\ e_{b_1}\rangle \hookrightarrow$ **error**, we have by E-AppErr-1 that $\langle S_{b_1};\ e_{b_1}\ e_{b_2}\rangle \hookrightarrow$ **error**, and again by Lemma 5, we have that $\langle S_{b_1} \sqcup_S S_{b_2};\ e_{b_1}\ e_{b_2}\rangle \hookrightarrow$ **error**. Therefore $\sigma_c =$ **error**.
  2. $\sigma_{c_2} =$ **error**.
     An argument analogous to the above applies, this time appealing to E-AppErr-2. Therefore $\sigma_c =$ **error**.
  3. $\sigma_{c_1} \neq$ **error** and $\sigma_{c_2} \neq$ **error**.
     Then $\sigma_{c_1} = \langle S_{c_1};\ e_{c_1}\rangle$, and $\sigma_{c_2} = \langle S_{c_2};\ e_{c_2}\rangle$.
     At least one of the following four possibilities must hold:

(a) $S_{c_1} \sqcup_S S_2 = \top_S$.
Then, since $\langle S_1;\, e_1' \rangle \hookrightarrow \langle S_{c_1};\, e_{c_1} \rangle$, by Lemma 4 we have that $\langle S_1 \sqcup_S S_2;\, e_1' \rangle \hookrightarrow$ **error**. Therefore, by E-APPERR-1, $\langle S_1 \sqcup_S S_2;\, e_1'\ e_2' \rangle \hookrightarrow$ **error**.
Next, we show that $\langle S_{b_1} \sqcup_S S_{b_2};\, e_{b_1}\ e_{b_2} \rangle$ must step to **error**, as well. At least one of the following three possibilities must hold:

i. $S_{c_1} \sqcup_S S_{b_2} = \top_S$.
Then, since $\langle S_{b_1};\, e_{b_1} \rangle \hookrightarrow \langle S_{c_1};\, e_{c_1} \rangle$, by Lemma 4 we have that $\langle S_{b_1} \sqcup_S S_{b_2};\, e_{b_1} \rangle \hookrightarrow$ **error**. Therefore, by E-APPERR-1, $\langle S_{b_1} \sqcup_S S_{b_2};\, e_{b_1}\ e_{b_2} \rangle \hookrightarrow$ **error**.

ii. $S_{b_1} \sqcup_S S_{c_2} = \top_S$.
Then, since $\langle S_{b_2};\, e_{b_2} \rangle \hookrightarrow \langle S_{c_2};\, e_{c_2} \rangle$, by Lemma 4 we have that $\langle S_{b_1} \sqcup_S S_{b_2};\, e_{b_2} \rangle \hookrightarrow$ **error**. Therefore, by E-APPERR-2, $\langle S_{b_1} \sqcup_S S_{b_2};\, e_{b_1}\ e_{b_2} \rangle \hookrightarrow$ **error**.

iii. $S_{c_1} \sqcup_S S_{b_2} \neq \top_S$ and $S_{b_1} \sqcup_S S_{c_2} \neq \top_S$.
Then, since $\langle S_{b_1};\, e_{b_1} \rangle \hookrightarrow \langle S_{c_1};\, e_{c_1} \rangle$ and $\langle S_{b_2};\, e_{b_2} \rangle \hookrightarrow \langle S_{c_2};\, e_{c_2} \rangle$, we have by Lemma 3 that $\langle S_{b_1} \sqcup_S S_{b_2};\, e_{b_1} \rangle \hookrightarrow \langle S_{c_1} \sqcup_S S_{b_2};\, e_{c_1} \rangle$ and $\langle S_{b_1} \sqcup_S S_{b_2};\, e_{b_2} \rangle \hookrightarrow \langle S_{b_1} \sqcup_S S_{c_2};\, e_{c_2} \rangle$. But since $S_{c_1} \sqcup_S S_2 = \top_S$, we have that $S_{c_1} \sqcup_S S_{c_2} = \top_S$, since $S_2 \sqsubseteq_S S_{c_2}$.
And since $S_{c_1} \sqcup_S S_{c_2} = \top_S$, we have that:

$$\begin{aligned}(S_{c_1} \sqcup_S S_{b_2}) \sqcup_S (S_{b_1} \sqcup_S S_{c_2}) &= S_{c_1} \sqcup_S S_{c_2} \sqcup_S S_{b_1} \sqcup_S S_{b_2} \\ &= \top_S \sqcup_S S_{b_1} \sqcup_S S_{b_2} \\ &= \top_S.\end{aligned}$$

Therefore, E-PARAPPERR applies, and $\langle S_{b_1} \sqcup_S S_{b_2};\, e_{b_1}\ e_{b_2} \rangle \hookrightarrow$ **error**.

Therefore, in this case, $\sigma_c =$ **error**.

(b) $S_1 \sqcup_S S_{c_2} = \top_S$.
Then, since $\langle S_2;\, e_2' \rangle \hookrightarrow \langle S_{c_2};\, e_{c_2} \rangle$, by Lemma 4 we have that $\langle S_1 \sqcup_S S_2;\, e_2' \rangle \hookrightarrow$ **error**. Therefore, by E-APPERR-2, $\langle S_1 \sqcup_S S_2;\, e_1'\ e_2' \rangle \hookrightarrow$ **error**.
Next, we show that $\langle S_{b_1} \sqcup_S S_{b_2};\, e_{b_1}\ e_{b_2} \rangle$ must step to **error**, as well. At least one of the following three possibilities must hold:

i. $S_{c_1} \sqcup_S S_{b_2} = \top_S$.
Then $\langle S_{b_1} \sqcup_S S_{b_2};\, e_{b_1}\ e_{b_2} \rangle \hookrightarrow$ **error** by the same argument as 3(a)i.

ii. $S_{b_1} \sqcup_S S_{c_2} = \top_S$.
Then $\langle S_{b_1} \sqcup_S S_{b_2};\, e_{b_1}\ e_{b_2} \rangle \hookrightarrow$ **error** by the same argument as 3(a)ii.

iii. $S_{c_1} \sqcup_S S_{b_2} \neq \top_S$ and $S_{b_1} \sqcup_S S_{c_2} \neq \top_S$.
Then the argument of 3(a)iii applies, with the modification that, since $S_1 \sqcup_S S_{c_2} = \top_S$, we have that $S_{c_1} \sqcup_S S_{c_2} = \top_S$, since $S_1 \sqsubseteq_S S_{c_1}$.
So, $\langle S_{b_1} \sqcup_S S_{b_2};\, e_{b_1}\ e_{b_2} \rangle \hookrightarrow$ **error**.

Therefore, in this case, $\sigma_c =$ **error**.

(c) $S_{c_1} \sqcup_S S_2 \neq \top_S$, $S_1 \sqcup_S S_{c_2} \neq \top_S$, and $(S_{c_1} \sqcup_S S_2) \sqcup_S (S_1 \sqcup_S S_{c_2}) = \top_S$.
Then, since $\langle S_1;\, e_1' \rangle \hookrightarrow \langle S_{c_1};\, e_{c_1} \rangle$ and $\langle S_2;\, e_2' \rangle \hookrightarrow \langle S_{c_2};\, e_{c_2} \rangle$, we have by Lemma 3 that $\langle S_1 \sqcup_S S_2;\, e_1' \rangle \hookrightarrow \langle S_{c_1} \sqcup_S S_2;\, e_{c_1} \rangle$ and $\langle S_1 \sqcup_S S_2;\, e_2' \rangle \hookrightarrow \langle S_1 \sqcup_S S_{c_2};\, e_{c_2} \rangle$. But since $(S_{c_1} \sqcup_S S_2) \sqcup_S (S_1 \sqcup_S S_{c_2}) = \top_S$, we have by E-PARAPPERR that $\langle S_1 \sqcup_S S_2;\, e_1'\ e_2' \rangle \hookrightarrow$ **error**.
Next, we show that $\langle S_{b_1} \sqcup_S S_{b_2};\, e_{b_1}\ e_{b_2} \rangle$ must step to **error**, as well. At least one of the following three possibilities must hold:

i. $S_{c_1} \sqcup_S S_{b_2} = \top_S$.
Then $\langle S_{b_1} \sqcup_S S_{b_2};\, e_{b_1}\ e_{b_2} \rangle \hookrightarrow$ **error** by the same argument as 3(a)i.

ii. $S_{b_1} \sqcup_S S_{c_2} = \top_S$.
Then $\langle S_{b_1} \sqcup_S S_{b_2};\, e_{b_1}\ e_{b_2} \rangle \hookrightarrow$ **error** by the same argument as 3(a)ii.

iii. $S_{c_1} \sqcup_S S_{b_2} \neq \top_S$ and $S_{b_1} \sqcup_S S_{c_2} \neq \top_S$.
Then the argument of 3(a)iii applies, with the modification that, since $(S_{c_1} \sqcup_S S_2) \sqcup_S (S_1 \sqcup_S S_{c_2}) = \top_S$, we have that $S_{c_1} \sqcup_S S_{c_2} = \top_S$, since $S_1 \sqsubseteq_S S_{c_1}$ and $S_2 \sqsubseteq_S S_{c_2}$.
So, $\langle S_{b_1} \sqcup_S S_{b_2};\, e_{b_1}\ e_{b_2} \rangle \hookrightarrow$ **error**.

Therefore, in this case, $\sigma_c = \textbf{error}$.

(d) $S_{c_1} \sqcup_S S_2 \neq \top_S$, $S_1 \sqcup_S S_{c_2} \neq \top_S$, and $(S_{c_1} \sqcup_S S_2) \sqcup_S (S_1 \sqcup_S S_{c_2}) \neq \top_S$.
Then, since $\langle S_1;\ e_1' \rangle \longrightarrow \langle S_{c_1};\ e_{c_1} \rangle$ and $\langle S_2;\ e_2' \rangle \longrightarrow \langle S_{c_2};\ e_{c_2} \rangle$, we have by Lemma 3 that $\langle S_1 \sqcup_S S_2;\ e_1' \rangle \longrightarrow \langle S_{c_1} \sqcup_S S_2;\ e_{c_1} \rangle$ and $\langle S_1 \sqcup_S S_2;\ e_2' \rangle \longrightarrow \langle S_1 \sqcup_S S_{c_2};\ e_{c_2} \rangle$.
So, by E-PARAPP, we have that $\langle S_1 \sqcup_S S_2;\ e_1'\ e_2' \rangle \longrightarrow \langle (S_{c_1} \sqcup_S S_2) \sqcup_S (S_1 \sqcup_S S_{c_2});\ e_{c_1}\ e_{c_2} \rangle$.
Since $S_1 \sqsubseteq_S S_{c_1}$ and $S_2 \sqsubseteq_S S_{c_2}$, we can simplify $(S_{c_1} \sqcup_S S_2) \sqcup_S (S_1 \sqcup_S S_{c_2})$ to $S_{c_1} \sqcup_S S_{c_2}$, so we have that $S_{c_1} \sqcup_S S_{c_2} \neq \top_S$, and $\langle S_1 \sqcup_S S_2;\ e_1'\ e_2' \rangle \longrightarrow \langle S_{c_1} \sqcup_S S_{c_2};\ e_{c_1}\ e_{c_2} \rangle$.
Next, we show that $\langle S_{b_1} \sqcup_S S_{b_2};\ e_{b_1}\ e_{b_2} \rangle$ must step to $\langle S_{c_1} \sqcup_S S_{c_2};\ e_{c_1}\ e_{c_2} \rangle$, as well. At least one of the following possibilities must hold:

i. $S_{c_1} \sqcup_S S_{b_2} = \top_S$.
Can't happen, because if it were true, we would have $S_{c_1} \sqcup_S S_{c_2} = \top_S$ (since $S_{b_2} \sqsubseteq_S S_{c_2}$), which would contradict $S_{c_1} \sqcup_S S_{c_2} \neq \top_S$, above.

ii. $S_{b_1} \sqcup_S S_{c_2} = \top_S$.
Can't happen, because if it were true, we would have $S_{c_1} \sqcup_S S_{c_2} = \top_S$ (since $S_{b_1} \sqsubseteq_S S_{c_1}$), which would contradict $S_{c_1} \sqcup_S S_{c_2} \neq \top_S$, above.

iii. $S_{c_1} \sqcup_S S_{b_2} \neq \top_S$ and $S_{b_1} \sqcup_S S_{c_2} \neq \top_S$.
Then, since $\langle S_{b_1};\ e_{b_1} \rangle \longrightarrow \langle S_{c_1};\ e_{c_1} \rangle$ and $\langle S_{b_2};\ e_{b_2} \rangle \longrightarrow \langle S_{c_2};\ e_{c_2} \rangle$, we have by Lemma 3 that $\langle S_{b_1} \sqcup_S S_{b_2};\ e_{b_1} \rangle \longrightarrow \langle S_{c_1} \sqcup_S S_{b_2};\ e_{c_1} \rangle$ and $\langle S_{b_1} \sqcup_S S_{b_2};\ e_{b_2} \rangle \longrightarrow \langle S_{b_1} \sqcup_S S_{c_2};\ e_{c_2} \rangle$.
Then, since $S_{c_1} \sqcup_S S_{c_2} \neq \top_S$ and $S_{b_1} \sqsubseteq_S S_{c_1}$ and $S_{b_2} \sqsubseteq_S S_{c_2}$, we have that $(S_{c_1} \sqcup_S S_{b_2}) \sqcup_S (S_{b_1} \sqcup_S S_{c_2}) \neq \top_S$. Therefore, E-PARAPP applies, and we have that $\langle S_{b_1} \sqcup_S S_{b_2};\ e_{b_1}\ e_{b_2} \rangle \longrightarrow \langle (S_{c_1} \sqcup_S S_{b_2}) \sqcup_S (S_{b_1} \sqcup_S S_{c_2});\ e_{c_1}\ e_{c_2} \rangle$. Since $(S_{c_1} \sqcup_S S_{b_2}) \sqcup_S (S_{b_1} \sqcup_S S_{c_2})$ simplifies to $S_{c_1} \sqcup_S S_{c_2}$, we have that $\langle S_{b_1} \sqcup_S S_{b_2};\ e_{b_1}\ e_{b_2} \rangle \longrightarrow \langle S_{c_1} \sqcup_S S_{c_2};\ e_{c_1}\ e_{c_2} \rangle$.

