A. PROOFS

Lemma 2.1. Given a type-checked program $g \vdash g : \omega$, an expression $e$ that type-checks $\Delta; \Pi_1 \vdash e : \tau \rightarrow \Delta'; \Pi'_1$, and an environment $\Delta''; \Pi_2$, then $e$ also type-checks with a larger set of permissions $(\Delta \cup \Delta''; \Pi_1, \Pi_2 \vdash e : \tau \rightarrow \Delta' \cup \Delta''; \Pi'_1, \Pi'_2)$ in which the unused permissions are not changed.

Proof. As $e$ must have been typed using one of the rules, it suffices to show that, for each rule, if that rule applies to $e$ using $\Delta; \Pi_1$ (and returning $\Delta'; \Pi'_1$), it also applies to $e$ using $\Delta \cup \Delta''; \Pi_1, \Pi_2$ (and returning $\Delta' \cup \Delta''; \Pi'_1, \Pi'_2$). Additionally, we inductively assume the lemma applies to subexpressions of $e$. We now look at each rule as a separate case:

Unit, Num, True, False, Address

As these rules neither depend upon nor alter the environment, they type under any environment and the lemma is trivially true.

Plus $e_1 * e_2$

By hypothesis, we know that
$$\Delta; \Pi_1 \vdash \omega \ e_1 + e_2 \ : \ \text{int} \rightarrow \Delta'; \Pi'_1$$
. To have typed this using the Plus rule, the following must be true:
$$\Delta; \Pi_1 \vdash \omega \ e_1 : \text{int} \rightarrow \Delta_1; \Pi''_1 \vdash \omega \ e_2 : \text{int} \rightarrow \Delta'; \Pi'_1$$
Inductively,
$$\Delta \cup \Delta''; \Pi_1, \Pi_2 \vdash \omega \ e_1 : \text{int} \rightarrow \Delta_1 \cup \Delta''; \Pi''_1, \Pi_2$$
$$\Delta_1 \cup \Delta''; \Pi''_1, \Pi_2 \vdash \omega \ e_2 : \text{int} \rightarrow \Delta_1' \cup \Delta''; \Pi'_1, \Pi_2$$
So, by applying the rule Plus, we arrive at
$$\Delta \cup \Delta''; \Pi_1, \Pi_2 \vdash \omega \ e_1 + e_2 : \text{int} \rightarrow \Delta_1' \cup \Delta''; \Pi'_1, \Pi_2$$

Equal $e_1 = e_2$

The types are different than in Plus, but the environment is treated in the same manner. Thus the proof is essentially the same.

Read $e \cdot f$

By hypothesis:
$$\Delta; \Pi_1 \vdash \omega \ e \cdot f : \tau \rightarrow \Delta'; \Pi'_1$$
As we assume we applied the Read rule, the following must be true:
$$\Delta; \Pi_1 \vdash \omega \ e : \text{ptr}(l) \rightarrow \Delta' \cup \Delta''; \Pi''_1$$
$$\Pi_1 = l.f : \tau \setminus \{\ldots\} \cdot \Pi$$
$$\tau \sim \tau$$

By induction,
$$\Delta \cup \Delta''; \Pi_1, \Pi_2 \vdash \omega \ e \cdot f : \tau \rightarrow \Delta' \cup \Delta''; \Pi''_1, \Pi_2$$.

From the second fact and from the definition of the comma operator, we can conclude
$$\Pi_1, \Pi_2 = l.f : \tau \setminus \{\ldots\} \cdot \Pi, \Pi_2.$$ This gives us sufficient information to apply the Read rule with the larger environment:
$$\Delta \cup \Delta''; \Pi_1, \Pi_2 \vdash \omega \ e \cdot f : \tau \rightarrow \Delta' \cup \Delta''; \Pi''_1, \Pi_2$$

New new $\{f_i : 1 \leq i \leq n\}$

The only requirement for this rule to have been applied is for $r$ to be fresh (given context $\Delta$). We can safely assume that $r$ is still fresh given context $\Delta \cup \Delta''$ because we can type the expression in the original environment with such a $r$. Thus, we can successfully apply the New rule despite the altered environment.

Write $c_1 . f : = e_2$

The typing prerequisites follow inductively as with Plus, one may add extra permissions to both sides of an equation here as in Read and the storage compatibility requirements are unaffected by the permissions. Thus we can prove all that is necessary to apply Write in the new environment.

Seq $e; e'$

Again, this is essentially the same as Plus.

If if $c_0$ then $e_1$ else $e_2$

If this rule were successfully applied, the following must be true:
$$\Delta; \Pi_1 \vdash \omega \ c_0 : \text{bool} \rightarrow \Delta_0; \Pi''_1$$
$$\Delta_0; \Pi''_1 \vdash \omega \ e_1 : \text{unit} \rightarrow \Delta_1; \Pi''_1$$
$$\Delta_0; \Pi''_1 \vdash \omega \ e_2 : \text{unit} \rightarrow \Delta_2; \Pi''_1$$
$$\Delta; \Pi''_1 = \Delta_1; \Pi''_1 \cup \Delta_2; \Pi''_1$$

