Investigations into a Separating-typed Language

Mehmet Emre, Harlan Kringen, Zachary Sisco

1 Introduction

In this project we explore connections between linear logic and separation logic through linearly-typed λ-calculus with explicit malloc/free-style memory management. Separation logic (sometimes called bunched logic) is propositional logic extended with separating connectives ($*$ and $-\!\!*$), which describe how propositions (usually some kind of program resource like heap memory) can be separated into disjoint regions and hold true within each. Separation logic has been used to reason about memory models (malloc/free and garbage-collected languages), concurrency models, and amortized resource analysis [3].

The first work to explore the Curry-Howard correspondence between separation logic and a type system was with bunched typing [2, 4]. Bunched typing provides a language with the ability to handle both additive pairs and functions ($\wedge$ and $\rightarrow$) as well as multiplicative pairs and functions ($*$ and $-\!\!*$). There are subtle differences between corresponding types in linear and bunched types in terms of how resources are shared/used. For instance, a linear function type describes a function that uses its arguments exactly once. The corresponding function type $-\!\!*$ from separation logic denotes a function that does not share its store with its arguments. Similarly, the separating pair type $*$ describes a pair whose components do not interfere with each other.

Our project takes cues from [2, 4] but with a different approach to the semantics, typing rules, and proofs. Rather than express the semantics of the language in denotational semantics [4], we use operational semantics to make our formalisms more amenable to implementation as well as for syntactic proofs of progress and preservation. Additionally, instead of designing the language in continuation-passing style [2] we take a direct-style approach.

2 Syntax

Our syntax (Figure 1) follows the usual terms found in the λ-calculus with pairs but with the addition of terms for explicit memory management—**alloc** and **gaf** (for allocate and get-and-free, respectively). Figure 2 defines evaluation contexts and other run-time data.

$$
\begin{aligned}
e \in \mathsf{Expr} &::= n \in \mathbb{N} \mid b \in \mathbb{B} \mid x \in \mathsf{Variable} \mid \ell \in \mathsf{Location} \mid q\,\langle e_1, e_2\rangle \\
&\quad \mid q\,\lambda\, x{:}\tau.\, e \mid e_1\, e_2 \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \mid \mathbf{split}\ e_1\ \mathbf{as}\ x, y\ \mathbf{in}\ e_2 \\
&\quad \mid \mathbf{ite}\ e\ e_1\, e_2 \mid \mathbf{alloc}\ e \mid \mathbf{gaf}\ e \\
p \in \mathsf{PreType} &::= \mathbf{num} \mid \mathbf{bool} \mid \tau_1 * \tau_2 \mid \tau_1 \mathrel{-\!\!*} \tau_2 \\
\tau \in \mathsf{Type} &::= q\, p \\
q \in \mathsf{Qual} &::= \mathbf{lin} \mid \mathbf{non}
\end{aligned}
$$

Figure 1: The syntax. Note that types consist of linearity qualifiers and pre-qualified types.

$$
\begin{aligned}
E \in \mathsf{EvalCtx} &::= [\,] \mid E\ e \mid v\ E \mid q\,\langle E, e\rangle \mid q\,\langle v, E\rangle \mid \mathbf{let}\ x = E\ \mathbf{in}\ e \\
&\quad \mid \mathbf{split}\ E\ \mathbf{as}\ x, y\ \mathbf{in}\ e \mid \mathbf{ite}\ E\ e_1\, e_2 \mid \mathbf{alloc}\ E \mid \mathbf{gaf}\ E \\
v \in \mathsf{Value} &::= n \mid b \mid \ell \mid \mathbf{non}\ \lambda\, x{:}\tau.\, e \mid \mathbf{non}\ \langle v, v\rangle \\
\sigma \in \mathsf{Store} &\subseteq \mathsf{Location} \to \mathsf{Value}
\end{aligned}
$$

Figure 2: The run-time data: evaluation contexts, values, and stores.

3 Semantics

Figure 3 defines the substitution function used by our operational semantics.

We define disjoint union over stores as:

$$
(\sigma_1 \oplus \sigma_2)(\ell) = \begin{cases} \sigma_1(\ell) & \ell \in \mathrm{dom}(\sigma_1) \\ \sigma_2(\ell) & \ell \in \mathrm{dom}(\sigma_2) \end{cases}
$$

$\oplus$ is undefined if domains of σ_1 and σ_2 are not disjoint. Our definition of $\oplus$ is commutative and associative (it is similar to $\ulcorner, \urcorner$ defined in Section 4).

Figure 4 gives small-step operational semantics for our programming language. Here, we denote substituting the $[\,]$ with e in evaluation context E as $E[e]$.

Our semantics is two-fold: we interpret linear values as references to the store and use abstract machine semantics to reason about these references. The type system we introduce in the next section will ensure that the linear values hold unique references. As for nonlinear values, we use substitution semantics and we treat them as values that cannot reference the store. The E-Split and E-App rules deal with nonlinear values.

Our language also has safe manual memory management through **alloc**, **gaf**, linear pair and function constructors (which are not values), and the E-SplitLin and E-AppLin rules. The last four mechanisms operate in a way

similar to David Walker's presentation of linearly-typed λ-calculus. Our addition of **alloc** and **gaf** allows the programmer to move complex data structures to the store without re-constructing them. In our formulation, the semantics of the last four mechanisms can be seen as fusion of **alloc** and **gaf** with pair and function construction/deconstruction.

The reason we have both **alloc** and **gaf** and the existing methods is that **alloc** can allocate only clonable values[1] and **gaf** can free and retrieve only clonable values as well.

3.1 Possible extensions

We considered certain extensions to showcase applications of algorithmic linear type systems to reason about separation, and designed our type system with them in mind. However, due to time constraints we could not finish the proofs for the following extensions we had in mind:

1. Parallel processing: we designed our language and type system so that we can prove that two sub-expressions do not share the store. Then, we could evaluate them concurrently without race conditions. Proving this turned out to be tricky because the allocation order may be different leading to different stores under different executions and we needed some sort of store isomorphism.

2. Swapping/re-assigning references without consuming them. Our store typings turned out to be not expressive enough to allow swapping a location with a new value (e.g., **swap** $e_1\, e_2$ with the expected meaning). We entertained several variations of the swap expression and concluded that fusing **swap** and **let** could be a viable solution (similar to how [2] can handle assignment because they work on programs in continuation-passing style) however we could not modify our theorems in time to accommodate **swap**.

4 Typing Rules

We define the following contexts to reason about variables and locations in our type system:

$$\begin{array}{ll} \Gamma : \mathsf{Variable} \to \mathsf{Type} & \textit{Typing context} \\ \Sigma : \mathsf{Location} \to \mathsf{Type} & \textit{Store typing} \end{array}$$

Γ and Σ carry the information about types of variables and locations. We denote the empty map via $\cdot$ and we denote a single-element map

We define the operator $\ulcorner, \urcorner$ over maps as pointwise concatenation (here, $\lambda x. \ldots$ denotes a mathematical partial function, it is not syntax of our language):

[1] values that do not contain linear references inside

$$\boxed{e[x \mapsto v]}$$

$$
\begin{aligned}
n[x \mapsto v] &\stackrel{\mathsf{def}}{=} n \\
b[x \mapsto v] &\stackrel{\mathsf{def}}{=} b \\
l[x \mapsto v] &\stackrel{\mathsf{def}}{=} l \\
x[x \mapsto v] &\stackrel{\mathsf{def}}{=} v \\
y[x \mapsto v] &\stackrel{\mathsf{def}}{=} y \qquad \textit{if } x \neq y \\
(q\langle e_1, e_2\rangle)[x \mapsto v] &\stackrel{\mathsf{def}}{=} q\langle e_1[x \mapsto v], e_2[x \mapsto v]\rangle \\
(q\lambda x : \tau.e)[x \mapsto v] &\stackrel{\mathsf{def}}{=} q\lambda x : \tau.e \\
(q\lambda y : \tau.e)[x \mapsto v] &\stackrel{\mathsf{def}}{=} q\lambda x : \tau.e[x \mapsto v] \qquad \textit{if } x \neq y \\
(e_1\ e_2)[x \mapsto v] &\stackrel{\mathsf{def}}{=} (e_1[x \mapsto v]\ e_2[x \mapsto v]) \\
(\mathbf{let}\, x = e_1 \mathbf{in}\, e_2)[x \mapsto v] &\stackrel{\mathsf{def}}{=} \mathbf{let}\, x = e_1[x \mapsto v] \mathbf{in}\, e_2 \\
(\mathbf{let}\, y = e_1 \mathbf{in}\, e_2)[x \mapsto v] &\stackrel{\mathsf{def}}{=} \mathbf{let}\, y = e_1[x \mapsto v] \mathbf{in}\, e_2[x \mapsto v] \qquad \textit{if } x \neq y \\
(\mathbf{split}\, e_1 \mathbf{as}\, y, z \mathbf{\,in}\, e_2)[x \mapsto v] &\stackrel{\mathsf{def}}{=} \mathbf{split}\, e_1[x \mapsto v] \mathbf{as}\, y, z \mathbf{\,in}\, e_2 \qquad \textit{if } x = y \vee x = z \\
(\mathbf{split}\, e_1 \mathbf{as}\, y, z \mathbf{\,in}\, e_2)[x \mapsto v] &\stackrel{\mathsf{def}}{=} \mathbf{split}\, e_1[x \mapsto v] \mathbf{as}\, y, z \mathbf{\,in}\, e_2[x \mapsto v] \qquad \textit{otherwise} \\
(\mathbf{ite}\, e\, e_1\, e_2)[x \mapsto v] &\stackrel{\mathsf{def}}{=} \mathbf{ite}\, e[x \mapsto v]\, e_1[x \mapsto v]\, e_2[x \mapsto v] \\
(\mathbf{alloc}\, e)[x \mapsto v] &\stackrel{\mathsf{def}}{=} \mathbf{alloc}\, e[x \mapsto v] \\
(\mathbf{gaf}\, e)[x \mapsto v] &\stackrel{\mathsf{def}}{=} \mathbf{gaf}\, e[x \mapsto v]
\end{aligned}
$$

Figure 3: The substitution function, defined recursively over the expression.