Therefore, in this case, $\sigma_c = \langle S_{c_1} \sqcup_S S_{c_2};\ e_{c_1}\ e_{c_2} \rangle$.

– E-PARAPP/E-BETA:

In this case, we have the following facts:

* $\langle S;\ e_1\ e_2 \rangle = \langle S;\ \lambda x.\, e_{11}\ v \rangle$ for some e_{11} and some value v; and
* $\sigma_b = \langle S;\ e_{11}[x := v] \rangle$.

We're required to show that there exists σ_c such that

* $\langle S_1 \sqcup_S S_2;\ e_1'\ e_2' \rangle \longrightarrow \sigma_c$, and
* $\langle S;\ e_{11}[x := v] \rangle \longrightarrow \sigma_c$.

Choose $\sigma_c = \langle S;\ e_{11}[x := v] \rangle$. We have from E-REFL that $\langle S;\ e_{11}[x := v] \rangle \longrightarrow \langle S;\ e_{11}[x := v] \rangle$, so it remains to show that $\langle S_1 \sqcup_S S_2;\ e_1'\ e_2' \rangle \longrightarrow \langle S;\ e_{11}[x := v] \rangle$.

From the premises of E-PARAPP, we have that $\langle S;\ e_1 \rangle \longrightarrow \langle S_1;\ e_1' \rangle$ and $\langle S;\ e_2 \rangle \longrightarrow \langle S_2;\ e_2' \rangle$. But $e_1 = \lambda x.\, e_{11}$, a value, and $e_2 = v$, a value. So it must be the case that $e_1 = e_1'$, $e_2 = e_2'$, and $S = S_1 = S_2$. Therefore, $\langle S_1 \sqcup_S S_2;\ e_1'\ e_2' \rangle = \langle S;\ \lambda x.\, e_{11}\ v \rangle$, so we have only to show that $\langle S;\ e_1\ e_2 \rangle \longrightarrow \langle S;\ e_{11}[x := v] \rangle$, which is immediate by E-BETA.

– E-PARAPP/E-APPERR-1:

In this case, we have the following facts:

* $\sigma_b = \textbf{error}$;
* $\langle S;\ e_1 \rangle \longrightarrow \textbf{error}$ (from the premise of E-APPERR-1).

We're required to show that there exists σ_c such that

* $\langle S_1 \sqcup_S S_2;\ e_1'\ e_2' \rangle \longrightarrow \sigma_c$, and
* $\textbf{error} \longrightarrow \sigma_c$.

Choose $\sigma_c = \textbf{error}$. We have immediately that $\textbf{error} \longrightarrow \textbf{error}$ by E-REFLERR, so it remains to show that $\langle S_1 \sqcup_S S_2;\ e_1'\ e_2' \rangle \longrightarrow \textbf{error}$.

Since $\langle S;\ e_1\rangle \hookrightarrow$ **error** and $\langle S;\ e_1\rangle \hookrightarrow \langle S_1;\ e_1'\rangle$ (from the premises of E-PARAPP, above), we have by IH that there exists σ_{c_1} such that **error** $\hookrightarrow \sigma_{c_1}$ and $\langle S_1;\ e_1'\rangle \hookrightarrow \sigma_{c_1}$. Since **error** can only step to **error**, $\sigma_{c_1} =$ **error**.

Therefore, $\langle S_1;\ e_1'\rangle \hookrightarrow$ **error**, so we have that $\langle S_1;\ e_1'\ e_2'\rangle \hookrightarrow$ **error** by E-APPERR-1, and therefore $\langle S_1 \sqcup_S S_2;\ e_1'\ e_2'\rangle \hookrightarrow$ **error** by Lemma 5, as we were required to show.

– E-PARAPP/E-APPERR-2:

In this case, we have the following facts:

* $\sigma_b =$ **error**;
* $\langle S;\ e_2\rangle \hookrightarrow$ **error** (from the premise of E-APPERR-2).

We're required to show that there exists σ_c such that

* $\langle S_1 \sqcup_S S_2;\ e_1'\ e_2'\rangle \hookrightarrow \sigma_c$, and
* **error** $\hookrightarrow \sigma_c$.

Choose $\sigma_c =$ **error**. We have immediately that **error** $\hookrightarrow$ **error** by E-REFLERR, so it remains to show that $\langle S_1 \sqcup_S S_2;\ e_1'\ e_2'\rangle \hookrightarrow$ **error**.

Since $\langle S;\ e_2\rangle \hookrightarrow$ **error** and $\langle S;\ e_2\rangle \hookrightarrow \langle S_2;\ e_2'\rangle$ (from the premises of E-PARAPP, above), we have by IH that there exists σ_{c_2} such that **error** $\hookrightarrow \sigma_{c_2}$ and $\langle S_2;\ e_2'\rangle \hookrightarrow \sigma_{c_2}$. Since **error** can only step to **error**, $\sigma_{c_2} =$ **error**.

Therefore, $\langle S_2;\ e_2'\rangle \hookrightarrow$ **error**, so we have that $\langle S_2;\ e_1'\ e_2'\rangle \hookrightarrow$ **error** by E-APPERR-2, and therefore $\langle S_1 \sqcup_S S_2;\ e_1'\ e_2'\rangle \hookrightarrow$ **error** by Lemma 5, as we were required to show.

– E-PARAPP/E-PARAPPERR:

In this case, we have the following facts:

* $\sigma_b =$ **error**;
* $\langle S;\ e_1\rangle \hookrightarrow \langle S_{b_1};\ e_{b_1}\rangle$ for some S_{b_1} and e_{b_1} (from the first premise of E-PARAPPERR);
* $\langle S;\ e_2\rangle \hookrightarrow \langle S_{b_2};\ e_{b_2}\rangle$ for some S_{b_2} and e_{b_2} (from the second premise of E-PARAPPERR);
* $S_{b_1} \sqcup_S S_{b_2} = \top_S$ (from the third premise of E-PARAPPERR).

We're required to show that there exists σ_c such that

* $\langle S_1 \sqcup_S S_2;\ e_1'\ e_2'\rangle \hookrightarrow \sigma_c$, and
* **error** $\hookrightarrow \sigma_c$.

Choose $\sigma_c =$ **error**. We have immediately that **error** $\hookrightarrow$ **error** by E-REFLERR, so it remains to show that $\langle S_1 \sqcup_S S_2;\ e_1'\ e_2'\rangle \hookrightarrow$ **error**.

Since $\langle S;\ e_1\rangle \hookrightarrow \langle S_1;\ e_1'\rangle$ (from the first premise of E-PARAPP) and $\langle S;\ e_1\rangle \hookrightarrow \langle S_{b_1};\ e_{b_1}\rangle$, and since $\langle S;\ e_2\rangle \hookrightarrow \langle S_2;\ e_2'\rangle$ (from the second premise of E-PARAPP) and $\langle S;\ e_2\rangle \hookrightarrow \langle S_{b_2};\ e_{b_2}\rangle$, we have by IH that there exist σ_{c_1} and σ_{c_2} such that $\langle S_1;\ e_1'\rangle \hookrightarrow \sigma_{c_1}$ and $\langle S_{b_1};\ e_{b_1}\rangle \hookrightarrow \sigma_{c_1}$, and that $\langle S_2;\ e_2'\rangle \hookrightarrow \sigma_{c_2}$ and $\langle S_{b_2};\ e_{b_2}\rangle \hookrightarrow \sigma_{c_2}$.

We consider the following possibilities, at least one of which must hold:

* $\sigma_{c_1} =$ **error**.
  In this case, since $\langle S_1;\ e_1'\rangle \hookrightarrow$ **error**, we have by E-APPERR-1 that $\langle S_1;\ e_1'\ e_2'\rangle \hookrightarrow$ **error**. Therefore, by Lemma 5, we have that $\langle S_1 \sqcup_S S_2;\ e_1'\ e_2'\rangle \hookrightarrow$ **error**, as we were required to show.
* $\sigma_{c_2} =$ **error**.
  In this case, since $\langle S_2;\ e_2'\rangle \hookrightarrow$ **error**, we have by E-APPERR-2 that $\langle S_2;\ e_1'\ e_2'\rangle \hookrightarrow$ **error**. Therefore, by Lemma 5, we have that $\langle S_1 \sqcup_S S_2;\ e_1'\ e_2'\rangle \hookrightarrow$ **error**, as we were required to show.
* $\sigma_{c_1} = \langle S_{c_1};\ e_{c_1}\rangle \neq$ **error** and $\sigma_{c_2} = \langle S_{c_2};\ e_{c_2}\rangle \neq$ **error**.
  In this case, at least one of the following three possibilities must hold:
  1. $S_{c_1} \sqcup_S S_2 = \top_S$.
     Since $\langle S_1;\ e_1'\rangle \hookrightarrow \langle S_{c_1};\ e_{c_1}\rangle$ and $S_{c_1} \sqcup_S S_2 = \top_S$, we have by Lemma 4 that $\langle S_1 \sqcup_S S_2;\ e_1'\rangle \hookrightarrow$ **error**. Therefore, by E-APPERR-1, $\langle S_1 \sqcup_S S_2;\ e_1'\ e_2'\rangle \hookrightarrow$ **error**, as we were required to show.

2. $S_1 \sqcup_S S_{c_2} = \top_S$.
   Since $\langle S_2;\, e'_2\rangle \hookrightarrow \langle S_{c_2};\, e_{c_2}\rangle$ and $S_1 \sqcup_S S_{c_2} = \top_S$, we have by Lemma 4 that $\langle S_1 \sqcup_S S_2;\, e'_2\rangle \hookrightarrow$ **error**. Therefore, by E-APPERR-2, $\langle S_1 \sqcup_S S_2;\, e'_1\ e'_2\rangle \hookrightarrow$ **error**, as we were required to show.
3. $S_{c_1} \sqcup_S S_2 \neq \top_S$ and $S_1 \sqcup_S S_{c_2} \neq \top_S$.
   In this case, since $\langle S_1;\, e'_1\rangle \hookrightarrow \langle S_{c_1};\, e_{c_1}\rangle$ and $S_{c_1} \sqcup_S S_2 \neq \top_S$, we have by Lemma 3 that $\langle S_1 \sqcup_S S_2;\, e'_1\rangle \hookrightarrow \langle S_{c_1} \sqcup_S S_2;\, e_{c_1}\rangle$.
   Likewise, since $\langle S_2;\, e'_2\rangle \hookrightarrow \langle S_{c_2};\, e_{c_2}\rangle$ and $S_1 \sqcup_S S_{c_2} \neq \top_S$, we have by Lemma 3 that $\langle S_1 \sqcup_S S_2;\, e'_2\rangle \hookrightarrow \langle S_1 \sqcup_S S_{c_2};\, e_{c_2}\rangle$.
   But since $S_{b_1} \sqsubseteq_S S_{c_1}$ and $S_{b_2} \sqsubseteq_S S_{c_2}$ and $S_{b_1} \sqcup_S S_{b_2} = \top_S$, it must be the case that $S_{c_1} \sqcup_S S_{c_2} = \top_S$. Therefore we have that
   $(S_{c_1} \sqcup_S S_2) \sqcup_S (S_1 \sqcup_S S_{c_2}) = S_{c_1} \sqcup_S S_{c_2} = \top_S$. So, by E-PARAPPERR, $\langle S_1 \sqcup_S S_2;\, e'_1\ e'_2\rangle \hookrightarrow$ **error**, as we were required to show.

A.6.3 E-PUT-1

- E-PUT-1: $\sigma = \langle S;\, \mathsf{put}\ e_1\ e_2\rangle$, and $\sigma_a = \langle S_1;\, \mathsf{put}\ e'_1\ e_2\rangle$.

  Given:

  - $\langle S;\, \mathsf{put}\ e_1\ e_2\rangle \hookrightarrow \langle S_1;\, \mathsf{put}\ e'_1\ e_2\rangle$, and
  - $\langle S;\, \mathsf{put}\ e_1\ e_2\rangle \hookrightarrow \sigma_b$.

  To show: There exists σ_c such that

  - $\langle S_1;\, \mathsf{put}\ e'_1\ e_2\rangle \hookrightarrow \sigma_c$, and
  - $\sigma_b \hookrightarrow \sigma_c$.

  From the premise of E-PUT-1, we have that $\langle S;\, e_1\rangle \hookrightarrow \langle S_1;\, e'_1\rangle$.

  We proceed by subcases, on the last rule in the derivation of $\langle S;\, \mathsf{put}\ e_1\ e_2\rangle \hookrightarrow \sigma_b$. By the operational semantics, there are seven possibilities: E-PUT-1/E-REFL, E-PUT-1/E-PUT-1, E-PUT-1/E-PUT-2, E-PUT-1/E-PUTVAL, E-PUT-1/E-PUTERR-1, E-PUT-1/E-PUTERR-2, and E-PUT-1/E-PUTVALERR.