By induction,
$$\Delta \cup \Delta''; \Pi_1, \Pi_2 \vdash \omega \ c_0 : \text{bool} \rightarrow \Delta_0 \cup \Delta''; \Pi''_1, \Pi_2$$
$$\Delta_0 \cup \Delta''; \Pi''_1, \Pi_2 \vdash \omega \ e_1 : \text{unit} \rightarrow \Delta_1 \cup \Delta''; \Pi''_1, \Pi_2$$
$$\Delta_0 \cup \Delta''; \Pi''_1, \Pi_2 \vdash \omega \ e_2 : \text{unit} \rightarrow \Delta_2 \cup \Delta''; \Pi''_1, \Pi_2$$

Because $\Delta'', \Pi_2$ is an environment, if $r$ appears in $\Pi_2$ then $r \not\in \Delta''$. From the definition of the $\lor$ operator, we can also conclude that $\Delta_1 \cup \Delta''; \Pi''_1 = \sigma(\Delta \cup \Delta'')$; $\sigma, \Pi''_1$, and therefore $\Delta_1 \cup \Delta''; \Pi''_1, \Pi_2 = \sigma(\Delta \cup \Delta'')$; $\sigma, \Pi''_1, \Pi_2$. But because every $r$ that appears in $\Pi_2$ must also be in $\Delta'' \subseteq (\Delta_0 \cup \Delta'') \cap (\Delta_1 \cup \Delta'')$, $\sigma, \Pi = \Pi_2$. Therefore, $\Delta_1 \cup \Delta''; \Pi''_1, \Pi_2 = \sigma(\Delta_0 \cup \Delta_1)$; $\sigma, \Pi''_1, \Pi_2$. We further assume the fresh variables in $\Delta'$ could be chosen so as not to intersect with $\Delta''$. If not, simply rename to new fresh variables in the original typing proof. Therefore the variables in $\Delta' = ((\Delta_1 \cap \Delta_2) \cup \Delta'')$ are also fresh, and we may apply the definition of $\lor$ to determine that
$$\Delta' \cup \Delta''; \Pi''_1, \Pi_2 = \Delta_1 \cup \Delta''; \Pi''_1, \Pi_2 \cup \Delta_2 \cup \Delta''; \Pi''_1, \Pi_2$$

This, combined with the earlier inductive results is sufficient to apply the If rule and type check the expression in the new environment.

IfEqual if $e = e'$ then $e_1$ else $e_2$

By hypothesis:
$$\Delta; \Pi_1 \vdash \omega \ if \ e = e' \ then \ e_1 \ else \ e_2 : \text{unit} \rightarrow \Delta'; \Pi''_1$$
As we are assuming the IfThen rule was applied, we know the following must have been true:

\[ \Delta; \Pi_1 \vdash e : \text{ptr}(l) \vdash \Delta_0; \Pi_1 \]

\[ \Delta_0; \Pi_1 \vdash e' : \text{ptr}(l') \vdash \Delta_2; \Pi_2 \]

\[ \Delta_2; (\Pi_{12}, l = l') \vdash \Delta_2; \Pi_{12} \]

\[ \Delta_2; (\Pi_{12}, l \neq l') \vdash \Delta_2; \Pi_{12} \]

By induction:

\[ \Delta \cup \Delta''; \Pi_1, \Pi_2 \vdash e : \text{ptr}(l) \vdash \Delta_0 \cup \Delta''; \Pi_{11}, \Pi_2 \]

\[ \Delta_0 \cup \Delta''; \Pi_{12}, \Pi_2 \vdash e' : \text{ptr}(l') \vdash \Delta_2 \cup \Delta''; \Pi_{12}, \Pi_2 \]

\[ \Delta_2 \cup \Delta''; \Pi_{12}, \Pi_2 \]

Further, we can reiterate the argument from the If case to conclude that

\[ \Delta' \cup \Delta''; \Pi_1', \Pi_2 = \Delta_1 \cup \Delta''; \Pi_{10}, \Pi_2 \cup \Delta_2 \cup \Delta''; \Pi_{1b}, \Pi_2 \]

Therefore, we can apply the IfThen rule to type the expression in the new environment.

**IfTrue if true then e₁ else e₂**

Trivially true by induction on e₁.

**IfFalse if false then e₁ else e₂**

Trivially true by induction on e₂.

**Call call p**

If we assume the Call rule had been applied to type our expression, then its prerequisites must be true. As before, we may assume that \( \Delta' \) is fresh with regard to \( \Delta \cup \Delta'' \) as well as \( \Delta \). Thus the only prerequisite for this rule that requires the environment (directly) is

\[ \sigma_1 : \Delta_1 \rightarrow \Delta \]

However, it is clear from the definition of substitutions that one may increase the range of \( \sigma_1 \) without altering its definition. In particular, one may define

\[ \sigma'_1 : \Delta_1 \rightarrow \Delta \cup \Delta'' \]

where \( \sigma'_1 \) and \( \sigma_1 \) are identical maps. By including the \( \Pi_2 \) from the lemma in the \( \Pi_1 \) from the Call rule, we may directly apply the Call rule in the environment \( \Delta \cup \Delta''; \Pi_1, \Pi_2 \) and get back the environment \( \Delta \cup \Delta''; \Pi_1', \Pi_2 \) (where \( \Pi'_1 \) in the lemma equals \( \sigma'_1 \Pi_2, \Pi_3 \) in the Call rule).