$$\frac{(\sigma, e) \to (\sigma', e')}{(\sigma, E) \to_E (\sigma', [e'])}\text{E-Ctx}$$

$$\frac{}{(\sigma, (q\,\lambda\, x{:}\tau.\, e)\, v) \to (\sigma, e[x \mapsto v])}\text{E-App} \qquad \frac{}{(\sigma, \mathbf{let}\ x = v\ \mathbf{in}\ e) \to (\sigma, e[x \mapsto v])}\text{E-Let}$$

$$\frac{}{(\sigma, \mathbf{split}\ \langle v_1, v_2\rangle\ \mathbf{as}\ x, y\ \mathbf{in}\ e) \to (\sigma, e[x \mapsto v_1, y \mapsto v_2])}\text{E-Split}$$

$$\frac{\ell \text{ fresh}}{(\sigma, \mathbf{alloc}\ v) \to (\sigma[\ell \mapsto v], \ell)}\text{E-Alloc} \qquad \frac{v = \sigma(\ell) \quad \sigma = \sigma' \oplus (\ell \mapsto v)}{(\sigma, \mathbf{gaf}\ \ell) \to (\sigma', v)}\text{E-GetAndFree}$$

$$\frac{}{(\sigma, \mathbf{ite\ true}\ e_1\, e_2) \to (\sigma, e_1)}\text{E-IfTrue} \qquad \frac{}{(\sigma, \mathbf{ite\ false}\ e_1\, e_2) \to (\sigma, e_2)}\text{E-IfFalse}$$

$$\frac{\ell \text{ fresh}}{(\sigma, \mathbf{lin}\ \langle v_1, v_2\rangle) \to (\sigma[\ell \to \mathbf{non}\ \langle v_1, v_2\rangle], \ell)}\ \text{E-AllocPair}$$

$$\frac{\ell \text{ fresh}}{(\sigma, \mathbf{lin}\, \lambda x : \tau.e) \to (\sigma[\ell \to \mathbf{non}\, \lambda x : \tau.e], \ell)}\ \text{E-AllocFun}$$

$$\frac{\sigma = \sigma' \oplus (\ell : \mathbf{non}\ \langle v_1, v_2\rangle)}{(\sigma, \mathbf{split}\, \ell\, \mathbf{as}\, x, y\, \mathbf{in}\, e) \to (\sigma', e[x \mapsto v_1][y \mapsto v_2])}\ \text{E-SplitLin}$$

$$\frac{\sigma = \sigma' \oplus (\ell : \mathbf{non}\, \lambda x : \tau.e)}{(\sigma, \ell\, v) \to (\sigma', e[x \mapsto v])}\ \text{E-AppLin}$$

Figure 4: The operational semantics.

$$\Gamma_1, \Gamma_2 = \lambda x. \begin{cases} \Gamma_1(x) & x \in \Gamma_1 \\ \Gamma_2(x) & x \in \Gamma_2 \end{cases}$$

We define $\ulcorner, \urcorner$ for store typings Σ the same way as well. Notice that $\ulcorner, \urcorner$ over typing contexts (and store typings) forms a commutative monoid so we have the exchange structural rule automatically under this definition and we will avoid injecting it into our typing rules for our algorithmic type system.

We denote disjointness of maps Γ_1 and Γ_2 with

$$\Gamma_1 \perp \Gamma_2 \stackrel{\mathsf{def}}{=} \mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) = \emptyset.$$

We define the context difference operator $\div$ inductively–as defined in [1][2]:

$$\frac{}{\Gamma \div \cdot = \Gamma} \qquad \frac{\Gamma_1 \div \Gamma_2 = \Gamma_3 \quad x : \mathbf{lin}\, p \notin \Gamma_3}{\Gamma_1 \div \Gamma_2, x : \mathbf{lin}\, p = \Gamma_3} \qquad \frac{\Gamma_1 \div \Gamma_2, x : \mathbf{non}\, p = \Gamma_3}{\Gamma_1 \div \Gamma_2 = \Gamma_3, x : \mathbf{non}\, p}$$

This operator is undefined if both arguments contain the same linear binding (e.g. $x : \mathbf{lin}\,\mathbf{bool} \div x : \mathbf{lin}\,\mathbf{bool}$). We will exploit this fact to enforce proper use of linear variables and scoping when type checking λ and **let** expressions.

We also define the following operator $\ulcorner \hat{-} \urcorner$ to allow weakening of nonlinear bindings. It removes x from the context if and only if x maps to a linear type. Here, $\Gamma \setminus (x : \tau)$ subtracts the binding on the right-hand-side from Γ.

$$\Gamma \mathbin{\hat{-}} x = \begin{cases} \Gamma \setminus (x : \mathbf{lin}\, p) & \Gamma(x) = \mathbf{lin}\, p \\ \Gamma & \text{otherwise} \end{cases}$$

When using linear values hence converting them to non-linear values that can be duplicated, we need to be careful to prevent duplicating values containing references (e.g. pairs with linear values inside). We define the unary relation lin-free over types as below to give that guarantee:

$$\frac{}{\text{lin-free}\,\mathbf{non}\,\mathbf{bool}} \qquad \frac{}{\text{lin-free}\,\mathbf{non}\,\mathbf{num}} \qquad \frac{}{\text{lin-free}\,\mathbf{non}\,\tau_1 \multimap \tau_2} \qquad \frac{\text{lin-free}\,\tau_1 \quad \text{lin-free}\,\tau_2}{\text{lin-free}\,\mathbf{non}\,\tau_1 * \tau_2}$$

We define $\Sigma \vdash \sigma$ informally as, "σ is well-formed under store typing Σ.", i.e. for each location l in Σ, the value $\sigma(l)$ is well-typed and has the type $\Sigma(l)$.

[2]Our definition is slightly simpler because our definition of $\ulcorner, \urcorner$ gives us exchange so we don't need to bake it into this definition

$\boxed{\Gamma; \Sigma \vdash e : \tau \parallel \Gamma'}$

$$\frac{\Gamma(x) = \tau}{\Gamma; \Sigma \vdash x : \tau \parallel (\Gamma \hat{-} x)}\ \text{T-Var} \qquad \frac{}{\Gamma; \Sigma \vdash q\ b : q\ \mathbf{bool} \parallel \Gamma}\ \text{T-Bool} \qquad \frac{}{\Gamma; \Sigma \vdash q\ n : q\ \mathbf{num} \parallel \Gamma}\ \text{T-Num}$$

$$\frac{\Sigma(\ell) = \mathbf{non}\ \tau}{\Gamma; \Sigma \vdash q\ \ell : \mathbf{lin}\ \tau \parallel \Gamma}\ \text{T-Loc} \qquad \frac{\Gamma; \Sigma \vdash e : \mathbf{non}\, p \parallel \Gamma'}{\Gamma; \Sigma \vdash \mathbf{alloc}\, e : \mathbf{lin}\, p \parallel \Gamma'}\ \text{T-Alloc}$$

$$\frac{\Gamma; \Sigma \vdash e : \mathbf{lin}\, p \parallel \Gamma' \quad \text{lin-free}\, \mathbf{non}\, \tau}{\Gamma; \Sigma \vdash \mathbf{gaf}\, e : \mathbf{non}\, p \parallel \Gamma'}\ \text{T-GetAndFree}$$

$$\frac{\Gamma_1; \Sigma_1 \vdash e_1 : \tau_1 \parallel \Gamma_2 \quad \Gamma_2; \Sigma_2 \vdash e_2 : \tau_2 \parallel \Gamma_3 \quad q(\tau_1) \quad q(\tau_2)}{\Gamma_1; (\Sigma_1, \Sigma_2) \vdash q\, \langle e_1, e_2 \rangle : q\, (\tau_1 * \tau_2) \parallel \Gamma_3}\ \text{T-Pair}$$

$$\frac{\Gamma_1; \Sigma_1 \vdash e_1 : q\, (\tau_1 * \tau_2) \parallel \Gamma_2 \quad \Gamma_2, x{:}\tau_1, y{:}\tau_2; \Sigma_2 \vdash e_2 : \tau \parallel \Gamma_3}{\Gamma_1; (\Sigma_1, \Sigma_2) \vdash \mathbf{split}\, e_1\, \mathbf{as}\, x, y\, \mathbf{in}\, e_2 : \tau \parallel \Gamma_3 \div (x{:}\tau_1, y{:}\tau_2)}\ \text{T-Split}$$

$$\frac{\Gamma_1; \Sigma_1 \vdash e : \mathbf{non}\ \mathbf{bool} \parallel \Gamma_2 \quad \Gamma_2; \Sigma_2 \vdash e_1 : \tau \parallel \Gamma_3 \quad \Gamma_2; \Sigma_2 \vdash e_2 : \tau \parallel \Gamma_3}{\Gamma_1; (\Sigma_1, \Sigma_2) \vdash \mathbf{ite}\ e\ e_1\ e_2 : \tau \parallel \Gamma_3}\ \text{T-If}$$

$$\frac{\tau_2 = \mathbf{non}\ \tau \Rightarrow \Gamma_2 = \Gamma_3 \div (x{:}\tau_1) \quad \Gamma_1; \Sigma_1 \vdash e_1 : \tau_1 \parallel \Gamma_2 \quad \Gamma_2, x{:}\tau_1; \Sigma_2 \vdash e_2 : \tau_2 \parallel \Gamma_3}{\Gamma_1; (\Sigma_1, \Sigma_2) \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau_2 \parallel \Gamma_3 \div (x{:}\tau_1)}\ \text{T-Let}$$

$$\frac{q = \mathbf{non} \Rightarrow \Gamma_1 = \Gamma_2 \div (x{:}\tau_1) \quad \Gamma_1, x{:}\tau_1; \Sigma_1 \vdash e : \tau_2 \parallel \Gamma_2}{\Gamma_1; \Sigma_1 \vdash q\, \lambda\, x{:}\tau.\, e : q\, \tau_1 \multimap \tau_2 \parallel \Gamma_2 \div (x{:}\tau_1)}\ \text{T-Abs}$$

$$\frac{\Gamma_1; \Sigma_1 \vdash e_1 : q\, \tau_1 \multimap \tau_2 \parallel \Gamma_2 \quad \Gamma_2; \Sigma_2 \vdash e_2 : \tau_1 \parallel \Gamma_3}{\Gamma_1; (\Sigma_1, \Sigma_2) \vdash e_1\, e_2 : \tau_2 \parallel \Gamma_3}\ \text{T-App}$$

Figure 5: Type-checking rules with store-typing context.