  - E-PUT-1/E-REFL:
    Analogous to the E-REFL/E-PUT-1 case, with σ_a and σ_b reversed.
  - E-PUT-1/E-PUT-1:
    In this case, we have the following facts:
    * $\sigma_b = \langle S_{b_1};\, \mathsf{put}\ e_{b_1}\ e_2\rangle$, and
    * $\langle S;\, e_1\rangle \hookrightarrow \langle S_{b_1};\, e_{b_1}\rangle$.

    Since $\langle S;\, e_1\rangle \hookrightarrow \langle S_1;\, e'_1\rangle$ and $\langle S;\, e_1\rangle \hookrightarrow \langle S_{b_1};\, e_{b_1}\rangle$, we have by IH that there exists σ_{c_1} such that $\langle S_1;\, e'_1\rangle \hookrightarrow \sigma_{c_1}$ and $\langle S_{b_1};\, e_{b_1}\rangle \hookrightarrow \sigma_{c_1}$. Either σ_{c_1} is **error**, or it is some non-**error** configuration $\langle S_{c_1};\, e_{c_1}\rangle$.
    We're required to show that there exists σ_c such that
    * $\langle S_1;\, \mathsf{put}\ e'_1\ e_2\rangle \hookrightarrow \sigma_c$, and
    * $\langle S_{b_1};\, \mathsf{put}\ e_{b_1}\ e_2\rangle \hookrightarrow \sigma_c$.

    We consider the following possibilities, one of which must hold.

    1. $\sigma_{c_1} =$ **error**.
       Then, since $\langle S_1;\, e'_1\rangle \hookrightarrow$ **error**, we have by E-PUTERR-1 that $\langle S_1;\, \mathsf{put}\ e'_1\ e_2\rangle \hookrightarrow$ **error**. Likewise, since $\langle S_{b_1};\, e_{b_1}\rangle \hookrightarrow$ **error**, we have by E-PUTERR-1 that $\langle S_{b_1};\, \mathsf{put}\ e_{b_1}\ e_2\rangle \hookrightarrow$ **error**. Therefore $\sigma_c =$ **error**.

2. $\sigma_{c_1} = \langle S_{c_1};\, e_{c_1} \rangle$.
   Then, since $\langle S_1;\, e_1' \rangle \longhookrightarrow \langle S_{c_1};\, e_{c_1} \rangle$, we have by E-PUT-1 that $\langle S_1;\, \mathsf{put}\, e_1'\ e_2 \rangle \longhookrightarrow \langle S_{c_1};\, \mathsf{put}\, e_{c_1}\ e_2 \rangle$.
   Likewise, since $\langle S_{b_1};\, e_{b_1} \rangle \longhookrightarrow \langle S_{c_1};\, e_{c_1} \rangle$, we have by E-PUT-1 that $\langle S_{b_1};\, \mathsf{put}\, e_{b_1}\ e_2 \rangle \longhookrightarrow \langle S_{c_1};\, \mathsf{put}\, e_{c_1}\ e_2 \rangle$.
   Therefore $\sigma_c = \langle S_{c_1};\, \mathsf{put}\, e_{c_1}\ e_2 \rangle$.

– E-PUT-1/E-PUT-2:

(NB: In this case we assume that configurations are renamed as necessary to meet non-conflicting requirements.)

In this case, we have the following facts:

* $\sigma_b = \langle S_{b_2};\, \mathsf{put}\, e_1\ e_{b_2} \rangle$, and
* $\langle S;\, e_2 \rangle \longhookrightarrow \langle S_{b_2};\, e_{b_2} \rangle$.

We're required to show that there exists σ_c such that

* $\langle S_1;\, \mathsf{put}\, e_1'\ e_2 \rangle \longhookrightarrow \sigma_c$, and
* $\langle S_{b_2};\, \mathsf{put}\, e_1\ e_{b_2} \rangle \longhookrightarrow \sigma_c$.

We consider the following two possibilities, one of which must hold:

1. $S_1 \sqcup_S S_{b_2} = \top_S$.
   Since $\langle S;\, e_2 \rangle \longhookrightarrow \langle S_{b_2};\, e_{b_2} \rangle$ (from above), and since $S_1 \sqcup_S S_{b_2} = S_{b_2} \sqcup_S S_1 = \top_S$, we have by Lemma 4 that $\langle S \sqcup_S S_1;\, e_2 \rangle \longhookrightarrow \mathbf{error}$.
   Since $S \sqsubseteq_S S_1$, we have that $S \sqcup_S S_1 = S_1$, so $\langle S_1;\, e_2 \rangle \longhookrightarrow \mathbf{error}$.
   Therefore, by E-PUTERR-2, $\langle S_1;\, \mathsf{put}\, e_1'\ e_2 \rangle \longhookrightarrow \mathbf{error}$.
   Similarly, since $\langle S;\, e_1 \rangle \longhookrightarrow \langle S_1;\, e_1' \rangle$ (from the premise of E-PUT-1), and since $S_1 \sqcup_S S_{b_2} = \top_S$, we have by Lemma 4 that $\langle S \sqcup_S S_{b_2};\, e_1 \rangle \longhookrightarrow \mathbf{error}$.
   Since $S \sqsubseteq_S S_{b_2}$, we have that $S \sqcup_S S_{b_2} = S_{b_2}$, so $\langle S_{b_2};\, e_1 \rangle \longhookrightarrow \mathbf{error}$.
   Therefore, by E-PUTERR-1, $\langle S_{b_2};\, \mathsf{put}\, e_1\ e_{b_2} \rangle \longhookrightarrow \mathbf{error}$.
   Therefore $\sigma_c = \mathbf{error}$.
2. $S_1 \sqcup_S S_{b_2} \neq \top_S$.
   Since $\langle S;\, e_2 \rangle \longhookrightarrow \langle S_{b_2};\, e_{b_2} \rangle$ (from above), and since $S_1 \sqcup_S S_{b_2} = S_{b_2} \sqcup_S S_1 \neq \top_S$, we have by Lemma 3 that $\langle S \sqcup_S S_1;\, e_2 \rangle \longhookrightarrow \langle S_{b_2} \sqcup_S S_1;\, e_{b_2} \rangle$.
   Since $S \sqsubseteq_S S_1$, we have that $S \sqcup_S S_1 = S_1$, so $\langle S_1;\, e_2 \rangle \longhookrightarrow \langle S_{b_2} \sqcup_S S_1;\, e_{b_2} \rangle$.
   Therefore, by E-PUT-2, $\langle S_1;\, \mathsf{put}\, e_1'\ e_2 \rangle \longhookrightarrow \langle S_{b_2} \sqcup_S S_1;\, \mathsf{put}\, e_1'\ e_{b_2} \rangle$.
   Similarly, since $\langle S;\, e_1 \rangle \longhookrightarrow \langle S_1;\, e_1' \rangle$ (from the premise of E-PUT-1), and since $S_1 \sqcup_S S_{b_2} \neq \top_S$, we have by Lemma 3 that $\langle S \sqcup_S S_{b_2};\, e_1 \rangle \longhookrightarrow \langle S_1 \sqcup_S S_{b_2};\, e_1' \rangle$.
   Since $S \sqsubseteq_S S_{b_2}$, we have that $S \sqcup_S S_{b_2} = S_{b_2}$, so $\langle S_{b_2};\, e_1 \rangle \longhookrightarrow \langle S_1 \sqcup_S S_{b_2};\, e_1' \rangle$.
   Therefore, by E-PUT-1, $\langle S_{b_2};\, \mathsf{put}\, e_1\ e_{b_2} \rangle \longhookrightarrow \langle S_1 \sqcup_S S_{b_2};\, \mathsf{put}\, e_1'\ e_{b_2} \rangle$.
   Therefore $\sigma_c = \langle S_1 \sqcup_S S_{b_2};\, \mathsf{put}\, e_1'\ e_{b_2} \rangle$.

– E-PUT-1/E-PUTVAL:

In this case, we have the following facts:

* $\langle S;\, \mathsf{put}\, e_1\ e_2 \rangle = \langle S;\, \mathsf{put}\, l\ \{d_1\} \rangle$,
* $\sigma_b = \langle S[l \mapsto d_1 \sqcup d_2];\, \{\} \rangle$, and
* $S(l) = d_2 \wedge d_1 \in D \wedge d_1 \sqcup d_2 \neq \top$ (from the premises of E-PUTVAL).

We're required to show that there exists σ_c such that

* $\langle S_1;\, \mathsf{put}\, e_1'\ e_2 \rangle \longhookrightarrow \sigma_c$, and
* $\langle S[l \mapsto d_1 \sqcup d_2];\, \{\} \rangle \longhookrightarrow \sigma_c$.

Choose $\sigma_c = \langle S[l \mapsto d_1 \sqcup d_2];\, \{\} \rangle$. We have from E-REFL that $\langle S[l \mapsto d_1 \sqcup d_2];\, \{\} \rangle \longhookrightarrow \langle S[l \mapsto d_1 \sqcup d_2];\, \{\} \rangle$, so it remains to show that $\langle S_1;\, \mathsf{put}\, e_1'\ e_2 \rangle \longhookrightarrow \langle S[l \mapsto d_1 \sqcup d_2];\, \{\} \rangle$.

From the premise of E-PUT-1, we have that $\langle S;\, e_1 \rangle \longhookrightarrow \langle S_1;\, e_1' \rangle$. But $e_1 = l$, a value, so it must be the case that $e_1 = e_1'$ and $S = S_1$. Therefore, $\langle S_1;\, \mathsf{put}\, e_1'\ e_2 \rangle = \langle S;\, \mathsf{put}\, e_1\ e_2 \rangle$. Further, since $e_1 = l$ and $e_2 = \{d_1\}$, $\langle S_1;\, \mathsf{put}\, e_1'\ e_2 \rangle = \langle S;\, \mathsf{put}\, l\ \{d_1\} \rangle$. So we have only to show that $\langle S;\, \mathsf{put}\, l\ \{d_1\} \rangle \longhookrightarrow \langle S[l \mapsto d_1 \sqcup d_2];\, \{\} \rangle$, which is immediate by E-PUTVAL, since all of the premises hold.

– E-PUT-1/E-PUTERR-1:

In this case, we have the following facts:

* $\sigma_b = \textbf{error}$, and
* $\langle S;\, e_1\rangle \hookrightarrow \textbf{error}$ (from the premise of E-PUTERR-1).

We're required to show that there exists σ_c such that

* $\langle S_1;\, \mathsf{put}\ e_1'\ e_2\rangle \hookrightarrow \sigma_c$, and
* $\textbf{error} \hookrightarrow \sigma_c$.

Choose $\sigma_c = \textbf{error}$. We have immediately that $\textbf{error} \hookrightarrow \textbf{error}$ by E-REFLERR, so it remains to show that $\langle S_1;\, \mathsf{put}\ e_1'\ e_2\rangle \hookrightarrow \textbf{error}$.

Since $\langle S;\, e_1\rangle \hookrightarrow \textbf{error}$ and $\langle S;\, e_1\rangle \hookrightarrow \langle S_1;\, e_1'\rangle$ (from the premise of E-PUT-1, above), we have by IH that there exists σ_{c_1} such that $\textbf{error} \hookrightarrow \sigma_{c_1}$ and $\langle S_1;\, e_1'\rangle \hookrightarrow \sigma_{c_1}$. Since **error** can only step to **error**, $\sigma_{c_1} = \textbf{error}$.

Therefore, $\langle S_1;\, e_1'\rangle \hookrightarrow \textbf{error}$, so we have that $\langle S_1;\, \mathsf{put}\ e_1'\ e_2\rangle \hookrightarrow \textbf{error}$ by E-PUTERR-1, as we were required to show.

– E-PUT-1/E-PUTERR-2:

In this case, we have the following facts:

* $\sigma_b = \textbf{error}$, and
* $\langle S;\, e_2\rangle \hookrightarrow \textbf{error}$ (from the premise of E-PUTERR-2).

We're required to show that there exists σ_c such that

* $\langle S_1;\, \mathsf{put}\ e_1'\ e_2\rangle \hookrightarrow \sigma_c$, and
* $\textbf{error} \hookrightarrow \sigma_c$.

Choose $\sigma_c = \textbf{error}$. We have immediately that $\textbf{error} \hookrightarrow \textbf{error}$ by E-REFLERR, so it remains to show that $\langle S_1;\, \mathsf{put}\ e_1'\ e_2\rangle \hookrightarrow \textbf{error}$.

Since $\langle S;\, e_2\rangle \hookrightarrow \textbf{error}$, we have by E-PUTERR-2 that $\langle S;\, \mathsf{put}\ e_1'\ e_2\rangle \hookrightarrow \textbf{error}$. So, since $S \sqsubseteq_S S_1$, we have by Lemma 5 that $\langle S_1;\, \mathsf{put}\ e_1'\ e_2\rangle \hookrightarrow \textbf{error}$, as we were required to show.

– E-PUT-1/E-PUTVALERR:

In this case, we have the following facts:

* $\langle S;\, \mathsf{put}\ e_1\ e_2\rangle = \langle S;\, \mathsf{put}\ l\ \{d_1\}\rangle$,
* $\sigma_b = \textbf{error}$, and
* $S(l) = d_2 \wedge d_1 \in D \wedge d_1 \sqcup d_2 = \top$ (from the premises of E-PUTVALERR).

We're required to show that there exists σ_c such that

* $\langle S_1;\, \mathsf{put}\ e_1'\ e_2\rangle \hookrightarrow \sigma_c$, and
* $\textbf{error} \hookrightarrow \sigma_c$.

Choose $\sigma_c = \textbf{error}$. We have from E-REFL-ERR that $\textbf{error} \hookrightarrow \textbf{error}$, so it remains to show that $\langle S_1;\, \mathsf{put}\ e_1'\ e_2\rangle \hookrightarrow \textbf{error}$.