**Nest nest e₁ f in e₂, f₂**

By hypothesis:

\[ \Delta; \Pi_1 \vdash \text{nest } e₁ f \text{ in } e₂, f₂ \vdash \Delta' ; (k') : \]

\[ \tau' \{ k' \in \{ k' \} : k' \neq k \land k \neq \tau \}, \Pi'' \]

Therefore, the following must have been true to apply the Nest rule:

\[ \Delta; \Pi_1 \vdash e : \text{ptr}(l) \vdash \Delta_0; \Pi_0 \]

\[ \Delta_0; \Pi_0 \vdash e' : \text{ptr}(l') \vdash \Delta'; \Pi'_1 \]

\[ k = l, f \]

\[ k' = l', f' \]

\[ \Pi'_1 = (k : k \neq \tau_1 \land \ldots \land k \neq \tau_n), k' : \]

\[ \tau' \{ k \in \{ k \} : k \neq \tau_1 \land \ldots \land k \neq \tau_n \}, \Pi'' \]

By induction,

\[ \Delta \cup \Delta''; \Pi_1, \Pi_2 \vdash e : \text{ptr}(l) \vdash \Delta_0 \cup \Delta''; \Pi_0, \Pi_2 \]

\[ \Delta_0 \cup \Delta''; \Pi_0, \Pi_2 \vdash e' : \text{ptr}(l') \vdash \Delta' \cup \Delta''; \Pi'_1, \Pi_2 \]

And, from the definition of the comma operator, we can deduce that,

\[ \Pi'_1, \Pi_2 = (k : k \neq \tau_1 \land \ldots \land k \neq \tau_n), k' : \]

\[ \tau' \{ k \in \{ k \} : k \neq \tau_1 \land \ldots \land k \neq \tau_n \}, \Pi'' \]

This is sufficient information to apply the Nest rule and type the expression in the new environment.

\[ \square \]

We also have a substitution lemma:

**Lemma 2.2.** Given a program \( g \) as an expression \( e \) that types-checks in an environment \( E = (\Delta; \Pi) \) (\( E \vdash \tau : \tau \rightarrow E' \)), and given a substitution \( \sigma : \Delta_1 \rightarrow \Delta_2 \), where \( \Delta_1 \uplus \Delta_2 \) is a partition of \( \Delta \), then \( e \) also type checks in the substituted environment \( \sigma E = (\Delta_2; \sigma \Pi) \) (\( \sigma E \vdash \tau : \sigma \tau \rightarrow \sigma E' \)).

**Proof.** It is worth mentioning that \( \sigma \text{unit} = \text{unit} \), \( \sigma \text{int} = \text{int} \), and \( \sigma \text{bool} = \text{bool} \) for any substitution \( \sigma \), as substitutions apply only to location variables. Further, if \( \tau \) is a pointer type, then \( \sigma \tau \) is also, because the range of \( \sigma \) is confined to location variables and literals. Thus, if \( \tau_1 \sim \tau_2 \), then \( \sigma \tau_1 \sim \sigma \tau_2 \) for any substitution \( \sigma \).
Then, by induction:
\[ \sigma E \vdash \omega e_1 : \sigma \tau_1 \vdash \sigma E' \vdash \omega e_2 : \sigma \tau_2 \vdash \sigma E'' \]
As discussed above, \( \sigma \tau_1 \sim \sigma \tau_2 \). Therefore, we can apply the \textsc{Equal} rule to get
\[ \sigma E \vdash \omega e_1 = e_2 : \sigma \text{bool} \vdash \sigma E'' \]

\textbf{Read} \( e.f \)

We may assume the prerequisites for the \textsc{Read} rule are true. Then, by induction,
\[ \sigma E \vdash \omega e : \sigma \text{ptr}(\rho) \vdash \sigma \Delta' ; \sigma \Pi' \]

Directly applying the substitution to both sides of the equation:
\[ \sigma \Pi' = \sigma p.f : \sigma \tau \setminus \{ \sigma \ldots \}, \sigma \Pi_1 \]

And as storage compatibility holds over substitutions,
\[ \sigma \tau \sim \sigma \tau \]

Therefore we may apply the \textsc{Read} rule to get the desired result.

\textbf{New} \( \text{new} \{ f_i \mid 1 \leq i \leq n \} \)

As the new reference variable introduced by this rule is fresh, the rule may be applied in the substituted environment. Since the fresh variable is in neither the domain nor the range of the substitution (nor are its concrete field types), the appropriate substituted environment results.

\textbf{Write} \( e_1.f \equiv e_2 \)

The typing preconditions for this rule in the substituted environment follow by double induction, as with \textsc{Plus}. The equality and storage compatibility preconditions follow as in the \textsc{Read} rule.

\textbf{Seq} \( ; e' \)

Follows immediately from applying induction twice.