Formally,

$$\frac{}{\cdot \vdash \cdot}\ \text{ST1} \qquad \frac{\cdot; \Sigma \vdash v : \mathbf{non}\, p \parallel \cdot \quad \Sigma \vdash \sigma \quad \text{lin-free}\, \mathbf{non}\, p}{\ell{:}\mathbf{lin}\, p, \Sigma \vdash \ell{:}v, \sigma}\ \text{ST2}$$

$$\frac{\cdot; \Sigma \vdash \mathbf{lin}\, \lambda x : \tau_1.e : \mathbf{lin}\, \tau_1 \multimap \tau_2 \parallel \cdot \quad \Sigma \vdash \sigma}{\ell{:}\mathbf{lin}\, \tau_1 \multimap \tau_2, \Sigma \vdash (\ell : \mathbf{non}\, \lambda x : \tau_1.e), \sigma}\ \text{ST3}$$

$$\frac{\cdot; \Sigma \vdash \mathbf{lin}\ \langle v_1, v_2 \rangle : \mathbf{lin}\, \tau_1 * \tau_2 \parallel \cdot \quad \Sigma \vdash \sigma}{\ell{:}\mathbf{lin}\, \tau_1 * \tau_2, \Sigma \vdash (\ell : \mathbf{non}\ \langle v_1, v_2 \rangle), \sigma}\ \text{ST4}$$

Here, we define store typing carefully for reference-free types (using lin-free as a side condition) and linear pairs & functions. This definition will ensure that **gaf** preserves types properly. For the values of types not free-able by **gaf** (hence must be destroyed by **split** or function application), we inspect the value in the store to construct an almost-value expression (using linear constructors) to check the types. We need this unusual deconstruction & construction mechanism because we do not allow linear pairs or functions to be values so that the only linearly-typed values are locations which will point to the store.

We define our type system in Figure 5. Notice that the type system is nondeterministic for non-empty store typings but we put the restriction that the initial (user) programs are location-free so the type system becomes algorithmic because there is only one way to split the empty store typing.

5 Proof of Soundness

Theorem 1 (Progress)**.** *If* $\cdot; \Sigma \vdash e : \tau \parallel \cdot$*, then either* e *is a value or for any* σ *such that* $\Sigma \vdash \sigma$ *then* $(\sigma, e) \to (\sigma', e')$*.*

Proof. The proof follows by structural induction on derivations of terms of e. We consider all possible cases as follows:

Case 1: $e \in \mathsf{Value}$

A value does not take a step so this case vacuously holds.

Case 2: $e = x$, where $x \in \mathsf{Variable}$

The only rule to type check e is T-Var but it does not apply under empty typing context so this case also vacuously holds.

Case 3: $e = (e_1\, e_2)$

By the induction hypothesis, e_1 is either a value or takes a step. Similarly for e_2. If e_1 or e_2 are not values, then by rule E-Ctx they take a step. If e_1 and e_2 are values, then, since e is well-typed, by T-App e_1 is a value of function type $q\,\tau_1 \multimap \tau_2$ and e_2 is of type τ_1. There are two possibilities:

(a) $q = \mathbf{non}$. Thus, $e_1 = \mathbf{non}\,\lambda x{:}\tau.\, e_3$ and $e_2 = v$, where $v \in \mathsf{Value}$. Then, by E-App (σ, e) takes a step.

(b) $q = \mathbf{lin}$. Then, $e_1 = \ell$ is a location and it is well-typed through T-Loc so $\Sigma(\ell) = \mathbf{lin}\,\tau_1 \multimap \tau_2$. Also $\Sigma \vdash \sigma$ so $\sigma(\ell) = \mathbf{non}\,\lambda x{:}\tau.e_3$ by the store typing judgments. Then, (σ, e) can take a step through E-AppLin.

Case 4: $e = \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$

By the induction hypothesis, either e_1 is a value or $(\sigma, e_1) \to (\sigma', e_1')$ by E-Ctx. If e_1 is a value, then, by E-Let, $(\sigma, e) \to (\sigma, e_2[x \mapsto e_1])$.

Case 5: $e = \textbf{split}\ e_1\ \textbf{as}\ x, y\ \textbf{in}\ e_2$

By the induction hypothesis, either e_1 is a value or takes a step. If e_1 is not a value, then, by E-Ctx, it takes a step. If e_1 is a value, then, since e is well-typed, e_1 must be of a pair type $q\,\tau_1 * \tau_2$. There are two possibilities for q (similar to Case *Case 3*: above):

(a) $q = \textbf{non}$. Hence, e can be well-typed only through T-Pair. Thus, $e = \textbf{non}\ \langle v_1, v_2 \rangle$. Then, by E-Split, (σ, e) takes a step.

(b) $q = \textbf{lin}$. Then, $e_1 = \ell$ is a location and it is well-typed through T-Loc so $\Sigma(\ell) = \textbf{lin}\,\tau_1 * \tau_2$. Also $\Sigma \vdash \sigma$ so $\sigma(\ell) = \textbf{non}\ \langle v_1, v_2 \rangle .e_3$ by the store typing judgments. Then, (σ, e) can take a step through E-SplitLin.

Case 6: $e = \textbf{alloc}\ e_1$

By the induction hypothesis, either e_1 is a value or takes a step. If e_1 is not a value, then, by E-Ctx, e_1 takes a step. Otherwise, if e_1 is a value, then, since e is well-typed, e_1 must be of some type $\textbf{non}\,\tau$. Then, by E-Alloc, $(\sigma, \textbf{alloc}\ e_1) \to (\sigma[\ell \mapsto e_1], \ell)$ for some fresh location ℓ.

Case 7: $e = \textbf{gaf}\ e_1$

By the induction hypothesis, either e_1 is a value or takes a step. If e_1 is not a value, then, by E-Ctx, e_1 takes a step. Otherwise, if e_1 is a value, then, since e is well-typed, e_1 must be of some type $\textbf{lin}\,\tau$. Furthermore, since a location is the only kind of value that can be linearly typed (by T-Loc), then e_1 must be a location. Then, we can use E-GetAndFree to show that (σ, e) takes a step.

Case 8: $e = \textbf{ite}\ e_1\ e_2\ e_3$

By the induction hypothesis, either e_1 is a value or takes a step. If e_1 is not a value, then, by E-Ctx, e_1 takes a step. Otherwise, if e_1 is a value, then, since e is well-typed, e_1 must be of type **non bool**. Thus, e_1 has value either **true** or **false**. If **true**, then, by E-IfTrue, (σ, e) takes a step. Otherwise, if **false**, then, by E-IfFalse, (σ, e) takes a step.

Case 9: $e = \textbf{lin}\ \langle e_1, e_2 \rangle$

By the induction hypothesis, e_1 is either a value or takes a step. Similarly for e_2. If e_1 or e_2 are not values, than by rule E-Ctx they take a step.

If e_1 and e_2 are values, then, since e is well-typed, by T-Pair, e_1 is type τ_1 and e_2 is of type τ_2, both $\textbf{lin}\,(\tau_1)$ and $\textbf{lin}\,(\tau_2)$ hold. Then, by E-AllocPair (σ, e) takes a step.

Case 10: $e = \textbf{lin}\,\lambda\, x{:}\tau.\, e_1$

Here, rule E-AllocFun can be applied directly so that (σ, e) takes a step.

This shows that progress can be made for all possible derivations of e, and thus, proves progress overall. □

Lemma 2 (Substitution)**.** *If* $\Gamma, x : \tau; \Sigma_1 \vdash e : \tau' \parallel \Gamma'$*, and* $\cdot; \Sigma_2 \vdash v : \tau \parallel \cdot$ *then* $\Gamma; (\Sigma_1, \Sigma_2) \vdash e[x \mapsto v] : \tau' \parallel (\Gamma' \div x)$.

Proof. by structural induction on e, following the definition of the substitution function. Most of the cases that involve multiple substitution hinge on an argument on x occurs only once or v is lin-free. The argument is fleshed out in Case 3. The cases after 3 refer to that argument.

Case 1: $e = n$ or $e = b$ or $e = l$ or $e = y$ for some $y \neq x$. Then $e[x \mapsto v] = e$ and the result holds.

Case 2: $e = x$. Then $\tau = \tau'$, $e[x \mapsto v] = e$ and $\Sigma_1 = \cdot$. We already have $\cdot; \Sigma_2 \vdash v : \tau \parallel \cdot$ so $\Gamma; (\Sigma_1, \Sigma_2) \vdash v : \tau' \parallel \Gamma$. Also, e is well-typed only through T-Var rule so $\Gamma' = (\Gamma, x : \tau) \hat{-} x$ thus $\Gamma = \Gamma' \div x$ hence we are done.

Case 3: $e = q\,\langle e_1, e_2 \rangle$. By T-Pair, we obtain the following judgments:

$$\Gamma, x{:}\tau; \Sigma_1' \vdash e_1 : \tau_1 \parallel \Gamma_1$$
$$\Gamma_1; \Sigma_2' \vdash e_2 : \tau_2 \parallel \Gamma'$$

where $\Sigma_1', \Sigma_2' = \Sigma_1$. Now, we need to case split on τ. There are 2 cases, for each case we will define Σ_3 and Σ_4 such that $\Sigma_3 = \cdot \vee \Sigma_4 = \cdot$; $\Sigma_3, \Sigma_4 = \Sigma_2$; and the applications of the induction hypothesis below work out:

(a) τ is nonlinear. Then, v is a nonlinear value and v is typed under the empty store typing (it cannot contain locations inside, as enforced by T-Abs and T-Pair rules). So, $\Sigma_3 = \cdot$ and $\Sigma_4 = \Sigma_2$ work.

(b) τ is linear. Then, the only way x is used (i.e. free) in e is either only in e_1 or e_2 or none of them: Suppose x is free in e_1, then it needs to be typed with T-Var somewhere in the proof tree so x cannot appear in the output context of e_1, which is the input context of e_2 so if it is free in e_1 it cannot be free in e_2 as well. By looking at the proof tree of well-typedness of e, we can choose:

   i. If x is free in e_1: let $\Sigma_3 = \Sigma_2$, $\Sigma_4 = \cdot$.

   ii. Otherwise, let $\Sigma_3 = \cdot$, $\Sigma_4 = \Sigma_2$. The same assignment will handle when x is not free in e as well.