From the premise of E-PUT-1, we have that $\langle S;\, e_1\rangle \hookrightarrow \langle S_1;\, e_1'\rangle$. But $e_1 = l$, a value, so it must be the case that $e_1 = e_1'$ and $S = S_1$. Therefore, $\langle S_1;\, \mathsf{put}\ e_1'\ e_2\rangle = \langle S;\, \mathsf{put}\ e_1\ e_2\rangle$. Further, since $e_1 = l$ and $e_2 = \{d_1\}$, $\langle S_1;\, \mathsf{put}\ e_1'\ e_2\rangle = \langle S;\, \mathsf{put}\ l\ \{d_1\}\rangle$. So we have only to show that $\langle S;\, \mathsf{put}\ l\ \{d_1\}\rangle \hookrightarrow \textbf{error}$, which is immediate by E-PUTVALERR, since all of the premises hold.

A.6.4 E-PUT-2

- E-PUT-2: $\sigma = \langle S;\, \mathsf{put}\ e_1\ e_2\rangle$, and $\sigma_a = \langle S_1;\, \mathsf{put}\ e_1\ e_2'\rangle$.

  Given:

– $\langle S;\ \mathsf{put}\ e_1\ e_2\rangle \hookrightarrow \langle S_2;\ \mathsf{put}\ e_1\ e_2'\rangle$, and
– $\langle S;\ \mathsf{put}\ e_1\ e_2\rangle \hookrightarrow \sigma_b$.

To show: There exists σ_c such that

– $\langle S_2;\ \mathsf{put}\ e_1\ e_2'\rangle \hookrightarrow \sigma_c$, and
– $\sigma_b \hookrightarrow \sigma_c$.

From the premise of E-PUT-2, we have that $\langle S;\ e_2\rangle \hookrightarrow \langle S_2;\ e_2'\rangle$.

We proceed by subcases, on the last rule in the derivation of $\langle S;\ \mathsf{put}\ e_1\ e_2\rangle \hookrightarrow \sigma_b$. By the operational semantics, there are seven possibilities: E-PUT-2/E-REFL, E-PUT-2/E-PUT-1, E-PUT-2/E-PUT-2, E-PUT-2/E-PUTVAL, E-PUT-2/E-PUTERR-1, E-PUT-2/E-PUTERR-2, and E-PUT-2/E-PUTVALERR.

– E-PUT-2/E-REFL:
  Analogous to the E-REFL/E-PUT-2 case, with σ_a and σ_b reversed.
– E-PUT-2/E-PUT-1:
  Analogous to the E-PUT-1/E-PUT-2 case, with σ_a and σ_b reversed.
– E-PUT-2/E-PUT-2:
  In this case, we have the following facts:
  * $\sigma_b = \langle S_{b_2};\ \mathsf{put}\ e_1\ e_{b_2}\rangle$, and
  * $\langle S;\ e_2\rangle \hookrightarrow \langle S_{b_2};\ e_{b_2}\rangle$.

  Since $\langle S;\ e_2\rangle \hookrightarrow \langle S_2;\ e_2'\rangle$ and $\langle S;\ e_2\rangle \hookrightarrow \langle S_{b_2};\ e_{b_2}\rangle$, we have by IH that there exists σ_{c_2} such that $\langle S_2;\ e_2'\rangle \hookrightarrow \sigma_{c_2}$ and $\langle S_{b_2};\ e_{b_2}\rangle \hookrightarrow \sigma_{c_2}$. Either σ_{c_2} is **error**, or it is some non-**error** configuration $\langle S_{c_2};\ e_{c_2}\rangle$.
  We're required to show that there exists σ_c such that
  * $\langle S_2;\ \mathsf{put}\ e_1\ e_2'\rangle \hookrightarrow \sigma_c$, and
  * $\langle S_{b_2};\ \mathsf{put}\ e_1\ e_{b_2}\rangle \hookrightarrow \sigma_c$.

  We consider the following possibilities, one of which must hold.

  1. $\sigma_{c_2} = \textbf{error}$.
     Then, since $\langle S_2;\ e_2'\rangle \hookrightarrow \textbf{error}$, we have by E-PUTERR-2 that $\langle S_2;\ \mathsf{put}\ e_1\ e_2'\rangle \hookrightarrow \textbf{error}$. Likewise, since $\langle S_{b_2};\ e_{b_2}\rangle \hookrightarrow \textbf{error}$, we have by E-PUTERR-2 that $\langle S_{b_2};\ \mathsf{put}\ e_1\ e_{b_2}\rangle \hookrightarrow \textbf{error}$. Therefore $\sigma_c = \textbf{error}$.
  2. $\sigma_{c_2} = \langle S_{c_2};\ e_{c_2}\rangle$.
     Then, since $\langle S_2;\ e_2'\rangle \hookrightarrow \langle S_{c_2};\ e_{c_2}\rangle$, we have by E-PUT-2 that $\langle S_2;\ \mathsf{put}\ e_1\ e_2'\rangle \hookrightarrow \langle S_{c_2};\ \mathsf{put}\ e_1\ e_{c_2}\rangle$.
     Likewise, since $\langle S_{b_2};\ e_{b_2}\rangle \hookrightarrow \langle S_{c_2};\ e_{c_2}\rangle$, we have by E-PUT-2 that $\langle S_{b_2};\ \mathsf{put}\ e_1\ e_{b_2}\rangle \hookrightarrow \langle S_{c_2};\ \mathsf{put}\ e_1\ e_{c_2}\rangle$.
     Therefore $\sigma_c = \langle S_{c_2};\ \mathsf{put}\ e_1\ e_{c_2}\rangle$.
– E-PUT-2/E-PUTVAL:
  In this case, we have the following facts:
  * $\langle S;\ \mathsf{put}\ e_1\ e_2\rangle = \langle S;\ \mathsf{put}\ l\ \{d_1\}\rangle$,
  * $\sigma_b = \langle S[l \mapsto d_1 \sqcup d_2];\ \{\}\rangle$, and
  * $S(l) = d_2 \wedge d_1 \in D \wedge d_1 \sqcup d_2 \neq \top$ (from the premises of E-PUTVAL).

  We're required to show that there exists σ_c such that
  * $\langle S_2;\ \mathsf{put}\ e_1\ e_2'\rangle \hookrightarrow \sigma_c$, and
  * $\langle S[l \mapsto d_1 \sqcup d_2];\ \{\}\rangle \hookrightarrow \sigma_c$.

Choose $\sigma_c = \langle S[l \mapsto d_1 \sqcup d_2];\ \{\}\rangle$. We have from E-REFL that $\langle S[l \mapsto d_1 \sqcup d_2];\ \{\}\rangle \hookrightarrow \langle S[l \mapsto d_1 \sqcup d_2];\ \{\}\rangle$, so it remains to show that $\langle S_2;\ \mathsf{put}\ e_1\ e_2'\rangle \hookrightarrow \langle S[l \mapsto d_1 \sqcup d_2];\ \{\}\rangle$.

From the premise of E-PUT-2, we have that $\langle S;\ e_2\rangle \hookrightarrow \langle S_2;\ e_2'\rangle$. But $e_2 = \{d_1\}$, a value, so it must be the case that $e_2 = e_2'$ and $S = S_2$. Therefore, $\langle S_2;\ \mathsf{put}\ e_1\ e_2'\rangle = \langle S;\ \mathsf{put}\ e_1\ e_2\rangle$. Further, since $e_1 = l$ and $e_2 = \{d_1\}$, $\langle S_2;\ \mathsf{put}\ e_1\ e_2'\rangle = \langle S;\ \mathsf{put}\ l\ \{d_1\}\rangle$. So we have only to show that $\langle S;\ \mathsf{put}\ l\ \{d_1\}\rangle \hookrightarrow \langle S[l \mapsto d_1 \sqcup d_2];\ \{\}\rangle$, which is immediate by E-PUTVAL, since all of the premises hold.

– E-PUT-2/E-PUTERR-1:

In this case, we have the following facts:

* $\sigma_b = \textbf{error}$, and
* $\langle S;\ e_1\rangle \hookrightarrow \textbf{error}$ (from the premise of E-PUTERR-1).

We're required to show that there exists σ_c such that

* $\langle S_2;\ \mathsf{put}\ e_1\ e_2'\rangle \hookrightarrow \sigma_c$, and
* $\textbf{error} \hookrightarrow \sigma_c$.

Choose $\sigma_c = \textbf{error}$. We have immediately that $\textbf{error} \hookrightarrow \textbf{error}$ by E-REFLERR, so it remains to show that $\langle S_2;\ \mathsf{put}\ e_1\ e_2'\rangle \hookrightarrow \textbf{error}$.

Since $\langle S;\ e_1\rangle \hookrightarrow \textbf{error}$, we have by E-PUTERR-1 that $\langle S;\ \mathsf{put}\ e_1\ e_2'\rangle \hookrightarrow \textbf{error}$. So, since $S \sqsubseteq_S S_2$, we have by Lemma 5 that $\langle S_2;\ \mathsf{put}\ e_1\ e_2'\rangle \hookrightarrow \textbf{error}$, as we were required to show.

– E-PUT-2/E-PUTERR-2:

In this case, we have the following facts:

* $\sigma_b = \textbf{error}$, and
* $\langle S;\ e_2\rangle \hookrightarrow \textbf{error}$ (from the premise of E-PUTERR-2).

We're required to show that there exists σ_c such that

* $\langle S_2;\ \mathsf{put}\ e_1\ e_2'\rangle \hookrightarrow \sigma_c$, and
* $\textbf{error} \hookrightarrow \sigma_c$.

Choose $\sigma_c = \textbf{error}$. We have immediately that $\textbf{error} \hookrightarrow \textbf{error}$ by E-REFLERR, so it remains to show that $\langle S_2;\ \mathsf{put}\ e_1\ e_2'\rangle \hookrightarrow \textbf{error}$.

Since $\langle S;\ e_2\rangle \hookrightarrow \textbf{error}$ and $\langle S;\ e_2\rangle \hookrightarrow \langle S_2;\ e_2'\rangle$ (from the premise of E-PUT-2, above), we have by IH that there exists σ_{c_2} such that $\textbf{error} \hookrightarrow \sigma_{c_2}$ and $\langle S_2;\ e_2'\rangle \hookrightarrow \sigma_{c_2}$. Since **error** can only step to **error**, $\sigma_{c_2} = \textbf{error}$.

Therefore, $\langle S_2;\ e_2'\rangle \hookrightarrow \textbf{error}$, so we have that $\langle S_2;\ \mathsf{put}\ e_1\ e_2'\rangle \hookrightarrow \textbf{error}$ by E-PUTERR-2, as we were required to show.

– E-PUT-2/E-PUTVALERR:

In this case, we have the following facts:

* $\langle S;\ \mathsf{put}\ e_1\ e_2\rangle = \langle S;\ \mathsf{put}\ l\ \{d_1\}\rangle$,
* $\sigma_b = \textbf{error}$, and
* $S(l) = d_2 \wedge d_1 \in D \wedge d_1 \sqcup d_2 = \top$ (from the premises of E-PUTVALERR).

We're required to show that there exists σ_c such that

* $\langle S_2;\ \mathsf{put}\ e_1\ e_2'\rangle \hookrightarrow \sigma_c$, and
* $\textbf{error} \hookrightarrow \sigma_c$.

Choose $\sigma_c = \textbf{error}$. We have from E-REFL-ERR that $\textbf{error} \hookrightarrow \textbf{error}$, so it remains to show that $\langle S_2;\ \mathsf{put}\ e_1\ e_2'\rangle \hookrightarrow \textbf{error}$.

From the premise of E-PUT-2, we have that $\langle S;\ e_2\rangle \hookrightarrow \langle S_2;\ e_2'\rangle$. But $e_2 = \{d_1\}$, a value, so it must be the case that $e_2 = e_2'$ and $S = S_2$. Therefore, $\langle S_2;\ \mathsf{put}\ e_1\ e_2'\rangle = \langle S;\ \mathsf{put}\ e_1\ e_2\rangle$. Further, since $e_1 = l$ and $e_2 = \{d_1\}$, $\langle S_2;\ \mathsf{put}\ e_1\ e_2'\rangle = \langle S;\ \mathsf{put}\ l\ \{d_1\}\rangle$. So we have only to show that $\langle S;\ \mathsf{put}\ l\ \{d_1\}\rangle \hookrightarrow \textbf{error}$, which is immediate by E-PUTVALERR, since all of the premises hold.

A.6.5 E-GET-1

- E-GET-1: $\sigma = \langle S;\ \mathsf{get}\ e_1\ e_2\rangle$, and $\sigma_a = \langle S_1;\ \mathsf{get}\ e_1'\ e_2\rangle$.

  Given:

  - $\langle S;\ \mathsf{get}\ e_1\ e_2\rangle \hookrightarrow \langle S_1;\ \mathsf{get}\ e_1'\ e_2\rangle$, and
  - $\langle S;\ \mathsf{get}\ e_1\ e_2\rangle \hookrightarrow \sigma_b$.

  To show: There exists σ_c such that

  - $\langle S_1;\ \mathsf{get}\ e_1'\ e_2\rangle \hookrightarrow \sigma_c$, and
  - $\sigma_b \hookrightarrow \sigma_c$.

  From the premise of E-GET-1, we have that $\langle S;\ e_1\rangle \hookrightarrow \langle S_1;\ e_1'\rangle$.

  We proceed by subcases, on the last rule in the derivation of $\langle S;\ \mathsf{get}\ e_1\ e_2\rangle \hookrightarrow \sigma_b$. By the operational semantics, there are six possibilities: E-GET-1/E-REFL, E-GET-1/E-GET-1, E-GET-1/E-GET-2, E-GET-1/E-GETVAL, E-GET-1/E-GETERR-1, and E-GET-1/E-GETERR-2.