\textbf{If} \( e_0 \text{ then } e_1 \text{ else } e_2 \)

By hypothesis,
\[ E \vdash \omega \text{ if } e_0 \text{ then } e_1 \text{ else } e_2 : \text{unit} \vdash \sigma E'' \]

The following must have been true to apply this rule:
\[ E \vdash \omega e_0 : \text{bool} \vdash \sigma E' \]
\[ E' \vdash \omega e_1 : \text{unit} \vdash \sigma E_1 \]
\[ E' \vdash \omega e_2 : \text{unit} \vdash \sigma E_2 \]
\[ E'' = E_1 \lor E_2 \]

By induction,
\[ \sigma E \vdash \omega e_0 : \sigma \text{bool} \vdash \sigma E' \]
\[ \sigma E' \vdash \omega e_1 : \sigma \text{unit} \vdash \sigma E_1 \]
\[ \sigma E' \vdash \omega e_2 : \sigma \text{unit} \vdash \sigma E_2 \]

We can apply some of these substitutions:
\[ \sigma E \vdash \omega e_0 : \text{bool} \vdash \sigma E' \]
\[ \sigma E' \vdash \omega e_1 : \text{unit} \vdash \sigma E_1 \]
\[ \sigma E' \vdash \omega e_2 : \text{unit} \vdash \sigma E_2 \]

If we can show that \( \sigma E'' = E_1 \lor E_2 \), we can then apply the \textsc{If} rule to prove that
\[ \sigma E \vdash \omega \text{ if } e_0 \text{ then } e_1 \text{ else } e_2 : \text{unit} \vdash \sigma E'' \]

We therefore argue that if \( E'' = E_1 \lor E_2 \), \( \sigma E'' = \sigma E_1 \lor \sigma E_2 \).

If \( (\Delta; \Pi') = (\Delta_1; \Pi_1) \lor (\Delta_2; \Pi_2) \) then by definition, there are substitutions \( \sigma_1 \) and \( \sigma_2 \) such that
\[ \Delta_1; \Pi_1 = \sigma_1 \Delta' ; \sigma_1 \Pi' \]
\[ \Delta_2; \Pi_2 = \sigma_2 \Delta' ; \sigma_2 \Pi' \]
\[ \Delta = \Delta_1 \cap \Delta_2 \]
\[ \Delta' - \Delta \text{ fresh} \]
\[ r \in \Delta \Rightarrow \sigma r = r \]

We assume that the domain of \( \sigma_1 \) and \( \sigma_2 \) is restricted to \( \Delta' \). If not, we can choose new \( \sigma_1 \), which are so restricted, and which are sufficient for the definition of \( \lor \). As such, the only variables in the domains of both \( \sigma \) and \( \sigma_1 \) are those in \( \Delta \), which \( \sigma_1 \) and \( \sigma_2 \) do not alter.

As \( \Delta' - \Delta \) fresh, and as \( \sigma \)'s domain was defined on variables existing before the \( \lor \) operation, \( \sigma \) can only affect the variables in \( \Delta \). It acts as the identity on the fresh variables in \( \Delta' \). Similarly, the range of \( \sigma \) is also confined to \( \Delta \). Thus \( \sigma \Delta' = (\Delta' - \Delta) \cup \sigma \Delta \) and \( \sigma \Delta' - \sigma \Delta = \Delta' - \Delta \) fresh.

As the domain and range of \( \sigma \) are disjoint by definition, \( \sigma r \) will be outside the domain of \( \sigma \) for any \( r \). Thus \( \sigma = \sigma \circ \sigma \). Additionally, because \( \sigma_1 \) and \( \sigma_2 \) behave as the identity for location variables in \( \Delta \) and because the range of \( \sigma \) cannot include the fresh variables which make up the (non-identity) domain of \( \sigma_1 \) and \( \sigma_2 \), \( \sigma \circ \sigma_1 \circ \sigma_2 = \sigma \circ \sigma_2 \circ \sigma_1 \), which is so.

We can now define two new substitutions, \( \sigma'_1 = \sigma \circ \sigma_1 \) and \( \sigma'_2 = \sigma \circ \sigma_2 \). Clearly, \( \sigma \Delta_1 ; \sigma \Pi_1 = \sigma \sigma_1 \Delta' ; \sigma \sigma_1 \Pi' = \sigma \sigma_1 \Delta' ; \sigma \sigma_1 \sigma \Pi' = \sigma \sigma_1 \Delta' ; \sigma \Pi' \). Similarly, \( \sigma \Delta_2 ; \sigma \Pi_2 = \sigma \sigma_2 \Delta' ; \sigma \Pi' \).

From \( \Delta = \Delta_1 \cap \Delta_2 \), we can conclude that \( \sigma \Delta = \sigma (\Delta_1 \cap \Delta_2) = \sigma \Delta_1 \cap \sigma \Delta_2 \). Then, for all \( r \in \Delta \), there exists a \( r' \in \Delta \) where \( \sigma r = r' \). Also, \( \forall r \in \sigma \Delta \)
\[ \sigma'_1 r = \sigma \sigma_1 r = \sigma \sigma_1 r' \quad \text{(definition of } r') \]
\[ = \sigma r' \quad \text{(from above)} \]
\[ = \sigma r' \quad (r' = r' \lor r' \in \Delta) \]
\[ = r \]

An identical argument may be applied to get \( \sigma'_2 r = r \quad \forall r \in \sigma \Delta \). These facts can now be used to apply the definition of \( \lor \) to determine that if \( (\Delta; \Pi') = (\Delta_1; \Pi_1) \lor (\Delta_2; \Pi_2) \), then \( \sigma (\Delta'; \Pi') = (\sigma \Delta_1; \sigma \Pi_1) \lor (\sigma \Delta_2; \sigma \Pi_2) \).
In particular, for the If rule, \( \sigma E'' = \sigma E_1 \lor \sigma E_2 \). This, in turn, is sufficient (combined with the earlier inductive results) to apply the If rule and conclude that

\[
\sigma E \vdash \omega \text{ if } e_0 \text{ then } e_1 \text{ else } e_2 : \sigma E''
\]