From this, and the assumption that $\cdot; \Sigma_2 \vdash v : \tau \parallel \cdot$, it follows from the induction hypothesis that,

$$\Gamma; (\Sigma_1', \Sigma_3) \vdash e_1[x \mapsto v] : \tau_1 \parallel \Gamma_1$$
$$\Gamma_1; (\Sigma_2', \Sigma_4) \vdash e_2[x \mapsto v] : \tau_2 \parallel \Gamma' \div x.$$

Again, we apply the T-Pair rule to obtain,

$$\Gamma; (\Sigma_1, \Sigma_2) \vdash (q\,\langle e_1, e_2 \rangle)[x \mapsto v] : q\,(\tau_1 * \tau_2) \parallel \Gamma' \div x,$$

which is what we want to show.

Case 4: $e = q\,\lambda x : \tau.e_2$. If $x = y$, then the desired result $\Gamma; (\Sigma_1, \Sigma_2) \vdash e[x \mapsto v] : \tau' \parallel (\Gamma \div x)$ is immediate because $e[x \mapsto v] = e$. Because no substitution occurs and x is not free in e, the type of e does not change.

Case 5: $e = q\,\lambda y : \tau.e_2$ where $y \neq x$. Since $y \neq x$. $e[x \mapsto v] = (q\,\lambda y : \tau_1.e_2[x \mapsto v]$. We need to show that $e_2[x \mapsto v]$ is well-typed. By typing rule T-Abs,

$$\Gamma_1, x{:}\tau_1, y{:}\tau_2; \Sigma_1 \vdash e_2 : \tau_2 \parallel \Gamma_1', \tag{1}$$

where $\Sigma_1', \Sigma_2' = \Sigma_1$. So, $(q\,\lambda x : \tau.e_2)$ is of type $q\,(\tau_1 \multimap \tau_2) = \tau')$. By (1) and $y{:}\tau_1; \Sigma_2 \vdash v : \tau \parallel y{:}\tau_1$, it follows from the induction hypothesis that

$$\Gamma_1, y{:}\tau_1; (\Sigma_1, \Sigma_2) \vdash e_2[x \mapsto v] : \tau_2 \parallel \Gamma_1' \div x.$$

Therefore, by T-Abs,

$$\Gamma; (\Sigma_1, \Sigma_2) \vdash (q\,\lambda x : \tau.e_2[x \mapsto v]) : q(\tau_1 \multimap \tau_2) \parallel \Gamma' \div x,$$

which is what we want to show.

Case 6: $e = (e_1\ e_2)$. By T-App, we obtain:

$$\Gamma, x{:}\tau; \Sigma_1' \vdash e_1 : q\,\tau_1 \multimap \tau_2 \parallel \Gamma_1$$
$$\Gamma_1; \Sigma_2' \vdash e_2 : \tau_1 \parallel \Gamma'.$$

where $\Sigma_1 = \Sigma_1', \Sigma_2'$. By a similar argument as the pair case (*Case 3*:), we can find Σ_3 and Σ_4 such that $\Sigma_3 = \cdot \vee \Sigma_4 = \cdot$; $\Sigma_3, \Sigma_4 = \Sigma_2$; and the applications of the induction hypothesis below work out. From this, and the assumption that $\cdot; \Sigma_2 \vdash v : \tau \parallel \cdot$, it follows from the induction hypothesis that,

$$\Gamma; (\Sigma_1', \Sigma_3) \vdash e_1[x \mapsto v] : q\,\tau_1 \multimap \tau_2 \parallel \Gamma_1$$
$$\Gamma_1; (\Sigma_2', \Sigma_4) \vdash e_2[x \mapsto v] : \tau_1 \parallel \Gamma' \div x.$$

Again, we apply the T-App rule to obtain,

$$\Gamma; (\Sigma_1, \Sigma_2) \vdash (e_1\ e_2)[x \mapsto v] : \tau_2 \parallel \Gamma' \div x,$$

which is what we want to show.

Case 7: $e = \mathbf{let}\,x = e_1\,\mathbf{in}\,e_2$. Then, $e[x \mapsto v] = \mathbf{let}\,x = e_1[x \mapsto v]\,\mathbf{in}\,e_2$. By induction hypothesis, $\Gamma_2; (\Sigma_1', \Sigma_2) \vdash e_2[x \mapsto v] : \tau' \parallel (\Gamma_3 \div x)$ (where Γ_2 and Γ_3 come from well-typedness of e and T-Let rule, and $\Sigma_1', \Sigma_2' = \Sigma_1$ such that Σ_1' is the part that is used for proving well-typedness of e_1 in the proof of well-typedness of e) so, by applying T-Let rule to $\mathbf{let}\,x = e_1[x \mapsto v]\,\mathbf{in}\,e_2$, we get $\Gamma; (\Sigma_1, \Sigma_2) \vdash e : \tau' \parallel (\Gamma \div x)$ and we are done.

Case 8: $e = \mathbf{let}\, y = e_1\, \mathbf{in}\, e_2$ where $y \neq x$. Assuming our term is well-typed, we employ inversion to determine how it could have become well-typed. To this end, we employ the T-Let rule to see that

$$\frac{\cdots \quad \Gamma, x:\tau; \Sigma_1 \vdash e_1 : \tau_1 \parallel \Gamma_2 \quad \Gamma_2, y{:}\tau_1; \Sigma_2 \vdash e_2 : \tau' \parallel \Gamma_3}{\Gamma, x:\tau; (\Sigma_1, \Sigma_2) \vdash \mathbf{let}\ y = e_1\ \mathbf{in}\ e_2 : \tau' \parallel \Gamma_3 \div (y{:}\tau_1)}\ \text{T-Let}$$

where $\Gamma_3 \div (y{:}\tau_1) = \Gamma'$ and $\Sigma_1 = \Sigma_1', \Sigma_2'$. By a similar argument as the pair case (*Case 3*:), we can find Σ_3 and Σ_4 such that $\Sigma_3 = \cdot \vee \Sigma_4 = \cdot$; $\Sigma_3, \Sigma_4 = \Sigma_2$; and the applications of the induction hypothesis below work out. From this, and the assumption that $\cdot; \Sigma_2 \vdash v : \tau \parallel \cdot$, it follows from the induction hypothesis that,

$$\Gamma; \Sigma_1 \vdash e_1[x \mapsto v] : \tau_1 \parallel \Gamma_2 \div x$$
$$\Gamma_2 \div x, y{:}\tau_1; \Sigma_2 \vdash e_2[x \mapsto v] : \tau' \parallel \Gamma_3 \div x$$

We can now apply T-Let to obtain $\Gamma; \Sigma_0, \Sigma_2 \vdash e[x \mapsto v] : \tau' \parallel \Gamma_3 \div x \div y$. Note that $\Gamma_3 \div x \div y = \Gamma_3 \div y \div x = \Gamma' \div x$ and we are done.

Case 9: $e = \mathbf{split}\, e_1\, \mathbf{as}\, y, z\, \mathbf{in}\, e_2$ where $x = y \vee x = z$. The split rule works essentially in the same manner as a **let** construct in that it binds variables in the first argument to terms in the second argument. If the value to be substituted names the same value that either z or y appearing in e_2 do then we do not propagate the substitution into the body of e_2. However, we may still propagate the substitution into e_1. To do this, we employ the T-Split rule to obtain:

$$\Gamma, x{:}\tau; \Sigma_1' \vdash e_1 : q\,(\tau_1 * \tau_2) \parallel \Gamma_1'$$

Where $\Sigma_1 = \Sigma_1', \Sigma_2'$ and Σ_1' is the part of Σ_1 used for proving well-typedness of e_1 in the proof of well-typedness of e. Then, by induction hypothesis, we have

$$\Gamma, x{:}\tau; \Sigma_1', \Sigma_2 \vdash e_1[x \mapsto v] : q\,(\tau_1 * \tau_2) \parallel \Gamma_1'$$

We can apply T-Split again to show that $e[x \mapsto v] = \mathbf{split}\, e_1[x \mapsto v]\, \mathbf{as}\, y, z\, \mathbf{in}\, e_2$ is well-typed under $\Sigma_1, \Sigma_2 = \Sigma_1', \Sigma_2', \Sigma_2$:

$$\frac{\Gamma; \Sigma_1', \Sigma_2 \vdash e_1[x \mapsto v] : q\,(\tau_1 * \tau_2) \parallel \Gamma_2 \quad \Gamma_2, y{:}\tau_1, z{:}\tau_2; \Sigma_2' \vdash e_2 : \tau \parallel \Gamma' \div x}{\Gamma; (\Sigma_1', \Sigma_2', \Sigma_2) \vdash \mathbf{split}\, e_1\, \mathbf{as}\, y, z\, \mathbf{in}\, e_2 : \tau \parallel \Gamma' \div x}\ \text{T-Split}$$

Case 10: $e = \mathbf{split}\, e_1\, \mathbf{as}\, y, z\, \mathbf{in}\, e_2$ where $x \neq y \wedge x \neq z$. The proof of this case is similar to Case 7–the second **let** case. First, note that $e[x \mapsto v] = \mathbf{split}\, e_1[x \mapsto v] \mathbf{as}\, y, z\, \mathbf{in}\, e_2[x \mapsto v]$. Then, we start with well-typedness of e to obtain a way to split the store typing:

$$\frac{\Gamma, x:\tau; \Sigma_1' \vdash e_1 : q\,(\tau_1 * \tau_2) \parallel \Gamma_2 \quad \Gamma_2, x{:}\tau_1, y{:}\tau_2; \Sigma_2' \vdash e_2 : \tau \parallel \Gamma'}{\Gamma, x:\tau; (\Sigma_1', \Sigma_2') \vdash \mathbf{split}\, e_1\, \mathbf{as}\, x, y\, \mathbf{in}\, e_2 : \tau \parallel \Gamma_3 \div (x{:}\tau_1, y{:}\tau_2)}\ \text{T-Split}$$

where $\Sigma_1 = \Sigma_1', \Sigma_2'$. By a similar argument as the pair case (*Case 3*:), we can find Σ_3 and Σ_4 such that $\Sigma_3 = \cdot \vee \Sigma_4 = \cdot$; $\Sigma_3, \Sigma_4 = \Sigma_2$; and the applications of the induction hypothesis below work out. From this, and the assumption that $\cdot; \Sigma_2 \vdash v : \tau \parallel \cdot$, it follows from the induction hypothesis that,

$$\Gamma; \Sigma_1', \Sigma_4 \vdash e_1[x \mapsto v] : \tau' \parallel \Gamma_2'$$
$$\Gamma_2'; \Sigma_2', \Sigma_4 \vdash e_2[x \mapsto v] : \tau' \parallel \Gamma' \div x$$

Then, we can apply T-Split to obtain the judgment $\Gamma; \Sigma_1', \Sigma_2', \Sigma_2 \vdash e[x \mapsto v] : \tau' \parallel \Gamma' \div x$.