  - E-GET-1/E-REFL:
    Analogous to the E-REFL/E-GET-1 case, with σ_a and σ_b reversed.
  - E-GET-1/E-GET-1:
    Analogous to E-PUT-1/E-PUT-1.
  - E-GET-1/E-GET-2:
    Analogous to E-PUT-1/E-PUT-2.
  - E-GET-1/E-GETVAL:
    In this case, we have the following facts:
    * $\langle S;\ \mathsf{get}\ e_1\ e_2\rangle = \langle S;\ \mathsf{get}\ l\ Q\rangle$,
    * $\sigma_b = \langle S;\ \{d_1\}\rangle$, and
    * $S(l) = d_2 \wedge \mathit{incomp}(Q) \wedge Q \subseteq D \wedge d_1 \in Q \wedge d_1 \sqsubseteq d_2$ (from the premises of E-GETVAL).

    We're required to show that there exists σ_c such that
    * $\langle S_1;\ \mathsf{get}\ e_1'\ e_2\rangle \hookrightarrow \sigma_c$, and
    * $\langle S;\ \{d_1\}\rangle \hookrightarrow \sigma_c$.

    Choose $\sigma_c = \langle S;\ \{d_1\}\rangle$. We have from E-REFL that $\langle S;\ \{d_1\}\rangle \hookrightarrow \langle S;\ \{d_1\}\rangle$, so it remains to show that $\langle S_1;\ \mathsf{get}\ e_1'\ e_2\rangle \hookrightarrow \langle S;\ \{d_1\}\rangle$.
    From the premise of E-GET-1, we have that $\langle S;\ e_1\rangle \hookrightarrow \langle S_1;\ e_1'\rangle$. But $e_1 = l$, a value, so it must be the case that $e_1 = e_1'$ and $S = S_1$. Therefore, $\langle S_1;\ \mathsf{get}\ e_1'\ e_2\rangle = \langle S;\ \mathsf{get}\ e_1\ e_2\rangle$. Further, since $e_1 = l$ and $e_2 = Q$, $\langle S_1;\ \mathsf{get}\ e_1'\ e_2\rangle = \langle S;\ \mathsf{get}\ l\ Q\rangle$. So we have only to show that $\langle S;\ \mathsf{get}\ l\ Q\rangle \hookrightarrow \langle S;\ \{d_1\}\rangle$, which is immediate by E-GETVAL, since all of the premises hold.
  - E-GET-1/E-GETERR-1:
    Analogous to E-PUT-1/E-PUTERR-1.
  - E-GET-1/E-GETERR-2:
    Analogous to E-PUT-1/E-PUTERR-2.

A.6.6 E-GET-2

- E-GET-2: $\sigma = \langle S;\ \mathsf{get}\ e_1\ e_2\rangle$, and $\sigma_a = \langle S_1;\ \mathsf{get}\ e_1\ e_2'\rangle$.

  Given:

  - $\langle S;\ \mathsf{get}\ e_1\ e_2\rangle \hookrightarrow \langle S_2;\ \mathsf{get}\ e_1\ e_2'\rangle$, and
  - $\langle S;\ \mathsf{get}\ e_1\ e_2\rangle \hookrightarrow \sigma_b$.

  To show: There exists σ_c such that

  - $\langle S_2;\ \mathsf{get}\ e_1\ e_2'\rangle \hookrightarrow \sigma_c$, and
  - $\sigma_b \hookrightarrow \sigma_c$.

  From the premise of E-GET-2, we have that $\langle S;\ e_2\rangle \hookrightarrow \langle S_2;\ e_2'\rangle$.

  We proceed by subcases, on the last rule in the derivation of $\langle S;\ \mathsf{get}\ e_1\ e_2\rangle \hookrightarrow \sigma_b$. By the operational semantics, there are six possibilities: E-GET-2/E-REFL, E-GET-2/E-GET-1, E-GET-2/E-GET-2, E-GET-2/E-GETVAL, E-GET-2/E-GETERR-1 and E-GET-2/E-GETERR-2.

  - E-GET-2/E-REFL:
    Analogous to the E-REFL/E-GET-2 case, with σ_a and σ_b reversed.
  - E-GET-2/E-GET-1:
    Analogous to E-PUT-2/E-PUT-1.
  - E-GET-2/E-GET-2:
    Analogous to E-PUT-2/E-PUT-2.
  - E-GET-2/E-GETVAL:
    In this case, we have the following facts:
    * $\langle S;\ \mathsf{get}\ e_1\ e_2\rangle = \langle S;\ \mathsf{get}\ l\ Q\rangle$,
    * $\sigma_b = \langle S;\ \{d_1\}\rangle$, and
    * $S(l) = d_2 \wedge \mathit{incomp}(Q) \wedge Q \subseteq D \wedge d_1 \in Q \wedge d_1 \sqsubseteq d_2$ (from the premises of E-GETVAL).

    We're required to show that there exists σ_c such that
    * $\langle S_2;\ \mathsf{get}\ e_1\ e_2'\rangle \hookrightarrow \sigma_c$, and
    * $\langle S;\ \{d_1\}\rangle \hookrightarrow \sigma_c$.

    Choose $\sigma_c = \langle S;\ \{d_1\}\rangle$. We have from E-REFL that $\langle S;\ \{d_1\}\rangle \hookrightarrow \langle S;\ \{d_1\}\rangle$, so it remains to show that $\langle S_2;\ \mathsf{get}\ e_1\ e_2'\rangle \hookrightarrow \langle S;\ \{d_1\}\rangle$.
    From the premise of E-GET-2, we have that $\langle S;\ e_2\rangle \hookrightarrow \langle S_2;\ e_2'\rangle$. But $e_2 = Q$, a value, so it must be the case that $e_2 = e_2'$ and $S = S_2$. Therefore, $\langle S_2;\ \mathsf{get}\ e_1\ e_2'\rangle = \langle S;\ \mathsf{get}\ e_1\ e_2\rangle$. Further, since $e_1 = l$ and $e_2 = Q$, $\langle S_2;\ \mathsf{get}\ e_1\ e_2'\rangle = \langle S;\ \mathsf{get}\ l\ Q\rangle$. So we have only to show that $\langle S;\ \mathsf{get}\ l\ Q\rangle \hookrightarrow \langle S;\ \{d_1\}\rangle$, which is immediate by E-GETVAL, since all of the premises hold.
  - E-GET-2/E-GETERR-1:
    Analogous to E-PUT-2/E-PUTERR-1.
  - E-GET-2/E-GETERR-2:
    Analogous to E-PUT-2/E-PUTERR-2.

A.6.7 E-CONVERT

- E-CONVERT: $\sigma = \langle S;\ \mathsf{convert}\ e\rangle$, and $\sigma_a = \langle S';\ \mathsf{convert}\ e'\rangle$.

  Given:

  - $\langle S;\ \mathsf{convert}\ e\rangle \hookrightarrow \langle S';\ \mathsf{convert}\ e'\rangle$, and
  - $\langle S;\ \mathsf{convert}\ e\rangle \hookrightarrow \sigma_b$.

  To show: There exists σ_c such that

  - $\langle S';\ \mathsf{convert}\ e'\rangle \hookrightarrow \sigma_c$, and
  - $\sigma_b \hookrightarrow \sigma_c$.

  From the premise of E-CONVERT, we have that $\langle S;\ e\rangle \hookrightarrow \langle S';\ e'\rangle$.

  We proceed by subcases, on the last rule in the derivation of $\langle S;\ \mathsf{convert}\ e\rangle \hookrightarrow \sigma_b$. By the operational semantics, there are four possibilities: E-CONVERT/E-REFL, E-CONVERT/E-CONVERT, E-CONVERT/E-CONVERTVAL, and E-CONVERT/E-CONVERTERR.

  - E-CONVERT/E-REFL:
    Analogous to the E-REFL/E-CONVERT case, with σ_a and σ_b reversed.
  - E-CONVERT/E-CONVERT:
    In this case, we have the following facts:
    * $\sigma_b = \langle S_b;\ \mathsf{convert}\ e_b\rangle$, and
    * $\langle S;\ e\rangle \hookrightarrow \langle S_b;\ e_b\rangle$ (from the premise of E-CONVERT).

    Since $\langle S;\ e\rangle \hookrightarrow \langle S';\ e'\rangle$ and $\langle S;\ e\rangle \hookrightarrow \langle S_b;\ e_b\rangle$, we have by IH that there exists σ'_c such that $\langle S';\ e'\rangle \hookrightarrow \sigma'_c$ and $\langle S_b;\ e_b\rangle \hookrightarrow \sigma'_c$. Either σ'_c is **error**, or it is some non-**error** configuration $\langle S'_c;\ e'_c\rangle$.
    We're required to show that there exists σ_c such that
    * $\langle S';\ \mathsf{convert}\ e'\rangle \hookrightarrow \sigma_c$, and
    * $\langle S_b;\ \mathsf{convert}\ e_b\rangle \hookrightarrow \sigma_c$.

    We consider the following possibilities, one of which must hold.

    1. $\sigma'_c = \textbf{error}$.
       Then, since $\langle S';\ e'\rangle \hookrightarrow \textbf{error}$, we have by E-CONVERTERR that $\langle S';\ \mathsf{convert}\ e'\rangle \hookrightarrow \textbf{error}$. Likewise, since $\langle S_b;\ e_b\rangle \hookrightarrow \textbf{error}$, we have by E-CONVERTERR that $\langle S_b;\ \mathsf{convert}\ e_b\rangle \hookrightarrow \textbf{error}$. Therefore $\sigma_c = \textbf{error}$.
    2. $\sigma'_c = \langle S'_c;\ e'_c\rangle$.
       Then, since $\langle S';\ e'\rangle \hookrightarrow \langle S'_c;\ e'_c\rangle$, we have by E-CONVERT that $\langle S';\ \mathsf{convert}\ e'\rangle \hookrightarrow \langle S'_c;\ \mathsf{convert}\ e'_c\rangle$. Likewise, since $\langle S_b;\ e_b\rangle \hookrightarrow \langle S'_c;\ e'_c\rangle$, we have by E-CONVERT that $\langle S_b;\ \mathsf{convert}\ e_b\rangle \hookrightarrow \langle S'_c;\ \mathsf{convert}\ e'_c\rangle$. Therefore $\sigma_c = \langle S'_c;\ \mathsf{convert}\ e'_c\rangle$.
  - E-CONVERT/E-CONVERTVAL:
    In this case, we have the following facts:
    * $\langle S;\ \mathsf{convert}\ e\rangle = \langle S;\ \mathsf{convert}\ Q\rangle$, and
    * $\sigma_b = \langle S;\ \delta(Q)\rangle$.

    We're required to show that there exists σ_c such that
    * $\langle S';\ \mathsf{convert}\ e'\rangle \hookrightarrow \sigma_c$, and
    * $\langle S;\ \delta(Q)\rangle \hookrightarrow \sigma_c$.

Choose $\sigma_c = \langle S;\ \delta(Q)\rangle$. We have from E-REFL that $\langle S;\ \delta(Q)\rangle \longhookrightarrow \langle S;\ \delta(Q)\rangle$, so it remains to show that $\langle S';\ \mathsf{convert}\ e'\rangle \longhookrightarrow \langle S;\ \delta(Q)\rangle$.

From the premise of E-CONVERT, we have that $\langle S;\ e\rangle \longhookrightarrow \langle S';\ e'\rangle$. But $e = Q$, a value, so it must be the case that $e = e'$ and $S = S'$. Therefore, $\langle S';\ \mathsf{convert}\ e'\rangle = \langle S;\ \mathsf{convert}\ Q\rangle$. So we have only to show that $\langle S;\ \mathsf{convert}\ Q\rangle \longhookrightarrow \langle S;\ \delta(Q)\rangle$, which is immediate by E-CONVERTVAL.

– E-CONVERT/E-CONVERTERR:

In this case, we have the following facts:

* $\sigma_b = \textbf{error}$, and
* $\langle S;\ e\rangle \longhookrightarrow \textbf{error}$ (from the premise of E-CONVERTERR).

We're required to show that there exists σ_c such that

* $\langle S';\ \mathsf{convert}\ e'\rangle \longhookrightarrow \sigma_c$, and
* $\textbf{error} \longhookrightarrow \sigma_c$.

Choose $\sigma_c = \textbf{error}$. We have immediately that $\textbf{error} \longhookrightarrow \textbf{error}$ by E-REFLERR, so it remains to show that $\langle S';\ \mathsf{convert}\ e'\rangle \longhookrightarrow \textbf{error}$.

Since $\langle S;\ e\rangle \longhookrightarrow \textbf{error}$ and $\langle S;\ e\rangle \longhookrightarrow \langle S';\ e'\rangle$ (from the premise of E-CONVERT, above), we have by IH that there exists σ'_c such that $\textbf{error} \longhookrightarrow \sigma'_c$ and $\langle S';\ e'\rangle \longhookrightarrow \sigma'_c$. Since **error** can only step to **error**, $\sigma'_c = \textbf{error}$.

Therefore, $\langle S';\ e'\rangle \longhookrightarrow \textbf{error}$, so we have that $\langle S';\ \mathsf{convert}\ e'\rangle \longhookrightarrow \textbf{error}$ by E-CONVERTERR, as we were required to show.

A.6.8 E-BETA

- E-BETA: $\sigma = \langle S;\ (\lambda x.\, e)\ v\rangle$, and $\sigma_a = \langle S;\ e[x := v]\rangle$.

  Given:

  – $\langle S;\ (\lambda x.\, e)\ v\rangle \longhookrightarrow \langle S;\ e[x := v]\rangle$, and
  – $\langle S;\ (\lambda x.\, e)\ v\rangle \longhookrightarrow \sigma_b$.