\textbf{IfEqual if } e = e' \text{ then } e_1 \text{ else } e_2

By hypothesis,

\[
E \vdash \omega \text{ if } e = e' \text{ then } e_1 \text{ else } e_2 : \sigma E''
\]

As we are assuming the IfEqual rule was applied, the following must have been true:

\[
\begin{align*}
E & \vdash \omega \text{ e : ptr}(\rho) \land E' \vdash \omega \text{ e' : ptr}(\rho') \land \Delta; \Pi \\
\Delta_1 (\rho = \rho', \Pi) & \vdash \omega \ e_1: \text{unit } \vdash E_1 \\
\Delta_1 (\rho = \rho', \Pi) & \vdash \omega \ e_2: \text{unit } \vdash E_2 \\
E'' &= E_1 \lor E_2
\end{align*}
\]

After applying induction (and recalling that \( \text{unit} = \text{unit} \) and \( \text{bool} = \text{bool} \))

\[
\begin{align*}
\sigma E & \vdash \omega \text{ e : ptr}(\rho) \land E' \vdash \omega \text{ e' : ptr}(\rho') \land \Delta; \Pi \\
\sigma \Delta_1 (\rho = \rho', \Pi) & \vdash \omega \ e_1: \text{unit } \vdash \sigma E_1 \\
\sigma \Delta_1 (\rho = \rho', \Pi) & \vdash \omega \ e_2: \text{unit } \vdash \sigma E_2 \\
\end{align*}
\]

As \( E'' = E_1 \lor E_2 \), we can argue again that \( \sigma E'' = \sigma E_1 \lor \sigma E_2 \). Therefore, we can apply the IfEqual rule again to get

\[
\sigma E \vdash \omega \text{ if } e = e' \text{ then } e_1 \text{ else } e_2 : \sigma E''
\]

which is what needs to be proven.

\textbf{IfTrue if } true \text{ then } e_1 \text{ else } e_2

Follows immediately from induction on \( e_1 \).

\textbf{IfFalse if } false \text{ then } e_1 \text{ else } e_2

Follows immediately from induction on \( e_2 \).

\textbf{Call call } p

By hypothesis,

\[
\Delta; \sigma_1 \Pi_1, \Pi_3 \vdash \omega \text{ call } p: \text{unit } \vdash \Delta \lor \Delta'; \sigma_1 \Pi_2, \Pi_3
\]

Therefore, because we checked this using the Call rule:

\[
\omega(p) = \forall \Delta_1, \Pi_1 \rightarrow \exists \Delta_2, \sigma_2 \Pi_2
\]

\[
\sigma_1 : \Delta_1 \rightarrow \Delta
\]

\( \Delta' \) fresh

\[
\sigma_2 : \Delta' \rightarrow \Delta_2
\]

We can safely assume the variables in \( \Delta, \Delta_2, \Delta_1 \), and \( \Delta' \) are all disjoint as we may always find a typing using disjoint contexts. Therefore, \( \sigma_1 : \Delta_1 \rightarrow \sigma \Delta \). Also, \( \sigma \Delta' = \Delta' \), so \( \sigma \Delta \) fresh and \( \sigma_2 : \sigma \Delta' \rightarrow \Delta_2 \). We thus have:

\[
\omega(p) = \forall \Delta_1, \Pi_1 \rightarrow \exists \Delta_2, \sigma_2 \Pi_2
\]

\[
\sigma_1 : \Delta_1 \rightarrow \sigma \\
\sigma \Delta' \text{ fresh}
\]

\[
\sigma_2 : \sigma \Delta' \rightarrow \Delta_2
\]

We can now apply the \textbf{Call} rule to get:

\[
\sigma \Delta; \sigma_1 \Pi_1, \sigma_3 \Pi_3 \vdash \omega \text{ call } p : \text{unit } \vdash
\]

\[
\sigma \Delta \lor \sigma \Delta'; \sigma_1 \Pi_2, \sigma_3 \Pi_3
\]

\textbf{Nest nest } e . f \text{ in } e' . f'

All the equations still hold true if the substitution is applied equally to both sides, so this rule naturally falls out by inducting twice, then substituting over the equations, similar to \textbf{Write}.

Next, we prove a narrowing rule for consistency:

\textbf{Lemma 2.3.} If we have a memory and adoption information consistent with an environment \( \mu ; a \vdash \Delta; \Pi_1, \Pi_2 \) consistent, then they are also consistent with an environment with fewer permissions \( \mu ; a \vdash \Delta; \Pi_1 \).

\textbf{Proof.} We first prove that \( \mu; A; a \vdash \pi_1, \ldots, \pi_n \dashv \Pi \) if and only if \( \mu; A; a \vdash \pi_1 \dashv \Pi_1 \) and \( \Pi = \bigcup \Pi_i \). The result is susceptible to a simple proof by induction using \( n \in \{0, 1\} \) as base cases since there is only one rule for show consistency of a set of permissions, and the \( B \) sets are empty, the nondeterminism of the split of permissions falls away due to the associativity of the union of disjoint sets.