Case 11: $e = \mathbf{ite}\, e\, e_1\, e_2$. e is well-typed so we know

$$\frac{\Gamma_1; \Sigma_1' \vdash e : \mathbf{non\ bool} \parallel \Gamma_2 \quad \Gamma_2; \Sigma_2' \vdash e_1 : \tau \parallel \Gamma_3 \quad \Gamma_2; \Sigma_2' \vdash e_2 : \tau \parallel \Gamma'}{\Gamma_1; (\Sigma_1', \Sigma_2') \vdash \mathbf{ite}\ e\ e_1\ e_2 : \tau \parallel \Gamma'}\ \text{T-If}$$

where $\Sigma_1 = \Sigma_1', \Sigma_2'$. By a similar argument as the pair case (*Case 3*:), we can find Σ_3 and Σ_4 such that $\Sigma_3 = \cdot \vee \Sigma_4 = \cdot$; $\Sigma_3, \Sigma_4 = \Sigma_2$; and the applications of the induction hypothesis below work out. From this, and the assumption that $\cdot; \Sigma_2 \vdash v : \tau \parallel \cdot$, it follows from the induction hypothesis that,

$$\Gamma; \Sigma_1', \Sigma_3 \vdash e[x \mapsto v] : \mathbf{non\ bool} \parallel \Gamma_2$$
$$\Gamma_2; \Sigma_2', \Sigma_4 \vdash e_1[x \mapsto v] : \tau' \parallel \Gamma' \div x$$
$$\Gamma_2; \Sigma_2', \Sigma_4 \vdash e_2[x \mapsto v] : \tau' \parallel \Gamma' \div x$$

Note that $e[x \mapsto v] = \mathbf{ite}\, e[x \mapsto v]\, e_1[x \mapsto v]\, e_2[x \mapsto v]$. Then, by applying T-If rule, we obtain $\Gamma; \Sigma_1, \Sigma_2 \vdash \mathbf{ite}\, e[x \mapsto v]\, e_1[x \mapsto v]\, e_2[x \mapsto v] : \tau' \parallel \Gamma' \div x$, and we are done.

Case 12: $e = \mathbf{alloc}\, e_1$. $e[x \mapsto v] = \mathbf{alloc}\, e_1[x \mapsto v]$. By induction hypothesis, we get $\Gamma; (\Sigma_1, \Sigma_2) \vdash e_1[x \mapsto v] : \mathbf{non}\, p \parallel (\Gamma \div x)$ where $\tau' = \mathbf{lin}\, p$. By T-Alloc we get $\Gamma; (\Sigma_1, \Sigma_2) \vdash \mathbf{alloc}\, e_1[x \mapsto v] : \mathbf{lin}\, p \parallel (\Gamma \div x)$ and we are done.

Case 13: $e = \mathbf{gaf}\, e_1$. $e[x \mapsto v] = \mathbf{gaf}\, e_1[x \mapsto v]$. By induction hypothesis, we get $\Gamma; (\Sigma_1, \Sigma_2) \vdash e_1[x \mapsto v] : \mathbf{lin}\, p \parallel (\Gamma \div x)$ where $\tau' = \mathbf{non}\, p$. By T-GetAndFree we get $\Gamma; (\Sigma_1, \Sigma_2) \vdash \mathbf{gaf}\, e_1[x \mapsto v] : \mathbf{non}\, p \parallel (\Gamma \div x)$ and we are done.

□

Although the typing contexts and the store typings seem disparate, our substitution lemma shows that there is a cromulent relation between the two formed by substituting variables with values potentially referencing the store.

Lemma 3 (Contexts don't introduce variables)**.** *If x is not free in $E[e]$ then it is not free in e.*

Proof. By induction on E. None of the cases for evaluation contexts introduce variables that can be used in the holes. □

We use the lemma below implicitly to split stores along with splitting store typings.

Lemma 4. *If* $\Sigma_1, \Sigma_2 \vdash \sigma$ *then* $\exists \sigma_1, \sigma_2$ *such that* $\sigma_1 \perp \sigma_2$ *and* $\Sigma_i \vdash \sigma_i$ *for* $i \in \{ 1, 2 \}$.

Proof. The proof is rather uninteresting and by induction on Σ_1.

- $\Sigma_1 = \cdot$. Then, $\sigma_1 = [\,]$, $\sigma_2 = \sigma$, $\Sigma_2 = \Sigma$ satisfy the conditions.
- $\Sigma_1 = \ell : \tau, \Sigma_1'$. Then, by IH, there is a σ_1' such that $\Sigma_1', \Sigma_2 \vdash \sigma_1' \oplus \sigma_2$, $\sigma_1 \oplus \sigma_2 = \sigma'$, and $\sigma = [\ell \mapsto v] \oplus \sigma'$. Then, using the definition of $\Sigma \vdash \sigma$ and associativity of $\oplus$:

$$\frac{\Sigma_1', \Sigma_2 \vdash \sigma_1' \oplus \sigma_2 \quad \sigma = [\ell \mapsto v] \oplus \sigma_1' \oplus \sigma_2}{\Sigma_1, \Sigma_2 \vdash \sigma_1 \oplus \sigma_2}$$

□

We will prove a weak version of preservation theorem for evaluation contexts and expressions with no free variables.

Lemma 5 (Preservation for evaluation contexts)**.** *If* $\Sigma \vdash \sigma$, *and* $\cdot; \Sigma \vdash E[e_1] : \tau \parallel \cdot$ *then* $\exists \sigma_1, \sigma_1', \sigma_2, \Sigma_1, \Sigma_1', \Sigma_2$. $\forall \sigma_1'. \Sigma_1' \vdash \sigma_1' \wedge \Sigma_1' \perp \Sigma_2$ *such that*

$$\Sigma = \Sigma_1, \Sigma_2 \tag{2}$$
$$\Sigma_1 \vdash \sigma_1 \tag{3}$$
$$\Sigma_2 \vdash \sigma_2 \tag{4}$$
$$\cdot; \Sigma_1 \vdash e_1 : \tau_1 \parallel \cdot \tag{5}$$
$$\cdot; (\Sigma_1', \Sigma_2) \vdash E[e'] : \tau \parallel \cdot \tag{6}$$

hold for any given well-typed expression e' *such that* $\cdot; \Sigma_1' \vdash e' : \tau_1 \parallel \cdot$.

In prose, if we can split a top-level expression into an evaluation context and an inner expression, then we can also split the resources needed to type check the expression so that the resources needed (the store typing) for the evaluation context and the resources needed for the inner expression are disjoint.

Proof. By structural induction on E.

All the cases below use the induction hypothesis to get store typings handling the inner context E_1 (E_1 arises from scrunitizing E for structural induction), then use associativity of $\ulcorner, \urcorner$ to shuffle things around to obtain Σ_2 that satisfies the conditions for E. The cases are pretty similar but we produce all of them for completion nevertheless.

Case 1: $E = [\ ]$. This case trivially holds where $\Sigma_1' = \Sigma_1$. We will look at the proof tree of $\cdot; \Sigma \vdash E[e_1]$ for the rest of the cases to obtain $\Sigma_1, \Sigma_1', \Sigma_2$ that satisfy the conditions. The uses of IH in the proof trees below indicate use of the induction hypothesis.

Case 2: $E = E_1 e$.

$$\frac{\cdot; \Sigma_a \vdash E_1[e_1] : q\,\tau_1 \multimap \tau_2 \parallel \cdot \quad \cdot; \Sigma_b \vdash e : \tau_1 \parallel \cdot}{\cdot; \Sigma_a, \Sigma_b \vdash E_1[e_1]\, e : \tau_2 \parallel \cdot}\ \text{T-App}$$

So,

$$\frac{\overline{\cdot; \Sigma_a' \vdash E_1[e'] : q\,\tau_1 \multimap \tau_2 \parallel \cdot}^{\ \text{IH}} \quad \cdot; \Sigma_b \vdash e : \tau_1 \parallel \cdot}{\cdot; \Sigma_a', \Sigma_b \vdash E_1[e']\, e : \tau_2 \parallel \cdot}\ \text{T-App}$$

Then, by IH[3], there is a Σ' such that $\Sigma_a' = \Sigma_1', \Sigma'$ and $\Sigma_a = \Sigma_1, \Sigma'$. Choosing $\Sigma_2 = \Sigma_b, \Sigma'$ satisfies the conditions in our theorem statement by associativity of $\ulcorner, \urcorner$.

Case 3: $E = v E_1$. The proof for this case is similar to the case above. We apply IH to $E_1[e_1]$ and $E_1[e']$ then use associativity of $\ulcorner, \urcorner$ just like the case above.

Case 4: $E = q\,\langle E_1, e_2 \rangle$. The proof of this case and the other pair case is similar to the application case above.

$$\frac{\cdot; \Sigma_a \vdash E_1[e_1] : \tau_1 \parallel \cdot \quad \cdot; \Sigma_b \vdash e_2 : \tau_2 \parallel \cdot \quad q(\tau_1)\ \ q(\tau_2)}{\cdot; (\Sigma_a, \Sigma_b) \vdash q\ \langle E_1[e_1], e_2 \rangle : q(\tau_1 * \tau_2) \parallel \cdot}\ \text{T-Pair}$$

So, using IH on E_1, we get:

$$\frac{\overline{\cdot; \Sigma_a' \vdash E_1[e'] : \tau_1 \parallel \cdot}^{\ \text{IH}} \quad \cdot; \Sigma_b \vdash e_2 : \tau_2 \parallel \cdot \quad q(\tau_1) \quad q(\tau_2)}{\cdot; (\Sigma_a', \Sigma_b) \vdash q\ \langle E_1[e'], e_2 \rangle : q(\tau_1 * \tau_2) \parallel \cdot}\ \text{T-Pair}$$

Then, by IH, there is a Σ' such that $\Sigma_a' = \Sigma_1', \Sigma'$ and $\Sigma_a = \Sigma_1, \Sigma'$. Choosing $\Sigma_2 = \Sigma_b, \Sigma'$ satisfies our theorem statement.

Case 5: $E = q\,\langle v, E_1 \rangle$. The proof for this case is similar to the one above. We apply IH to the second part of the pair then use associativity of $\ulcorner, \urcorner$ to build Σ_1' and Σ_2.