  To show: There exists σ_c such that

  – $\langle S;\ e[x := v]\rangle \longhookrightarrow \sigma_c$, and
  – $\sigma_b \longhookrightarrow \sigma_c$.

  We proceed by subcases, on the last rule in the derivation of $\langle S;\ (\lambda x.\, e)\ v\rangle \longhookrightarrow \sigma_b$. By the operational semantics, there are six possibilities: E-BETA/E-REFL, E-BETA/E-PARAPP, E-BETA/E-BETA, E-BETA/E-APPERR-1, E-BETA/E-APPERR-2, and E-BETA/E-PARAPPERR.

  – E-BETA/E-REFL:

    Analogous to the E-REFL/E-BETA case, with σ_a and σ_b reversed.

  – E-BETA/E-PARAPP:

    Analogous to the E-PARAPP/E-BETA case, with σ_a and σ_b reversed.

  – E-BETA/E-BETA:

    In this case, by the operational semantics, $\sigma_b = \langle S;\ e[x := v]\rangle$. Since $\sigma_a = \sigma_b = \langle S;\ e[x := v]\rangle$, choose $\sigma_c = \langle S;\ e[x := v]\rangle$. By E-REFL, both σ_a and σ_b step to σ_c, as we were required to show.

  – E-BETA/E-APPERR-1:

    For this case to occur, we would need to have $\langle S;\ (\lambda x.\, e)\rangle \longhookrightarrow \textbf{error}$ (from the premise of E-APPERR-1. But $(\lambda x.\, e)$ is a value (and $S \neq \top_S$), so $\langle S;\ (\lambda x.\, e)\rangle$ can only step to $\langle S;\ (\lambda x.\, e)\rangle$, not **error**. Therefore, this case cannot occur.

– E-BETA/E-APPERR-2:
For this case to occur, we would need to have $\langle S;\ v\rangle \hookrightarrow$ **error** (from the premise of E-APPERR-2. But v is a value (and $S \neq \top_S$), so $\langle S;\ v\rangle$ can only step to $\langle S;\ v\rangle$, not **error**. Therefore, this case cannot occur.
– E-BETA/E-PARAPPERR:
For this case to occur, then by the premises of E-PARAPPERR, we would need to have $\langle S;\ (\lambda x.\, e)\rangle$ step to some configuration $\langle S_1;\ e_1'\rangle$ and to have $\langle S;\ v\rangle$ step to some configuration $\langle S_2;\ e_2'\rangle$, where $S_1 \sqcup_S S_2 = \top_S$. But $(\lambda x.\, e)$ and v are values (and $S \neq \top_S$), so $\langle S;\ (\lambda x.\, e)\rangle$ can only step to $\langle S;\ (\lambda x.\, e)\rangle$ and $\langle S;\ v\rangle$ can only step to $\langle S;\ v\rangle$. Therefore $S_1 = S_2 = S$, so $S_1 \sqcup_S S_2 = S \neq \top_S$, and so this case cannot occur.

A.6.9 E-NEW

- E-NEW: $\sigma = \langle S;\ \mathsf{new}\rangle$, and $\sigma_a = \langle S[l \mapsto \bot];\ l\rangle$.
Given:
– $\langle S;\ \mathsf{new}\rangle \hookrightarrow \langle S[l \mapsto \bot];\ l\rangle$, and
– $\langle S;\ \mathsf{new}\rangle \hookrightarrow \sigma_b$.
To show: There exists σ_c such that
– $\langle S[l \mapsto \bot];\ l\rangle \hookrightarrow \sigma_c$, and
– $\sigma_b \hookrightarrow \sigma_c$.
We proceed by subcases, on the last rule in the derivation of $\langle S;\ \mathsf{new}\rangle \hookrightarrow \sigma_b$. By the operational semantics, there are two possibilities: E-NEW/E-REFL and E-NEW/E-NEW.
– E-NEW/E-REFL:
Analogous to the E-REFL/E-NEW case, with σ_a and σ_b reversed.
– E-NEW/E-NEW:
In this case, $\sigma_b = \langle S[l' \mapsto \bot];\ l'\rangle$.
To show: There exists σ_c such that
∗ $\langle S[l \mapsto \bot];\ l\rangle \hookrightarrow \sigma_c$, and
∗ $\langle S[l' \mapsto \bot];\ l'\rangle \hookrightarrow \sigma_c$.
One of the following two possibilities must hold:
∗ $l' = l$.
In this case, both $\langle S[l \mapsto \bot];\ l\rangle$ and $\langle S[l' \mapsto \bot];\ l'\rangle$ step to $\langle S[l \mapsto \bot];\ l\rangle$ by E-REFL. Therefore $\sigma_c = \langle S[l \mapsto \bot];\ l\rangle$.
∗ $l' \neq l$.
In this case, $dom(S[l \mapsto \bot]) - dom(S) = \{l\}$, and $l' \notin dom(S[l \mapsto \bot])$ (since, by the side condition of E-NEW, $l \notin dom(S)$, and since $l' \neq l$. Therefore, by Definition 8, $\langle S[l \mapsto \bot];\ l\rangle$ is a safe renaming of $\langle S[l' \mapsto \bot];\ l'\rangle$. Stepping both configurations by E-REFL, we have that $\sigma_c = \langle S[l \mapsto \bot];\ l\rangle$ or a safe renaming thereof. Therefore the case holds up to safe renamings of σ_c.

A.6.10 E-PUTVAL

- E-PUTVAL: $\sigma = \langle S;\ \mathsf{put}\ l\ \{d_1\}\rangle$, and $\sigma_a = \langle S[l \mapsto d_1 \sqcup d_2];\ \{\}\rangle$.
Given:
– $\langle S;\ \mathsf{put}\ l\ \{d_1\}\rangle \hookrightarrow \langle S[l \mapsto d_1 \sqcup d_2];\ \{\}\rangle$, and

– $\langle S;\ \mathsf{put}\ l\ \{d_1\}\rangle \longrightarrow \sigma_b$.

To show: There exists σ_c such that

– $\langle S[l \mapsto d_1 \sqcup d_2];\ \{\}\rangle \longrightarrow \sigma_c$, and
– $\sigma_b \longrightarrow \sigma_c$.

We proceed by subcases, on the last rule in the derivation of $\langle S;\ \mathsf{put}\ l\ \{d_1\}\rangle \longrightarrow \sigma_b$. By the operational semantics, there are seven possibilities: E-PUTVAL/E-REFL, E-PUTVAL/E-PUT-1, E-PUTVAL/E-PUT-2, E-PUTVAL/E-PUTVAL, E-PUTVAL/E-PUTERR-1, E-PUTVAL/E-PUTERR-2, and E-PUTVAL/E-PUTVALERR.

– E-PUTVAL/E-REFL:
  Analogous to the E-REFL/E-PUTVAL case, with σ_a and σ_b reversed.
– E-PUTVAL/E-PUT-1:
  Analogous to the E-PUT-1/E-PUTVAL case, with σ_a and σ_b reversed.
– E-PUTVAL/E-PUT-2:
  Analogous to the E-PUT-2/E-PUTVAL case, with σ_a and σ_b reversed.
– E-PUTVAL/E-PUTVAL:
  In this case, by the operational semantics, $\sigma_b = \langle S[l \mapsto d_1 \sqcup d_2];\ \{\}\rangle$. Since $\sigma_a = \sigma_b = \langle S[l \mapsto d_1 \sqcup d_2];\ \{\}\rangle$, choose $\sigma_c = \langle S[l \mapsto d_1 \sqcup d_2];\ \{\}\rangle$. By E-REFL, both σ_a and σ_b step to σ_c, as we were required to show.
– E-PUTVAL/E-PUTERR-1:
  For this case to occur, we would need to have $\langle S;\ l\rangle \longrightarrow$ **error** (from the premise of E-PUTERR-1. But l is a value (and $S \neq \top_S$), so $\langle S;\ l\rangle$ can only step to $\langle S;\ l\rangle$, not **error**. Therefore, this case cannot occur.
– E-PUTVAL/E-PUTERR-2:
  For this case to occur, we would need to have $\langle S;\ \{d_1\}\rangle \longrightarrow$ **error** (from the premise of E-PUTERR-1. But $\{d_1\}$ is a value (and $S \neq \top_S$), so $\langle S;\ \{d_1\}\rangle$ can only step to $\langle S;\ \{d_2\}\rangle$, not **error**. Therefore, this case cannot occur.
– E-PUTVAL/E-PUTVALERR:
  For this case to occur, we would need to have $d_1 \sqcup d_2 = \top$ (from the last premise of E-PUTVALERR). But we have that $d_1 \sqcup d_2 \neq \top$ from the last premise of E-PUTVAL. Therefore, this case cannot occur.

A.6.11 E-GETVAL

- E-GETVAL: $\sigma = \langle S;\ \mathsf{get}\ l\ Q\rangle$, and $\sigma_a = \langle S;\ \{d_1\}\rangle$.

  Given:

  – $\langle S;\ \mathsf{get}\ l\ Q\rangle \longrightarrow \langle S;\ \{d_1\}\rangle$, and
  – $\langle S;\ \mathsf{get}\ l\ Q\rangle \longrightarrow \sigma_b$.

  To show: There exists σ_c such that

  – $\langle S;\ \{d_1\}\rangle \longrightarrow \sigma_c$, and
  – $\sigma_b \longrightarrow \sigma_c$.

  We proceed by subcases, on the last rule in the derivation of $\langle S;\ \mathsf{get}\ l\ Q\rangle \longrightarrow \sigma_b$. By the operational semantics, there are six possibilities: E-GETVAL/E-REFL, E-GETVAL/E-GET-1, E-GETVAL/E-GET-2, E-GETVAL/E-GETVAL, E-GETVAL/E-GETERR-1, and E-GETVAL/E-GETERR-2.

– E-GETVAL/E-REFL:
Analogous to the E-REFL/E-GETVAL case, with σ_a and σ_b reversed.

– E-GETVAL/E-GET-1:
Analogous to the E-GET-1/E-GETVAL case, with σ_a and σ_b reversed.

– E-GETVAL/E-GET-2:
Analogous to the E-GET-2/E-GETVAL case, with σ_a and σ_b reversed.

– E-GETVAL/E-GETVAL:
In this case, by the operational semantics, $\sigma_b = \langle S;\ \{d_1\}\rangle$. Since $\sigma_a = \sigma_b = \langle S;\ \{d_1\}\rangle$, choose $\sigma_c = \langle S;\ \{d_1\}\rangle$. By E-REFL, both σ_a and σ_b step to σ_c, as we were required to show.

– E-GETVAL/E-GETERR-1:
For this case to occur, we would need to have $\langle S;\ l\rangle \hookrightarrow$ **error** (from the premise of E-GETERR-1. But l is a value (and $S \neq \top_S$), so $\langle S;\ l\rangle$ can only step to $\langle S;\ l\rangle$, not **error**. Therefore, this case cannot occur.

– E-GETVAL/E-GETERR-2:
For this case to occur, we would need to have $\langle S;\ Q\rangle \hookrightarrow$ **error** (from the premise of E-GETERR-2. But Q is a value (and $S \neq \top_S$), so $\langle S;\ Q\rangle$ can only step to $\langle S;\ Q\rangle$, not **error**. Therefore, this case cannot occur.

A.6.12 E-CONVERTVAL

- E-CONVERTVAL: $\sigma = \langle S;\ \mathsf{convert}\ Q\rangle$, and $\sigma_a = \langle S;\ \delta(Q)\rangle$.
Given:

  – $\langle S;\ \mathsf{convert}\ Q\rangle \hookrightarrow \langle S;\ \delta(Q)\rangle$, and
  – $\langle S;\ \mathsf{convert}\ Q\rangle \hookrightarrow \sigma_b$.

  To show: There exists σ_c such that

  – $\langle S;\ \delta(Q)\rangle \hookrightarrow \sigma_c$, and
  – $\sigma_b \hookrightarrow \sigma_c$.

  We proceed by subcases, on the last rule in the derivation of $\langle S;\ \mathsf{convert}\ Q\rangle \hookrightarrow \sigma_b$. By the operational semantics, there are four possibilities: E-CONVERTVAL/E-REFL, E-CONVERTVAL/E-CONVERT, E-CONVERTVAL/E-CONVERTVAL, and E-CONVERTVAL/E-CONVERTERR.

  – E-CONVERTVAL/E-REFL:
  Analogous to the E-REFL/E-CONVERTVAL case, with σ_a and σ_b reversed.

  – E-CONVERTVAL/E-CONVERT:
  Analogous to the E-CONVERT/E-CONVERTVAL case, with σ_a and σ_b reversed.

  – E-CONVERTVAL/E-CONVERTVAL:
  In this case, by the operational semantics, $\sigma_b = \langle S;\ \delta(Q)\rangle$. Since $\sigma_a = \sigma_b = \langle S;\ \delta(Q)\rangle$, choose $\sigma_c = \langle S;\ \delta(Q)\rangle$. By E-REFL, both σ_a and σ_b step to σ_c, as we were required to show.

  – E-CONVERTVAL/E-CONVERTERR:
  For this case to occur, we would need to have $\langle S;\ Q\rangle \hookrightarrow$ **error** (from the premise of E-CONVERTERR. But Q is a value (and $S \neq \top_S$), so $\langle S;\ Q\rangle$ can only step to $\langle S;\ Q\rangle$, not **error**. Therefore, this case cannot occur.

A.6.13 E-REFLERR

- E-REFLERR: $\sigma = \textbf{error}$, and $\sigma_a = \textbf{error}$.

  Given:

  - $\textbf{error} \hookrightarrow \textbf{error}$, and
  - $\textbf{error} \hookrightarrow \sigma_b$.