Given this result, it is easy to see that the the consistency of the smaller set of permissions can be established using the same substitution and assumption sets; we simply end up with a subset of the requirements on the memory that must be fulfilled. Thus the result follows.

\textbf{An important property of this definition is that the transformed environment will be flattened to a subset of the flat permissions:}

\textbf{Lemma 2.4.} If we have \( \mu; a \vdash \Delta; \Pi \) consistent using \( \sigma; (A, A_T) \) such that \( \mu; A; \emptyset \vdash \Pi \dashv \Pi \) and \( (\Delta; \Pi) \succeq (\Delta'; \Pi') \) then \( \mu; A; a \vdash \sigma; \Pi' \vdash \Pi' \). We prove from the definition of the transformation relation, \( \mu; a \vdash \Delta; \Pi' \) consistent using \( \sigma; (A, A_T) \). Therefore, by the definition of consistency, there must be some \( \Pi' \) such that \( \mu; A; \emptyset \vdash \sigma; \Pi' \vdash \Pi' \). Now suppose \( l : \tau_{\text{atom}} \in \Pi' \). Then, \( l : \mu(l) : \tau_{\text{atom}} \).

Also, \( \exists \tau_{\text{atom}} : l : \tau_{\text{atom}} \in \Pi \). If not, we can define \( \mu' = \mu[l \mapsto v] \) where \( v : \tau_{\text{atom}} \) and \( \tau_{\text{atom}} \not\equiv \tau_{\text{atom}} \). Then \( \mu' ; a \vdash \Delta; \Pi \) consistent using \( \sigma; (A, A_T) \) because all the preconditions established by \( \mu \) still hold. But \( \mu' ; a \vdash \Delta'; \Pi' \) consistent using \( \sigma; (A, A_T) \) is clearly not true, as we will not be able to show that \( l : \mu'(l) : \tau_{\text{atom}} \) (because, of course, \( l : \mu'(l) : \tau_{\text{atom}} \)).

But from \( \mu; a \vdash \Delta; \Pi \) consistent using \( \sigma; (A, A_T) \) such that \( \mu; A; \emptyset \vdash \Pi \), we know that \( l : \mu(l) : \tau'_{\text{atom}} \). But then \( \tau_{\text{atom}} \) must equal \( \tau'_{\text{atom}} \), so \( l : \tau_{\text{atom}} \in \Pi \).
LEMMA 2.5. If we have two transformations: $\Delta; \Pi_1 \geq \Delta'_1; \Pi'_1$ and $\Delta; \Pi_2 \geq \Delta'_2; \Pi'_2$ where the fresh variables introduced are disjoint ($\Delta'_1 \cap \Delta'_2 = \emptyset$), then the two transformations can be merged: $\Delta; \Pi_1, \Pi_2 \geq \Delta'_1; \Pi'_1, \Pi'_2$ where $\Delta'_1 = \Delta'_1 \cup \Delta'_2$.

**Proof.** Suppose we have some $\mu, a, \sigma, A, \nu, A'$ such that $\mu; a \vdash \Delta; \Pi_1, \Pi_2$ consistent using $\sigma(A, \nu, A')$. Then, from Lemma 2.4 and the definition of transform (2), there are $A'_1$ and $\sigma_1$ such that $\mu; a \vdash \Delta; \Pi_1$ consistent using $\sigma_1(A, A')$. Thus, using the definition of transform again, there exists $A_1$ and $\sigma_1$ such that $\mu; a \vdash \Delta; \Pi'_1$ consistent using $\sigma_1(A, A_1)$. Similarly, there exists $A_2$ and $\sigma_2$ such that $\mu; a \vdash \Delta; \Pi'_2$ consistent using $\sigma_2(A, A_2)$.

From the definition of memory consistency we may conclude that the following facts must be true:

\[
\sigma: \Delta'_1 \rightarrow \emptyset
\]
\[
\sigma_2: \Delta'_2 \rightarrow \emptyset
\]
\[
\forall \tau. (l: \tau \in \Delta \Rightarrow (l \in a) \Rightarrow \mu; A, A_1; \Pi \vdash \sigma \Pi_1 \nvdash \Pi_1
\]
\[
\mu; A, A_1; \Pi_1 \vdash \sigma \Pi_1 \nvdash \Pi_1
\]
\[
\mu; A, A_1; \Pi_1 \vdash \sigma_1 \Pi_1 \nvdash \Pi_1
\]
\[
\mu; A, A_1; \Pi_1 \vdash \sigma_2 \Pi_2 \nvdash \Pi_2
\]
\[
(l: \tau_{atom}) \in \Pi_1 \Rightarrow \mu(l): \tau_{atom}
\]
\[
(l: \tau_{atom}) \in \Pi_2 \Rightarrow \mu(l): \tau_{atom}
\]
\[
t(v_1, \ldots, v_n) \in A_1 \Rightarrow A_2, A_1 \cup \Pi \vdash [v_1 \rightarrow v_1, \ldots, v_n \rightarrow v_n], T(t) = t
\]
\[
t(v_1, \ldots, v_n) \in A_2 \Rightarrow A_2, A_1 \cup \Pi \vdash [v_1 \rightarrow v_1, \ldots, v_n \rightarrow v_n], T(t) = t
\]