[3] induction hypothesis

Case 6: $E = \mathbf{let}\, x = E_1\, \mathbf{in}\, e$. We have

$$\frac{\cdot; \Sigma_a \vdash E_1[e_1] : \tau_1 \parallel \cdot \quad \cdot, x{:}\tau_1; \Sigma_b \vdash e_2 : \tau_2 \parallel \cdot}{\cdot; (\Sigma_a, \Sigma_b) \vdash \mathbf{let}\ x = E_1[e_1]\ \mathbf{in}\ e_2 : \tau_2 \parallel \cdot \div (x{:}\tau_1)}\ \text{T-Let}$$

So, by applying IH on E_1, we get

$$\frac{\overline{\cdot; \Sigma_a \vdash E_1[e'] : \tau_1 \parallel \cdot}^{\text{IH}} \quad \cdot, x{:}\tau_1; \Sigma_b \vdash e_2 : \tau_2 \parallel \cdot}{\cdot; (\Sigma_a, \Sigma_b) \vdash \mathbf{let}\ x = E_1[e']\ \mathbf{in}\ e_2 : \tau_2 \parallel \cdot \div (x{:}\tau_1)}\ \text{T-Let}$$

where there is a Σ' (obtained by IH) satisfying $\Sigma_a = \Sigma_1, \Sigma'$, and $\Sigma'_a = \Sigma'_1, \Sigma'$. Then, setting $\Sigma_2 = \Sigma_b, \Sigma'$ satisfies the conditions in our theorem statement.

Case 7: $E = \mathbf{split}\, E_1\, \mathbf{as}\, x, y\, \mathbf{in}\, e_2$. By well-typedness of $E[e_1]$, we have

$$\frac{\cdot; \Sigma_a \vdash E_1[e_1] : q\,(\tau_1 * \tau_2) \parallel \cdot \quad x{:}\tau_1, y{:}\tau_2; \Sigma_b \vdash e_2 : \tau \parallel \Gamma_3}{\cdot; (\Sigma_a, \Sigma_b) \vdash \mathbf{split}\, E_1[e_1]\, \mathbf{as}\, x, y\, \mathbf{in}\, e_2 : \tau \parallel \Gamma_3 \div (x{:}\tau_1, y{:}\tau_2)}\ \text{T-Split}$$

Where $\Gamma_3 \div (x : \tau_1, y : \tau_2) = \cdot$ because $E[e_1]$ is well-typed with output context $\cdot$. By applying IH on E_1, we get:

$$\frac{\overline{\cdot; \Sigma'_a \vdash E_1[e'] : q\,(\tau_1 * \tau_2) \parallel \cdot}^{\text{IH}} \quad x{:}\tau_1, y{:}\tau_2; \Sigma_b \vdash e_2 : \tau \parallel \Gamma_3}{\cdot; (\Sigma'_a, \Sigma_b) \vdash \mathbf{split}\, E_1[e']\, \mathbf{as}\, x, y\, \mathbf{in}\, e_2 : \tau \parallel \Gamma_3 \div (x{:}\tau_1, y{:}\tau_2)}\ \text{T-Split}$$

where there is a Σ' (obtained by IH) satisfying $\Sigma_a = \Sigma_1, \Sigma'$, and $\Sigma'_a = \Sigma'_1, \Sigma'$. Then, setting $\Sigma_2 = \Sigma_b, \Sigma'$ satisfies the conditions in our theorem statement.

Case 8: $E = \mathbf{ite}\, E_1\, e_2\, e_3$. We have

$$\frac{\cdot; \Sigma_a \vdash E_1[e_1] : \mathbf{non}\,\mathbf{bool} \parallel \cdot \quad \cdot; \Sigma_b \vdash e_2 : \tau \parallel \cdot \quad \cdot; \Sigma_b \vdash e_3 : \tau \parallel \cdot}{\cdot; (\Sigma_a, \Sigma_b) \vdash \mathbf{ite}\, E[e_1]\, e_2\, e_3 : \tau \parallel \cdot}\ \text{T-If}$$

So, by applying IH on E_1, we get:

$$\frac{\overline{\cdot; \Sigma'_a \vdash E_1[e'] : \mathbf{non}\,\mathbf{bool} \parallel \cdot}^{\text{IH}} \quad \cdot; \Sigma_b \vdash e_2 : \tau \parallel \cdot \quad \cdot; \Sigma_b \vdash e_3 : \tau \parallel \cdot}{\cdot; (\Sigma'_a, \Sigma_b) \vdash \mathbf{ite}\, E[e']\, e_2\, e_3 : \tau \parallel \cdot}\ \text{T-If}$$

where there is a Σ' (obtained by IH) satisfying $\Sigma_a = \Sigma_1, \Sigma'$, and $\Sigma'_a = \Sigma'_1, \Sigma'$. Then, setting $\Sigma_2 = \Sigma_b, \Sigma'$ satisfies the conditions in our theorem statement.

Case 9: $E = \mathbf{alloc}\, E_1$. By well-typedness of $E[e_1]$ we have:

$$\frac{\Gamma; \Sigma \vdash E_1[e_1] : \mathbf{non}\, p \parallel \Gamma'}{\Gamma; \Sigma \vdash \mathbf{alloc}\, E_1[e_1] : \mathbf{lin}\, p \parallel \Gamma'}\ \text{T-Alloc}$$

By applying IH on E_1, we get:

$$\dfrac{\dfrac{}{\Gamma; \Sigma \vdash E_1[e_1] : \mathbf{non}\, p \parallel \Gamma'}\ \text{IH}}{\Gamma; \Sigma \vdash \mathbf{alloc}\, E_1[e_1] : \mathbf{lin}\, p \parallel \Gamma'}\ \text{T-Alloc}$$

where there is a Σ_2 (obtained by IH) satisfying $\Sigma = \Sigma_1, \Sigma_2$ and other conditions stated in our theorem. This Σ_2 works directly because **alloc** itself does not need additional store typing in our type system.

Case 10: $E = \mathbf{gaf}\, E_1$. By well-typedness of $E[e_1]$ we have:

$$\dfrac{\Gamma; \Sigma \vdash E_1[e_1] : \mathbf{lin}\, p \parallel \Gamma'}{\Gamma; \Sigma \vdash \mathbf{gaf}\, E_1[e_1] : \mathbf{non}\, p \parallel \Gamma'}\ \text{T-GetAndFree}$$

By applying IH on E_1, we get:

$$\dfrac{\dfrac{}{\Gamma; \Sigma \vdash E_1[e_1] : \mathbf{lin}\, p \parallel \Gamma'}\ \text{IH}}{\Gamma; \Sigma \vdash \mathbf{gaf}\, E_1[e_1] : \mathbf{non}\, p \parallel \Gamma'}\ \text{T-GetAndFree}$$

where there is a Σ_2 (obtained by IH) satisfying $\Sigma = \Sigma_1, \Sigma_2$ and other conditions stated in our theorem. This Σ_2 works directly because **gaf** itself does not need additional store typing in our type system.

□

Lemma 6 (Frame rule)**.** *If* $(\sigma_1, e) \to (\sigma_1', e')$ *and* $\sigma_2 \perp \sigma_1 \wedge \sigma_2 \perp \sigma_1'$ *then* $(\sigma_1 \oplus \sigma_2, e) \to (\sigma_1' \oplus \sigma_2, e')$.

Proof. By structural induction on e, following the evaluation rules.

Case 1: $e = v$, a value does not take a step so this case vacuously holds.

Case 2: $e = x$, variables do not take a step so this case vacuously holds.

Case 3: $e = E[e_1]$, the only rule that applies is E-Ctx. So, $(\sigma_1, e_1) \to (\sigma_1', e_1')$. By induction hypothesis, $(\sigma_1 \oplus \sigma_2, e_1) \to (\sigma_1' \oplus \sigma_2, e_1')$. By applying E-Ctx, $(\sigma_1 \oplus \sigma_2, E[e_1]) \to (\sigma_1' \oplus \sigma_2, E[e_1'])$.

Case 4: $e = v_1 v_2$, by the argument from theorem 1 for the application case, $v_1 = q\lambda x : \tau.e_2$ is a lambda abstraction (if $q = \mathbf{non}$) or a location (if $q = \mathbf{lin}$).

In the first case, the only way for e to take a step is through E-App rule. Note that the store is not used for this rule, so if $(\sigma_1, e) \to (\sigma_1, e')$ then $(\sigma_1 \oplus \sigma_2, e) \to (\sigma_1 \oplus \sigma_2, e')$.

In the $q = \mathbf{lin}$ case, the only way for e to take a step is through E-AppLin rule. So, $(\sigma_1, \ell v_2) \to (\sigma_1', e_1[v_2 \mapsto x])$ where $\sigma_1 = \sigma_1' \oplus (\ell : \mathbf{non}\, \lambda x : \tau.e_1)$. Note that $\sigma_1 \oplus \sigma_2 = (\sigma_1' \oplus \sigma_2) \oplus (\ell : \mathbf{non}\, \lambda x : \tau.e_1)$ by associativity and commutativity of $\oplus$. So, E-AppLin rule also applies under $\sigma_1 \oplus \sigma_2$: $(\sigma_1 \oplus \sigma_2, e) \to (\sigma_1' \oplus \sigma_2, e_1[v_2 \mapsto x])$.

Case 5: $e = (\mathbf{let}\, x = v\, \mathbf{in}\, e_2)$, the proof for this case is similar to the case above. The only way e can take a step is via E-Let rule, which does not use the store. So, if $(\sigma_1, e) \to (\sigma_1, e')$ then $(\sigma_1 \oplus \sigma_2, e) \to (\sigma_1 \oplus \sigma_2, e')$.

Case 6: $e = \mathbf{ite}\, v\, e_1\, e_2$, the reasoning for this case is similar to the **let** case above. The only way e can take a step is via E-IfTrue and E-IfFalse rules. Neither of these rules uses the store and they both leave the store intact so if $(\sigma_1, e) \to (\sigma_1, e')$ then $(\sigma_1 \oplus \sigma_2, e) \to (\sigma_1 \oplus \sigma_2, e')$.

Case 7: $e = \mathbf{split}\, \mathbf{non}\, \langle v_1, v_2 \rangle\, \mathbf{as}\, x, y\, \mathbf{in}\, e_2$. The only way e can take a step is through E-Split rule. This rule also does not use or alter the store so if $(\sigma_1, e) \to (\sigma_1, e')$ then $(\sigma_1 \oplus \sigma_2, e) \to (\sigma_1 \oplus \sigma_2, e')$.