  To show: There exists σ_c such that

  - $\textbf{error} \hookrightarrow \sigma_c$, and
  - $\sigma_b \hookrightarrow \sigma_c$.

  For all subcases E-REFLERR/*, choose $\sigma_c = \textbf{error}$.

  To show:

  - $\textbf{error} \hookrightarrow \textbf{error}$, which is immediate from E-REFLERR, and
  - $\sigma_b \hookrightarrow \textbf{error}$, which follows from the fact that $\textbf{error} \hookrightarrow \sigma_b$, so since **error** can only step to **error**, $\sigma_b = \textbf{error}$.

A.6.14 E-APPERR-1

- E-APPERR-1: $\sigma = \langle S;\ e_1\ e_2 \rangle$, and $\sigma_a = \textbf{error}$.

  Given:

  - $\langle S;\ e_1\ e_2 \rangle \hookrightarrow \textbf{error}$, and
  - $\langle S;\ e_1\ e_2 \rangle \hookrightarrow \sigma_b$.

  To show: There exists σ_c such that

  - $\textbf{error} \hookrightarrow \sigma_c$, and
  - $\sigma_b \hookrightarrow \sigma_c$.

  Choose $\sigma_c = \textbf{error}$. We have immediately that $\textbf{error} \hookrightarrow \textbf{error}$ by E-REFLERR, so it remains to show that $\sigma_b \hookrightarrow \textbf{error}$.

  We proceed by subcases, on the last rule in the derivation of $\langle S;\ e_1\ e_2 \rangle \hookrightarrow \sigma_b$. By the operational semantics, there are seven possibilities: E-APPERR-1/E-REFL, E-APPERR-1/E-PARAPP, E-APPERR-1/E-BETA, E-APPERR-1/E-REFLERR, E-APPERR-1/E-APPERR-1, E-APPERR-1/E-APPERR-2, and E-APPERR-1/E-PARAPPERR.

  - E-APPERR-1/E-REFL:
    Analogous to the E-REFL/E-APPERR-1 case, with σ_a and σ_b reversed.
  - E-APPERR-1/E-PARAPP:
    Analogous to the E-PARAPP/E-APPERR-1 case, with σ_a and σ_b reversed.
  - E-APPERR-1/E-BETA:
    Analogous to the E-BETA/E-APPERR-1 case, with σ_a and σ_b reversed.
  - E-APPERR-1/E-REFLERR:
    Analogous to the E-REFLERR/E-APPERR-1 case, with σ_a and σ_b reversed.

- E-APPERR-1/E-APPERR-1:
  Choose $\sigma_c = \textbf{error}$. By E-APPERR-1, $\sigma_b = \textbf{error}$, so by E-REFLERR, both σ_a and σ_b step to **error**, as desired.
- E-APPERR-1/E-APPERR-2:
  Choose $\sigma_c = \textbf{error}$. By E-APPERR-2, $\sigma_b = \textbf{error}$, so by E-REFLERR, both σ_a and σ_b step to **error**, as desired.
- E-APPERR-1/E-PARAPPERR:
  Choose $\sigma_c = \textbf{error}$. By E-PARAPPERR, $\sigma_b = \textbf{error}$, so by E-REFLERR, both σ_a and σ_b step to **error**, as desired.

A.6.15 E-APPERR-2

- E-APPERR-2: $\sigma = \langle S;\ e_1\ e_2 \rangle$, and $\sigma_a = \textbf{error}$.

  Given:

  - $\langle S;\ e_1\ e_2 \rangle \hookrightarrow \textbf{error}$, and
  - $\langle S;\ e_1\ e_2 \rangle \hookrightarrow \sigma_b$.

  To show: There exists σ_c such that

  - $\textbf{error} \hookrightarrow \sigma_c$, and
  - $\sigma_b \hookrightarrow \sigma_c$.

  Choose $\sigma_c = \textbf{error}$. We have immediately that $\textbf{error} \hookrightarrow \textbf{error}$ by E-REFLERR, so it remains to show that $\sigma_b \hookrightarrow \textbf{error}$.

  We proceed by subcases, on the last rule in the derivation of $\langle S;\ e_1\ e_2 \rangle \hookrightarrow \sigma_b$. By the operational semantics, there are seven possibilities: E-APPERR-2/E-REFL, E-APPERR-2/E-PARAPP, E-APPERR-2/E-BETA, E-APPERR-2/E-REFLERR, E-APPERR-2/E-APPERR-1, E-APPERR-2/E-APPERR-2, and E-APPERR-2/E-PARAPPERR.

  - E-APPERR-2/E-REFL:
    Analogous to the E-REFL/E-APPERR-2 case, with σ_a and σ_b reversed.
  - E-APPERR-2/E-PARAPP:
    Analogous to the E-PARAPP/E-APPERR-2 case, with σ_a and σ_b reversed.
  - E-APPERR-2/E-BETA:
    Analogous to the E-BETA/E-APPERR-2 case, with σ_a and σ_b reversed.
  - E-APPERR-2/E-REFLERR:
    Analogous to the E-REFLERR/E-APPERR-2 case, with σ_a and σ_b reversed.
  - E-APPERR-2/E-APPERR-1:
    Analogous to the E-APPERR-1/E-APPERR-2 case, with σ_a and σ_b reversed.
  - E-APPERR-2/E-APPERR-2:
    Choose $\sigma_c = \textbf{error}$. By E-APPERR-2, $\sigma_b = \textbf{error}$, so by E-REFLERR, both σ_a and σ_b step to **error**, as desired.
  - E-APPERR-2/E-PARAPPERR:
    Choose $\sigma_c = \textbf{error}$. By E-PARAPPERR, $\sigma_b = \textbf{error}$, so by E-REFLERR, both σ_a and σ_b step to **error**, as desired.

A.6.16 E-PARAPPERR

- E-PARAPPERR: $\sigma = \langle S;\ e_1\ e_2 \rangle$, and $\sigma_a = \textbf{error}$.

  Given:

  - $\langle S;\ e_1\ e_2 \rangle \hookrightarrow \textbf{error}$, and
  - $\langle S;\ e_1\ e_2 \rangle \hookrightarrow \sigma_b$.

  To show: There exists σ_c such that

  - $\textbf{error} \hookrightarrow \sigma_c$, and
  - $\sigma_b \hookrightarrow \sigma_c$.

  Choose $\sigma_c = \textbf{error}$. We have immediately that $\textbf{error} \hookrightarrow \textbf{error}$ by E-REFLERR, so it remains to show that $\sigma_b \hookrightarrow \textbf{error}$.

  We proceed by subcases, on the last rule in the derivation of $\langle S;\ e_1\ e_2 \rangle \hookrightarrow \sigma_b$. By the operational semantics, there are seven possibilities: E-PARAPPERR/E-REFL, E-PARAPPERR/E-PARAPP, E-PARAPPERR/E-BETA, E-PARAPPERR/E-REFLERR, E-PARAPPERR/E-APPERR-1, E-PARAPPERR/E-APPERR-2, and E-PARAPPERR/E-PARAPPERR.

  - E-PARAPPERR/E-REFL:
    Analogous to the E-REFL/E-PARAPPERR case, with σ_a and σ_b reversed.
  - E-PARAPPERR/E-PARAPP:
    Analogous to the E-PARAPP/E-PARAPPERR case, with σ_a and σ_b reversed.
  - E-PARAPPERR/E-BETA:
    Analogous to the E-BETA/E-PARAPPERR case, with σ_a and σ_b reversed.
  - E-PARAPPERR/E-REFLERR:
    Analogous to the E-REFLERR/E-PARAPPERR case, with σ_a and σ_b reversed.
  - E-PARAPPERR/E-APPERR-1:
    Analogous to the E-APPERR-1/E-PARAPPERR case, with σ_a and σ_b reversed.
  - E-PARAPPERR/E-APPERR-2:
    Analogous to the E-APPERR-2/E-PARAPPERR case, with σ_a and σ_b reversed.
  - E-PARAPPERR/E-PARAPPERR:
    Choose $\sigma_c = \textbf{error}$. By E-PARAPPERR, $\sigma_b = \textbf{error}$, so by E-REFLERR, both σ_a and σ_b step to **error**, as desired.

A.6.17 E-PUTERR-1

- E-PUTERR-1: $\sigma = \langle S;\ \mathsf{put}\ e_1\ e_2 \rangle$, and $\sigma_a = \textbf{error}$.

  Given:

  - $\langle S;\ \mathsf{put}\ e_1\ e_2 \rangle \hookrightarrow \textbf{error}$, and
  - $\langle S;\ \mathsf{put}\ e_1\ e_2 \rangle \hookrightarrow \sigma_b$.

  To show: There exists σ_c such that

  - $\textbf{error} \hookrightarrow \sigma_c$, and
  - $\sigma_b \hookrightarrow \sigma_c$.

Choose $\sigma_c = \textbf{error}$. We have immediately that $\textbf{error} \longrightarrow \textbf{error}$ by E-REFLERR, so it remains to show that $\sigma_b \longrightarrow \textbf{error}$.

We proceed by subcases, on the last rule in the derivation of $\langle S; \mathsf{put}\ e_1\ e_2\rangle \longrightarrow \sigma_b$. By the operational semantics, there are eight possibilities: E-PUTERR-1/E-REFL, E-PUTERR-1/E-PUT-1, E-PUTERR-1/E-PUT-2, E-PUTERR-1/E-PUTVAL, E-PUTERR-1/E-REFLERR, E-PUTERR-1/E-PUTERR-1, E-PUTERR-1/E-PUTERR-2, and E-PUTERR-1/E-PUTVALERR.

- E-PUTERR-1/E-REFL:
  Analogous to the E-REFL/E-PUTERR-1 case, with σ_a and σ_b reversed.
- E-PUTERR-1/E-PUT-1:
  Analogous to the E-PUT-1/E-PUTERR-1 case, with σ_a and σ_b reversed.
- E-PUTERR-1/E-PUT-2:
  Analogous to the E-PUT-2/E-PUTERR-1 case, with σ_a and σ_b reversed.
- E-PUTERR-1/E-PUTVAL:
  Analogous to the E-PUTVAL/E-PUTERR-1 case, with σ_a and σ_b reversed.
- E-PUTERR-1/E-REFLERR:
  Analogous to the E-REFLERR/E-PUTERR-1 case, with σ_a and σ_b reversed.
- E-PUTERR-1/E-PUTERR-1:
  Choose $\sigma_c = \textbf{error}$. By E-PUTERR-1, $\sigma_b = \textbf{error}$, so by E-REFLERR, both σ_a and σ_b step to **error**, as desired.
- E-PUTERR-1/E-PUTERR-2:
  Choose $\sigma_c = \textbf{error}$. By E-PUTERR-2, $\sigma_b = \textbf{error}$, so by E-REFLERR, both σ_a and σ_b step to **error**, as desired.
- E-PUTERR-1/E-PUTVALERR:
  Choose $\sigma_c = \textbf{error}$. By E-PUTVALERR, $\sigma_b = \textbf{error}$, so by E-REFLERR, both σ_a and σ_b step to **error**, as desired.

A.6.18 E-PUTERR-2

- E-PUTERR-2: $\sigma = \langle S; \mathsf{put}\ e_1\ e_2\rangle$, and $\sigma_a = \textbf{error}$.

  Given:

  - $\langle S; \mathsf{put}\ e_1\ e_2\rangle \longrightarrow \textbf{error}$, and
  - $\langle S; \mathsf{put}\ e_1\ e_2\rangle \longrightarrow \sigma_b$.

  To show: There exists σ_c such that

  - $\textbf{error} \longrightarrow \sigma_c$, and
  - $\sigma_b \longrightarrow \sigma_c$.

  Choose $\sigma_c = \textbf{error}$. We have immediately that $\textbf{error} \longrightarrow \textbf{error}$ by E-REFLERR, so it remains to show that $\sigma_b \longrightarrow \textbf{error}$.

  We proceed by subcases, on the last rule in the derivation of $\langle S; \mathsf{put}\ e_1\ e_2\rangle \longrightarrow \sigma_b$. By the operational semantics, there are eight possibilities: E-PUTERR-2/E-REFL, E-PUTERR-2/E-PUT-1, E-PUTERR-2/E-PUT-2, E-PUTERR-2/E-PUTVAL, E-PUTERR-2/E-REFLERR, E-PUTERR-2/E-PUTERR-1, E-PUTERR-2/E-PUTERR-2, and E-PUTERR-2/E-PUTVALERR.

- E-PUTERR-2/E-REFL:
  Analogous to the E-REFL/E-PUTERR-2 case, with σ_a and σ_b reversed.
- E-PUTERR-2/E-PUT-1:
  Analogous to the E-PUT-1/E-PUTERR-2 case, with σ_a and σ_b reversed.
- E-PUTERR-2/E-PUT-2:
  Analogous to the E-PUT-2/E-PUTERR-2 case, with σ_a and σ_b reversed.
- E-PUTERR-2/E-PUTVAL:
  Analogous to the E-PUTVAL/E-PUTERR-2 case, with σ_a and σ_b reversed.
- E-PUTERR-2/E-REFLERR:
  Analogous to the E-REFLERR/E-PUTERR-2 case, with σ_a and σ_b reversed.
- E-PUTERR-2/E-PUTERR-1:
  Analogous to the E-PUTERR-1/E-PUTERR-2 case, with σ_a and σ_b reversed.
- E-PUTERR-2/E-PUTERR-2:
  Choose $\sigma_c =$ **error**. By E-PUTERR-2, $\sigma_b =$ **error**, so by E-REFLERR, both σ_a and σ_b step to **error**, as desired.
- E-PUTERR-2/E-PUTVALERR:
  Choose $\sigma_c =$ **error**. By E-PUTVALERR, $\sigma_b =$ **error**, so by E-REFLERR, both σ_a and σ_b step to **error**, as desired.