Let us define $\sigma': \Delta' \rightarrow \emptyset$ as follows:

\[
\sigma'(r) = \begin{cases} 
\sigma(r) & \text{if } r \in \Delta' \cap \Delta' \nvdash \emptyset \\
\sigma_1(r) & \text{if } r \in \Delta'_1 \nvdash \Delta' \nvdash \emptyset \\
\sigma_2(r) & \text{if } r \in \Delta'_2 \nvdash \Delta' \nvdash \emptyset
\end{cases}
\]

This substitution is well-defined, as $\Delta'_1 \nvdash \Delta'_2$ and $\Delta'_2 \nvdash \Delta'$ are disjoint. Because $\sigma_1 \geq \sigma_0 \geq \sigma$, $\forall r \in \Delta'_1, \sigma'(r) = \sigma_1(r)$ and thus $\sigma'(\Pi'_1) = \sigma_1(\Pi'_1)$). Similarly, $\forall r \in \Delta'_2, \sigma'(r) = \sigma_2(r)$ and thus $\sigma'(\Pi'_2) = \sigma_2(\Pi'_2)$. Also, from the definition of $\sigma'$, $r \in \Delta \Rightarrow \sigma' r = \sigma r$, so $\sigma' \geq \sigma$.

Suppose $A, A_1 \vdash \Gamma = \emptyset$. Then $A, A_1 \vdash A, A_1 \cup A' \vdash \Gamma = \emptyset$. Why? Because the only effect of adding assumptions to $A_1$ can have on the consistency of $\Gamma$ is with the rule CB-AxiomT which would serve to make something which had been undefined true. But no truth values were undefined in the original proof (or it wouldn’t have been a proof). Therefore, the same proof may be reiterated with extra unused assumptions. This in turn implies that if $\mu; A, A_1; \Pi \vdash \sigma_1 \Pi \nvdash \Pi$, then $\mu; A, A_1; A, A_1 \cup A' \vdash \sigma_1 \Pi \nvdash \Pi$ because the proof trees for the boolean formulae will not change out, and the the remainder of the assumption consistency rules do not refer to $A_1$ and so will not be affected by its contents. Again, the same proof may be reused within the larger set of assumptions.

We may therefore modify the above facts:

\[
\mu; A, A_1; A, A_1 \cup A_2; \emptyset \vdash \sigma \Pi \nvdash \Pi
\]
\[
\mu; A, A_1; A, A_1 \cup A_2; \emptyset \vdash \sigma \Pi \nvdash \Pi
\]
\[
(l: \tau_{atom}) \in \Pi_2 \Rightarrow \mu(l): \tau_{atom}
\]
\[
(l: \tau_{atom}) \in \Pi_2 \Rightarrow \mu(l): \tau_{atom}
\]
\[
t(v_1, \ldots, v_n) \in A_2 \Rightarrow A, A_1 \cup A_2 \vdash [v_1 \rightarrow v_1, \ldots, v_n \rightarrow v_n], T(t) = t
\]
\[
t(v_1, \ldots, v_n) \in A_2 \Rightarrow A, A_1 \cup A_2 \vdash [v_1 \rightarrow v_1, \ldots, v_n \rightarrow v_n], T(t) = t
\]

We now have enough facts to directly apply the definition of memory consistency and get:

\[
\mu; a \vdash \Delta; \Pi'_1, \Pi'_2 \text{ consistent using } \sigma(A, A_1, A_2)
\]

Thus, for any $\mu, a, \sigma, A, \nu, A'$ such that $\mu; a \vdash \Delta; \Pi_1, \Pi_2$ consistent using $\sigma$, we may produce an $A_1, A_2 \vdash \sigma(A_1, A_2)$ such that $\mu; a \vdash \Delta; \Pi'_1, \Pi'_2$ consistent using $\sigma(A_1, A_2)$. Therefore, by definition, $\Delta; \Pi_1, \Pi_2 \geq \Delta; \Pi'_1, \Pi'_2$. 

After adding TRANSFORM, we must re-prove Lemmas 2.1 and 2.2.

LEMMA 2.6 (2.1). Given a transformation $(\Delta; \Pi \geq \Delta'; \Pi')$ and an environment $(\Delta; \Pi_1)$, the corresponding larger sets also form a transformation $(\Delta \cup \Delta_1; \Pi_1 \geq \Delta'_1 \cup \Delta_1; \Pi'_1)$. Therefore, given a type-checked program $q \vdash g: \omega$, an expression $e$ that type-checks $(\Delta; \Pi_1 \geq \Delta'; \Pi'_1)$ under TRANSFORM, and an environment $\Delta_1: \Pi_2$, then $e$ also type-checks with a larger set of permissions $(\Delta \cup \Delta_1; \Pi_1, \Pi_2 \geq \Delta'_1 \cup \Delta_1; \Pi'_1, \Pi'_2)$ in which the unused permissions are not changed. That is, Lemma 2.1 holds even with the TRANSFORM rule added.