Case 8: $e = \mathbf{split}\, \ell\, \mathbf{as}\, x, y\, \mathbf{in}\, e_2$. The only way e can take a step is through E-SplitLin rule. So, $(\sigma_1, \ell v2) \to (\sigma_1', e_2[v_1 \mapsto x][v_2 \mapsto y])$ where $\sigma_1 = \sigma_1' \oplus (\ell : \mathbf{non}\, \langle v_1, v_2 \rangle)$. Note that $\sigma_1 \oplus \sigma_2 = (\sigma_1' \oplus \sigma_2) \oplus (\ell : \mathbf{non}\, \langle v_1, v_2 \rangle)$ by associativity and commutativity of $\oplus$. So, E-SplitLin rule also applies under $\sigma_1 \oplus \sigma_2$: $(\sigma_1 \oplus \sigma_2, e) \to (\sigma_1' \oplus \sigma_2, e_2[v_1 \mapsto x][v_2 \mapsto y])$.

Case 9: $e = \mathbf{alloc}\, v$. The only way e can take a step is via E-Alloc rule. So, $\sigma_1' = \sigma_1[\ell \mapsto v]$ and $(\sigma_1 \oplus \sigma_2, \mathbf{alloc}\, v) \to (\sigma_1[\ell \mapsto v] \oplus \sigma_2, \ell)$ also holds because $\sigma_2 \perp \sigma_1[\ell \mapsto v]$ is a premise in our theorem.

Case 10: $e = \mathbf{gaf}\, \ell$. The only way e can take a step is via E-GetAndFree rule. So, $\sigma_1 = \sigma_1'[\ell \mapsto v]$ and $(\sigma_1 \oplus \sigma_2, \mathbf{gaf}\, v) \to (\sigma_1' \oplus \sigma_2, \sigma_1(\ell))$ also holds because $\sigma_1 \perp \sigma_1$.

□

Theorem 7 (Preservation)**.** *If*

$$\begin{gathered} \cdot; \Sigma \vdash e : \tau \parallel \cdot \\ \Sigma \vdash \sigma \\ (\sigma, e) \to (\sigma', e') \end{gathered}$$

Then there exists a store typing Σ' such that $\Sigma' \vdash \sigma$ and $\cdot; \Sigma \vdash e' : \tau \parallel \cdot$.

Proof. The proof follows by structural induction on e, grouped by different cases $e \to e'$ applies. We consider all possible cases as follows:

Case 1: $e = v$, a value does not take a step so this case vacuously holds.

Case 2: $e = x$, the only rule to type check e is T-Var but it does not apply under empty typing context so this case also vacuously holds.

Case 3: $e = E[e_1]$ for some evaluation context E and expression e_1. The only applicable rule for $E[e_1]$ taking a step is by using E-Ctx rule: $(\sigma, E[e_1]) \to (\sigma', E[e_1'])$.

so $\forall \sigma_1$ such that $(\sigma_1, e_1) \to (\sigma_1', e_1')$ the following holds: $(\sigma, E[e_1]) \to (\sigma_1' \oplus \sigma_2, E[e_1'])$ where σ_2 denotes the rest of the store (the solution to $\sigma = \sigma_1 \oplus \sigma_2$) by frame rule and E-Ctx.[4]

By induction hypothesis, we know that there exists a Σ_1' such that $\cdot; \Sigma_1' \vdash e_1' : \tau_1 \parallel \cdot$ and $\Sigma_1' \vdash \sigma_1'$. By lemma 5, we know that

$$\begin{gathered} \Sigma = \Sigma_1, \Sigma_2 \\ \Sigma_1 \perp \Sigma_2 \\ \cdot; \Sigma_1 \vdash e_1 : \tau_1 \parallel \cdot \\ \cdot; (\Sigma_2, \Sigma_1') \vdash E[e'] : \tau \parallel \cdot \\ \Sigma_1 \vdash \sigma_1 \\ \Sigma_2 \vdash \sigma_2 \end{gathered}$$

So, $\cdot; (\Sigma_1', \Sigma_2) \vdash E[e'] : \tau \parallel \cdot$ and $(\Sigma_1', \Sigma_2) \vdash \sigma_1' \oplus \sigma_2$.

Case 4: $e = v_1 v_2$. e is well-typed and the only rule that applies is T-App. So, the type of v_1 is $q\,\tau_1 \multimap \tau$. By well-typedness of e, we know:

$$\frac{\cdot; \Sigma_1 \vdash v_1 : q\,\tau_1 \multimap \tau \parallel \cdot \quad \cdot; \Sigma_2 \vdash v_2 : \tau_1 \parallel \cdot}{\cdot; \Sigma_1, \Sigma_2 \vdash v_1\, v_2 : \tau \parallel \cdot}\ \text{T-App}$$

There are two possibilities:

(a) If $q = \mathbf{non}$ then v_1 has to be a function so $v_1 = q\lambda x : \tau_1.e_1$. So, $x : \tau_1; \Sigma_1 \vdash e_1 : \tau \parallel \cdot$ as demonstrated by the only valid proof tree for well-typedness of e below:

$$\frac{\cdot = \Gamma_2 \div x : \tau_1 \quad x : \tau_1; \Sigma_1 \vdash e_1 : \tau \parallel x : \Gamma_2}{\cdot; \Sigma_1 \vdash \mathbf{non}\,\lambda x : \tau_1.e_1 : \mathbf{non}\,\tau_1 \multimap \tau \parallel \cdot}\ \text{T-Abs}$$

So, e can take a step only by applying E-App rule: $(\sigma, (q\lambda x : t.e_1)v_2) \to (\sigma, e_1[x \mapsto v_2])$. So, for this case, $\sigma' = \sigma$ and $e' = e_1[x \mapsto v_2]$. $\Sigma_1 \vdash \sigma'$ and by substitution lemma, $\cdot; \Sigma_1 \vdash e_1[x \mapsto v] \parallel \cdot$ so this case holds.

(b) If $q = \mathbf{lin}$ then v has to be a location. Let $v = \ell$. Either of the following cases hold where $\sigma = \sigma', \ell : v$ and $\Sigma_1 = \ell : \mathbf{lin}\,\tau_1 \multimap \tau, \Sigma'$ because v is well-typed via T-Loc.

$$\frac{\cdot; \Sigma \vdash v : \mathbf{non}\,\tau_1 \multimap \tau \parallel \cdot \quad \Sigma \vdash \sigma \quad \text{lin-free}\,\mathbf{non}\,\tau_1 \multimap \tau}{\ell{:}\mathbf{lin}\,\tau_1 \multimap \tau, \Sigma_1 \vdash \ell{:}v, \sigma}\ \text{st2}$$

$$\frac{\cdot; \Sigma' \vdash \mathbf{lin}\,\lambda x : \tau_1.e_1 : \mathbf{lin}\,\tau_1 \multimap \tau \parallel \cdot \quad \Sigma' \vdash \sigma'}{\ell{:}\mathbf{lin}\,\tau_1 \multimap \tau, \Sigma_1' \vdash (\ell : \mathbf{non}\,\lambda x : \tau_1.e_1), \sigma'}\ \text{st3}$$

[4]Here, we are focusing on the part of the store needed for e_1 so we can use the induction hypothesis then extend the store.

If ST2 holds then $\sigma(\ell)$ is a lambda abstraction because it is a well-typed value of type $\mathbf{non}\,\tau_1 \multimap^* \tau$. So, in both cases, $\sigma(\ell) = \mathbf{non}\,\lambda x : \tau_1.e_1$ and $x : \tau_1; \Sigma_1 \vdash e_1 : \tau \parallel \Gamma_3$ where $\Gamma_3 \div x : \tau_1 = \cdot$ because the function in the store is well-typed as enforced by the first premise of each store typing rule above. So, e can only take a step via E-AppLin rule: $(\sigma, e) \to (\sigma', e_1[x \mapsto v_2])$. By our store typing rules above, we know that $\Sigma_1', \Sigma_2 \vdash \sigma_1', \sigma_2$ where Σ_1', σ_1' come from the store typing rules above and σ_2 is the part of σ satisfying $\Sigma_2 \vdash \sigma_2$. By substitution lemma, $\cdot; \Sigma_1', \Sigma_2 \vdash e_1[x \mapsto v_2] : \tau \parallel \cdot$ so we are done.

Case 5: $e = (\mathbf{let}\,x = v\,\mathbf{in}\,e_2)$. The only applicable typing rule is T-Let. So, we know the following:

$$\begin{aligned} \mathbf{non}\,(\tau_2) \Rightarrow \cdot = \Gamma_3 \div (x{:}\tau_1) \\ \cdot; \Sigma_1 \vdash v : \tau_1 \parallel \Gamma_2 \\ x{:}\tau_1; \Sigma_2 \vdash e_2 : \tau_2 \parallel \Gamma_3 \end{aligned}$$

where $\Sigma = \Sigma_1, \Sigma_2$. Also, the only way for e to take a step is via E-Let rule: $(\sigma, \mathbf{let}\,x = v\,\mathbf{in}\,e_2) \to (\sigma, e_2[x \mapsto v])$. So, we can apply the substitution lemma to obtain $\cdot; \Sigma \vdash e_2[x \mapsto v] : \tau \parallel \cdot$ and we are done.

Case 6: $e = \mathbf{split}\,\mathbf{non}\,\langle v_1, v_2\rangle\,\mathbf{as}\,x, y\,\mathbf{in}\,e_2$. This case is similar to the **let** case above. The only applicable typing rule is T-Split so we get

$$\begin{aligned} x{:}\tau_1, y{:}\tau_2; \Sigma_3 \vdash e_2 : \tau \parallel \Gamma' \\ \Gamma' \div (x{:}\tau_1, y{:}\tau_2) = \cdot \end{aligned}$$

where our store typing is split into three parts[5] $\Sigma = \Sigma_2, \Sigma_2, \Sigma_3$ such that $\cdot; \Sigma_2 \vdash v_2 : \tau_2 \parallel \cdot$ and $\cdot; \Sigma_2 \vdash v_2 : \tau_2 \parallel \cdot$ so, we can apply the substitution lemma to obtain

$$y{:}\tau_2; (\Sigma_1, \Sigma_3) \vdash e_2[x \mapsto v_1] : \tau \parallel \Gamma''\Gamma' \div (y{:}\tau_2) = \cdot$$

we can apply the substitution lemma again to obtain

$$\cdot; (\Sigma_1, \Sigma_2, \Sigma_3) \vdash e_2[x \mapsto v_1][y \mapsto v_2] : \tau \parallel \cdot$$

Notice that the only way for e to take a step is via E-Split: $(\sigma, e) \to (\sigma, e_2[x \mapsto v_1][y \mapsto v_2])$, and we just showed that $\cdot; \Sigma \vdash e_2[x \mapsto v_1][y \mapsto v_2] : \tau \parallel \cdot$ so we are done with this case.