A.6.19 E-GETERR-1

- E-GETERR-1: $\sigma = \langle S; \mathsf{put}\ e_1\ e_2 \rangle$, and $\sigma_a =$ **error**.

  Given:

  - $\langle S; \mathsf{put}\ e_1\ e_2 \rangle \hookrightarrow$ **error**, and
  - $\langle S; \mathsf{put}\ e_1\ e_2 \rangle \hookrightarrow \sigma_b$.

  To show: There exists σ_c such that

  - **error** $\hookrightarrow \sigma_c$, and
  - $\sigma_b \hookrightarrow \sigma_c$.

  Choose $\sigma_c =$ **error**. We have immediately that **error** $\hookrightarrow$ **error** by E-REFLERR, so it remains to show that $\sigma_b \hookrightarrow$ **error**.

  We proceed by subcases, on the last rule in the derivation of $\langle S; \mathsf{put}\ e_1\ e_2 \rangle \hookrightarrow \sigma_b$. By the operational semantics, there are seven possibilities: E-GETERR-1/E-REFL, E-GETERR-1/E-GET-1, E-GETERR-1/E-GET-2, E-GETERR-1/E-GETVAL, E-GETERR-1/E-REFLERR, E-GETERR-1/E-GETERR-1, and E-GETERR-1/E-GETERR-2.

  - E-GETERR-1/E-REFL:
    Analogous to the E-REFL/E-GETERR-1 case, with σ_a and σ_b reversed.
  - E-GETERR-1/E-GET-1:
    Analogous to the E-GET-1/E-GETERR-1 case, with σ_a and σ_b reversed.
  - E-GETERR-1/E-GET-2:
    Analogous to the E-GET-2/E-GETERR-1 case, with σ_a and σ_b reversed.
  - E-GETERR-1/E-GETVAL:
    Analogous to the E-GETVAL/E-GETERR-1 case, with σ_a and σ_b reversed.

- E-GETERR-1/E-REFLERR:
  Analogous to the E-REFLERR/E-GETERR-1 case, with σ_a and σ_b reversed.
- E-GETERR-1/E-GETERR-1:
  Choose $\sigma_c =$ **error**. By E-GETERR-1, $\sigma_b =$ **error**, so by E-REFLERR, both σ_a and σ_b step to **error**, as desired.
- E-GETERR-1/E-GETERR-2:
  Choose $\sigma_c =$ **error**. By E-GETERR-2, $\sigma_b =$ **error**, so by E-REFLERR, both σ_a and σ_b step to **error**, as desired.

A.6.20 E-GETERR-2

- E-GETERR-2: $\sigma = \langle S; \mathsf{get}\ e_1\ e_2 \rangle$, and $\sigma_a =$ **error**.

  Given:

  - $\langle S; \mathsf{get}\ e_1\ e_2 \rangle \hookrightarrow$ **error**, and
  - $\langle S; \mathsf{get}\ e_1\ e_2 \rangle \hookrightarrow \sigma_b$.

  To show: There exists σ_c such that

  - **error** $\hookrightarrow \sigma_c$, and
  - $\sigma_b \hookrightarrow \sigma_c$.

  Choose $\sigma_c =$ **error**. We have immediately that **error** $\hookrightarrow$ **error** by E-REFLERR, so it remains to show that $\sigma_b \hookrightarrow$ **error**.

  We proceed by subcases, on the last rule in the derivation of $\langle S; \mathsf{get}\ e_1\ e_2 \rangle \hookrightarrow \sigma_b$. By the operational semantics, there are seven possibilities: E-GETERR-2/E-REFL, E-GETERR-2/E-GET-1, E-GETERR-2/E-GET-2, E-GETERR-2/E-GETVAL, E-GETERR-2/E-REFLERR, E-GETERR-2/E-GETERR-1, and E-GETERR-2/E-GETERR-2.

  - E-GETERR-2/E-REFL:
    Analogous to the E-REFL/E-GETERR-2 case, with σ_a and σ_b reversed.
  - E-GETERR-2/E-GET-1:
    Analogous to the E-GET-1/E-GETERR-2 case, with σ_a and σ_b reversed.
  - E-GETERR-2/E-GET-2:
    Analogous to the E-GET-2/E-GETERR-2 case, with σ_a and σ_b reversed.
  - E-GETERR-2/E-GETVAL:
    Analogous to the E-GETVAL/E-GETERR-2 case, with σ_a and σ_b reversed.
  - E-GETERR-2/E-REFLERR:
    Analogous to the E-REFLERR/E-GETERR-2 case, with σ_a and σ_b reversed.
  - E-GETERR-2/E-GETERR-1:
    Analogous to the E-GETERR-1/E-GETERR-2 case, with σ_a and σ_b reversed.
  - E-GETERR-2/E-GETERR-2:
    Choose $\sigma_c =$ **error**. By E-GETERR-2, $\sigma_b =$ **error**, so by E-REFLERR, both σ_a and σ_b step to **error**, as desired.

A.6.21 E-CONVERTERR

- E-CONVERTERR: $\sigma = \langle S;\ \mathsf{convert}\ e\rangle$, and $\sigma_a = \textbf{error}$.

  Given:

  - $\langle S;\ \mathsf{convert}\ e\rangle \longhookrightarrow \textbf{error}$, and
  - $\langle S;\ \mathsf{convert}\ e\rangle \longhookrightarrow \sigma_b$.

  To show: There exists σ_c such that

  - $\textbf{error} \longhookrightarrow \sigma_c$, and
  - $\sigma_b \longhookrightarrow \sigma_c$.

  Choose $\sigma_c = \textbf{error}$. We have immediately that $\textbf{error} \longhookrightarrow \textbf{error}$ by E-REFLERR, so it remains to show that $\sigma_b \longhookrightarrow \textbf{error}$.

  We proceed by subcases, on the last rule in the derivation of $\langle S;\ \mathsf{convert}\ e\rangle \longhookrightarrow \sigma_b$. By the operational semantics, there are five possibilities: E-CONVERTERR/E-REFL, E-CONVERTERR/E-CONVERT, E-CONVERTERR/E-CONVERTVAL, E-CONVERTERR/E-REFLERR, and E-CONVERTERR/E-CONVERTERR.

  - E-CONVERTERR/E-REFL:
    Analogous to the E-REFL/E-CONVERTERR case, with σ_a and σ_b reversed.
  - E-CONVERTERR/E-CONVERT:
    Analogous to the E-CONVERT/E-CONVERTERR case, with σ_a and σ_b reversed.
  - E-CONVERTERR/E-CONVERTVAL:
    Analogous to the E-CONVERTVAL/E-CONVERTERR case, with σ_a and σ_b reversed.
  - E-CONVERTERR/E-REFLERR:
    Analogous to the E-REFLERR/E-CONVERTERR case, with σ_a and σ_b reversed.
  - E-CONVERTERR/E-CONVERTERR:
    Choose $\sigma_c = \textbf{error}$. By E-CONVERTERR, $\sigma_b = \textbf{error}$, so by E-REFLERR, both σ_a and σ_b step to **error**, as desired.

A.6.22 E-PUTVALERR

- E-PUTVALERR: $\sigma = \langle S;\ \mathsf{put}\ e_1\ e_2\rangle$, and $\sigma_a = \textbf{error}$.

  Given:

  - $\langle S;\ \mathsf{put}\ e_1\ e_2\rangle \longhookrightarrow \textbf{error}$, and
  - $\langle S;\ \mathsf{put}\ e_1\ e_2\rangle \longhookrightarrow \sigma_b$.

  To show: There exists σ_c such that

  - $\textbf{error} \longhookrightarrow \sigma_c$, and
  - $\sigma_b \longhookrightarrow \sigma_c$.

  Choose $\sigma_c = \textbf{error}$. We have immediately that $\textbf{error} \longhookrightarrow \textbf{error}$ by E-REFLERR, so it remains to show that $\sigma_b \longhookrightarrow \textbf{error}$.

  We proceed by subcases, on the last rule in the derivation of $\langle S;\ \mathsf{put}\ e_1\ e_2\rangle \longhookrightarrow \sigma_b$. By the operational semantics, there are eight possibilities: E-PUTVALERR/E-REFL, E-PUTVALERR/E-PUT-1, E-PUTVALERR/E-PUT-2, E-PUTVALERR/E-PUTVAL, E-PUTVALERR/E-REFLERR, E-PUTVALERR/E-PUTVALERR, E-PUTVALERR/E-PUTERR-2, and E-PUTVALERR/E-PUTVALERR.

- E-PUTVALERR/E-REFL:
  Analogous to the E-REFL/E-PUTVALERR case, with σ_a and σ_b reversed.
- E-PUTVALERR/E-PUT-1:
  Analogous to the E-PUT-1/E-PUTVALERR case, with σ_a and σ_b reversed.
- E-PUTVALERR/E-PUT-2:
  Analogous to the E-PUT-2/E-PUTVALERR case, with σ_a and σ_b reversed.
- E-PUTVALERR/E-PUTVAL:
  Analogous to the E-PUTVAL/E-PUTVALERR case, with σ_a and σ_b reversed.
- E-PUTVALERR/E-REFLERR:
  Analogous to the E-REFLERR/E-PUTVALERR case, with σ_a and σ_b reversed.
- E-PUTVALERR/E-PUTERR-1:
  Analogous to the E-PUTERR-1/E-PUTVALERR case, with σ_a and σ_b reversed.
- E-PUTVALERR/E-PUTERR-2:
  Analogous to the E-PUTERR-2/E-PUTVALERR case, with σ_a and σ_b reversed.
- E-PUTVALERR/E-PUTVALERR:
  Choose $\sigma_c =$ **error**. By E-PUTVALERR, $\sigma_b =$ **error**, so by E-REFLERR, both σ_a and σ_b step to **error**, as desired.

□

A.7 Strong One-Sided Confluence

Lemma 7 (Strong One-Sided Confluence)**.** *If $\sigma \longrightarrow \sigma'$ and $\sigma \longrightarrow^m \sigma''$, where $1 \leq m$, then there exist σ_c, i, j such that $\sigma' \longrightarrow^i \sigma_c$ and $\sigma'' \longrightarrow^j \sigma_c$ and $i \leq m$ and $j \leq 1$.*

Proof. We proceed by induction on m. In the base case of $m = 1$, the result is immediate from Corollary 1. For the induction step, suppose $\sigma \longrightarrow^m \sigma'' \longrightarrow \sigma'''$ and suppose the lemma holds for m. From the induction hypothesis, we have that there exist σ'_c, i', j' such that $\sigma' \longrightarrow^{i'} \sigma'_c$ and $\sigma'' \longrightarrow^{j'} \sigma'_c$ and $i' \leq m$ and $j' \leq 1$. We have two cases:

- If $j' = 0$, then $\sigma'' = \sigma'_c$. We can then choose $\sigma_c = \sigma'''$ and $i = i' + 1$ and $j = 0$.
- If $j' = 1$, then from $\sigma'' \longrightarrow \sigma'''$ and $\sigma'' \longrightarrow^{j'} \sigma'_c$ and Corollary 1, we have σ''_c and i'' and j'' such that $\sigma''' \longrightarrow^{i''} \sigma''_c$ and $\sigma'_c \longrightarrow^{j''} \sigma''_c$ and $i'' \leq 1$ and $j'' \leq 1$. So we also have $\sigma' \longrightarrow^{i'} \sigma'_c \longrightarrow^{j''} \sigma''_c$. In summary, we pick $\sigma_c = \sigma''_c$ and $i = i' + j''$ and $j = i''$, which is sufficient because $i = i' + j'' \leq m + 1$ and $j = i'' \leq 1$.

□

A.8 Strong Confluence

Lemma 8 (Strong Confluence)**.** *If $\sigma \longrightarrow^n \sigma'$ and $\sigma \longrightarrow^m \sigma''$, where $1 \leq n$ and $1 \leq m$, then there exist σ_c, i, j such that $\sigma' \longrightarrow^i \sigma_c$ and $\sigma'' \longrightarrow^j \sigma_c$ and $i \leq m$ and $j \leq n$.*

Proof. We proceed by induction on n. In the base case of $n = 1$, the result is immediate from Lemma 7. For the induction step, suppose $\sigma \longrightarrow^n \sigma' \longrightarrow \sigma'''$ and suppose the lemma holds for m. From the induction hypothesis, we have that there exist σ'_c, i', j' such that $\sigma' \longrightarrow^{i'} \sigma'_c$ and $\sigma'' \longrightarrow^{j'} \sigma'_c$ and $i' \leq m$ and $j' \leq n$. We have two cases:

- If $i' = 0$, then $\sigma' = \sigma'_c$. We can then choose $\sigma_c = \sigma'''$ and $i = 0$ and $j = j' + 1$.
- If $i' \geq 1$, then from $\sigma' \longrightarrow \sigma'''$ and $\sigma' \longrightarrow^{i'} \sigma'_c$ and Lemma 7, we have σ''_c and i'' and j'' such that $\sigma''' \longrightarrow^{i''} \sigma''_c$ and $\sigma'_c \longrightarrow^{j''} \sigma''_c$ and $i'' \leq i'$ and $j'' \leq 1$. So we also have $\sigma'' \longrightarrow^{j'} \sigma'_c \longrightarrow^{j''} \sigma''_c$. In summary, we pick $\sigma_c = \sigma''_c$ and $i = i''$ and $j = j' + j''$, which is sufficient because $i = i'' \leq i' \leq m$ and $j = j' + j'' \leq n + 1$.

□