[5] using the typing rules T-Pair and T-Split in the proof of well-typedness of e

Case 7: $e = \mathbf{split}\,\ell\,\mathbf{as}\,x, y\,\mathbf{in}\,e_2$. Let $\Sigma = \Sigma_1, \Sigma_2$ and $\sigma = \sigma_1 \oplus \sigma_2$ satisfying $\Sigma_i \vdash \sigma_i$ for $i \in \{ 1, 2 \}$. We will get Σ_1, Σ_2 from the well-typing proof of e:

$$\frac{\cdot; \Sigma_1 \vdash \ell : \mathbf{lin}\ (\tau_1 * \tau_2) \parallel \cdot \quad \cdot, x{:}\tau_1, y{:}\tau_2; \Sigma_2 \vdash e_2 : \tau \parallel \cdot}{\cdot; (\Sigma_1, \Sigma_2) \vdash \mathbf{split}\,\ell\,\mathbf{as}\,x, y\,\mathbf{in}\,e_2 : \tau \parallel \cdot \div (x{:}\tau_1, y{:}\tau_2)}\ \text{T-Split}$$

Note that $\Gamma_3 \div (x : \tau_1, y : \tau_2) = \cdot$ because the output context of well-typedness of e is empty $(*)$.

Also note that, ℓ is well-typed under empty context so we know that $\Sigma_1(\ell) = \mathbf{lin}\,\tau_1 * \tau_2$ for some τ_1, τ_2. By $\Sigma_1 \vdash \sigma_1$, we know that either of the following judgments hold:

$$\frac{\cdot; \Sigma_1' \vdash v : \mathbf{non}\,\tau_1 * \tau_2 \parallel \cdot \quad \Sigma_1' \vdash \sigma_1' \quad \text{lin-free}\,\mathbf{non}\,\tau_1 * \tau_2}{\ell{:}\mathbf{lin}\,\tau_1 * \tau_2, \Sigma_1' \vdash \ell{:}v, \sigma_1'}\ \text{ST2}$$

$$\frac{\cdot; \Sigma_1' \vdash \mathbf{lin}\ \langle v_1, v_2 \rangle : \mathbf{lin}\,\tau_1 * \tau_2 \parallel \cdot \quad \Sigma_1' \vdash \sigma_1'}{\ell{:}\mathbf{lin}\,\tau_1 * \tau_2, \Sigma_1' \vdash (\ell : \mathbf{non}\ \langle v_1, v_2 \rangle), \sigma_1'}\ \text{ST4}$$

In the first case (where ST2 holds), $\sigma_1(\ell) = v$ is a value of a nonlinear pair type so it has to be a pair $\sigma_1(\ell) = v = \mathbf{non}\ \langle v_1, v_2 \rangle$. In both cases, we can apply only E-SplitLin rule to step e. So, $(\sigma_1 \oplus \sigma_2, e) \to (\sigma_1' \oplus \sigma_2, e[x \mapsto v_1][y \mapsto v_2])$. By applying the substitution lemma twice and using the observation $(*)$ above, we obtain $\cdot; \Sigma_1', \Sigma_2 \vdash e[x \mapsto v_1][y \mapsto v_2] : \tau \parallel \cdot$. Also, $\Sigma_1' \vdash \sigma_1'$ by the store typing rules above and we have $\Sigma_2 \vdash \sigma_2$ so the store typing Σ_1', Σ_2 also satisfies $\Sigma_1', \Sigma_2 \vdash \sigma_1 \oplus \sigma_2$ and we are done.

Case 8: $e = \mathbf{alloc}\,v$. e is well-typed and the only typing rule that applies is T-Alloc so the type of e is $\mathbf{lin}\,\tau$ and $\cdot; \Sigma \vdash v : \mathbf{non}\,\tau \parallel \cdot$ holds. The only way e can take a step is via E-Alloc rule so $(\sigma, \mathbf{alloc}\,v) \to (\sigma[\ell \mapsto v], \ell)$. Note that $\Sigma, \ell : \mathbf{non}\,\tau \vdash \sigma[\ell \mapsto v]$ hence $\cdot; \Sigma, \ell : \mathbf{non}\,\tau \vdash \ell : \mathbf{lin}\,\tau \parallel \cdot$.

Case 9: $e = \mathbf{gaf}\,v$. e is well-typed and the only typing rule that applies is T-GetAndFree so the type of e is $\mathbf{non}\,p$ and $\cdot; \Sigma \vdash v : \mathbf{lin}\,p \parallel \cdot$ holds. Since v is a value that has a linear type, it has to be a location: $v = \ell$ for some $\ell \in \mathrm{dom}(\sigma)$. The only way e can take a step is via E-GetAndFree rule: $(\sigma, \mathbf{gaf}\,\ell) \to (\sigma', \sigma(\ell))$ where σ' is σ without the binding $\ell \mapsto \sigma(\ell)$. We need to find a Σ' such that $\Sigma' \vdash \sigma'$ and $\cdot; \Sigma' \vdash \sigma(\ell) \parallel \cdot$.

Note that $\Sigma \vdash \sigma$ and σ is non-empty. Because stores and store contexts are unordered, all proofs of $\Sigma \vdash \sigma$ commute (we can re-order the heaps to change the proof tree). So, the premises of the following judgment hold and are exactly what we need (where Σ' is Σ without the binding $\ell : \tau$, the last premise holds because it is a premise of well-typedness of $\mathbf{gaf}\,v$):

$$\frac{\cdot;\Sigma \vdash v : \mathbf{non}\, p \parallel \cdot \quad \Sigma \vdash \sigma \quad \text{lin-free}\, \mathbf{non}\, p}{\ell{:}\mathbf{lin}\, p, \Sigma \vdash \ell{:}v, \sigma}$$

Case 10: $e = \mathbf{ite}\, v\, e_1\, e_2$. The only way to type check e is via T-If so $\cdot;\cdot \vdash v : \mathbf{non}\, \mathbf{bool} \parallel \cdot$. So, we know that

$$\cdot;\Sigma \vdash e_1 : \tau \parallel \cdot$$
$$\cdot;\Sigma \vdash e_2 : \tau \parallel \cdot$$

The only ways e can take a step are via E-IfFalse and E-IfTrue, depending on v. So the possible next states are $(\sigma, e) - > (\sigma, e_1)$ or $(\sigma, e) - > (\sigma, e_2)$ and we just showed that the type of the expression is preserved in both cases.

Case 11: $e = \mathbf{lin}\, \lambda x : \tau_1.e_1$. e is well-typed so $\cdot;\Sigma \vdash e : \mathbf{lin}\, \tau_1 \multimap \tau_2 \parallel \cdot$. The only way e can take a step is via E-AllocFun: $((\sigma, \mathbf{lin}\, \lambda x : \tau.e) \rightarrow (\sigma[\ell \mapsto \mathbf{non}\, \lambda x : \tau.e], \ell)$ where ℓ is a fresh location. Let $\Sigma' = \Sigma, \ell : \mathbf{lin}\, \tau_1 \multimap \tau_2$. Then, the following judgment holds (the premises are the assumptions in our theorem).

$$\frac{\cdot;\Sigma \vdash \mathbf{lin}\, \lambda x : \tau_1.e : \mathbf{lin}\, \tau_1 \multimap \tau_2 \parallel \cdot \quad \Sigma \vdash \sigma}{\ell{:}\mathbf{lin}\, \tau_1 \multimap \tau_2, \Sigma \vdash (\ell : \mathbf{non}\, \lambda x : \tau_1.e), \sigma}$$

So, the type of our expression and the store is preserved under the store typing $\Sigma' = \Sigma, \ell : \mathbf{lin}\, \tau_1 \multimap \tau_2$ in this case.

Case 12: $e = \mathbf{lin}\, \langle v_1, v_2 \rangle$. The proof of this case is similar to the case above. e is well-typed so $\cdot;\Sigma \vdash e : \mathbf{lin}\, \tau_1 * \tau_2 \parallel \cdot$. The only way e can take a step is via E-AllocPair: $((\sigma, \mathbf{lin}\, \langle v_1, v_2 \rangle) \rightarrow (\sigma[\ell \mapsto \mathbf{non}\, \langle v_1, v_2 \rangle], \ell)$ where ℓ is a fresh location. Let $\Sigma' = \Sigma, \ell : \mathbf{lin}\, \tau_1 * \tau_2$. Then, the following judgment holds (the premises are the assumptions in our theorem).

$$\frac{\cdot;\Sigma \vdash \mathbf{lin}\, \langle v_1, v_2 \rangle : \mathbf{lin}\, \tau_1 * \tau_2 \parallel \cdot \quad \Sigma \vdash \sigma}{\ell{:}\mathbf{lin}\, \tau_1 * \tau_2, \Sigma \vdash (\ell : \mathbf{non}\, \langle v_1, v_2 \rangle), \sigma}\ \text{ST4}$$

□

References

[1] D. Aspinall and M. Hofmann. Dependent Types. In *Advanced Topics in Types and Programming Languages*. The MIT Press, 12 2004.

[2] J. Berdine and P. W. O'Hearn. Strong update, disposal, and encapsulation in bunched typing. In S. D. Brookes and M. W. Mislove, editors, *Proceedings of the 22nd Annual Conference on Mathematical Foundations of Programming Semantics, MFPS 2006, Genova, Italy, May 23-27, 2006*, volume 158 of *Electronic Notes in Theoretical Computer Science*, pages 81–98. Elsevier, 2006.

[3] Q. Cao, S. Cuellar, and A. W. Appel. Bringing order to the separation logic jungle. In *Asian Symposium on Programming Languages and Systems*, pages 190–211. Springer, 2017.

[4] P. W. O'Hearn. On bunched typing. *J. Funct. Program.*, 13(4):747–796, 2003.