**Proof.** It should be clear that adding extra, unused variables will not affect consistency. Thus we may conclude that $(\Delta \cup \Delta_1; \Pi_1 \geq \Delta'_1 \cup \Delta_1; \Pi'_1)$. Also, trivially $\Delta_1; \Pi_1 \geq \Delta_1; \Pi_1$ as whenever $\Delta_1; \Pi_1$ is consistent, $\Delta_1; \Pi_1$ is consistent. This second transformation introduces no fresh variables. Therefore, we may apply Lemma 2.5 to get $(\Delta \cup \Delta_1; \Pi_1 \geq \Delta'_1 \cup \Delta_1; \Pi'_1)$. 

\[\text{That the } \geq \text{ relation on substitutions is transitive follows immediately from its definition} \]
This fact may immediately be used with Lemma 2.1 to prove the second statement in this lemma. □

Lemma 2.7 (2.2). Given a transformation \((\Delta; \Pi \geq \Delta'; \Pi')\) and a substitution \((\sigma; \Delta \cup \Delta' \rightarrow \Delta'')\), the substituted environments also form a transformation \((\sigma\Delta; \Pi \geq \sigma\Delta'; \Pi')\). Therefore, given a program \(g\) as an expression \(e\) that type-checks under Transform in an environment \(E = (\Delta; \Pi)\)
\((E \geq E_1 \vdash e : \tau \rightarrow E_1' \geq E')\), and given a substitution \(\sigma: \Delta_1 \rightarrow \Delta_2\) where \(\Delta_1 \cup \Delta_2\) is a partition of \(\Delta\), then \(e\) also type checks in the substituted environment \(\sigma E = (\Delta_2; \Pi)\)
\((\sigma E \geq \sigma E_1 \vdash e : \sigma \tau \rightarrow \sigma E_1' \geq \sigma E')\).

Proof. Let \(E = (\Delta; \Pi)\) and \(E' = (\Delta'; \Pi')\). By definition, \(\sigma\Delta; \Pi \geq \sigma\Delta'; \Pi'\) if and only if
\[
\forall \mu; a, \sigma_1, A, \Delta \vdash (\mu; a) \vdash \sigma\Delta; \Pi \text{ consistent using } \sigma_1; (A, \Delta, A_T) \Rightarrow
\exists \sigma_2 \sigma_2' \sigma_3 \sigma_4 (\mu; a) \vdash \sigma_2 \Pi \vdash \tilde{\Pi}'
\]
So let us select arbitrary \(\mu; a, \sigma_1, A, \Delta, A_T\) such that \(\mu; a \vdash \sigma_1 E\) consistent using \(\sigma_1; (A, \Delta, A_T)\). Then by definition:
\[
\sigma_1 : \sigma\Delta \rightarrow \emptyset
\]
\[
\mu; A, \emptyset \vdash \sigma_1 \Pi \vdash \tilde{\Pi}
\]
\[
(l : \tau_{atom}) \in \tilde{\Pi} \Rightarrow (l : \tau_{atom})
\]
\[
(l : \tau < l') \in \Delta \Rightarrow (l : \tau < l') \in a
\]
\[
t(\nu_1, \ldots, \nu_n) \in A_T \Rightarrow A \vdash [r_1 \rightarrow \nu_1, \ldots, r_n \rightarrow \nu_n] T(t) = \text{true}
\]
If we now define \(\sigma_2 = \sigma_1 \sigma\), we can conclude that
\[
\sigma_2 : \Delta \rightarrow \emptyset
\]
\[
\mu; A, \emptyset \vdash \sigma_2 \Pi \vdash \tilde{\Pi}
\]
And therefore,
\[
\mu; a \vdash \Delta; \Pi \text{ consistent using } \sigma_2; (A, \Delta, A_T)
\]
Because \(\Delta; \Pi \geq \Delta'; \Pi'\), this in turn implies that
\[
\mu; a \vdash \Delta'; \Pi' \text{ consistent using } \sigma_2; (A, \Delta, A_T)
\]
which is to say:
\[
a \text{ is acyclic}
\]
\[
\sigma_2 : \Delta' \rightarrow \emptyset
\]
\[
\mu; A, \emptyset \vdash \sigma_2 \Pi' \vdash \tilde{\Pi}'
\]
\[
(l : \tau_{atom}) \in \tilde{\Pi}' \Rightarrow (l : \tau_{atom})
\]
\[
(l : \tau < l') \in \Delta \Rightarrow (l : \tau < l') \in a
\]
\[
t(\nu_1, \ldots, \nu_n) \in A_T \Rightarrow A \vdash [r_1 \rightarrow \nu_1, \ldots, r_n \rightarrow \nu_n] T(t) = \text{true}
\]
Substituting again with \(\sigma_2 = \sigma_1 \sigma\) gives:
\[
\sigma_1 : \sigma\Delta' \rightarrow \emptyset
\]
\[
\mu; A, \emptyset \vdash \sigma_1 \Pi \vdash \tilde{\Pi}'
\]
Therefore, by definition,
\[
\exists \sigma_2 \sigma_2' \sigma_3 \sigma_4 (\mu; a) \vdash \sigma_2 \Pi \vdash \tilde{\Pi}' \text{ consistent using } \sigma_1; (A, \Delta, A_T)
\]
which is what needed to be proven.

Now let \(E = (\Delta; \Pi)\), \((E \geq E_1 \vdash e : \tau \rightarrow E_1' \geq E')\), and \(\sigma: \Delta_1 \rightarrow \Delta_2\) where \(\Delta_1 \cup \Delta_2\) is a partition of \(\Delta\). As above, \(\sigma E \geq \sigma E_1\) and \(\sigma E_1' \geq \sigma E'\). Let \(E_1 = (\Delta'; \Pi)\). Then define \(\sigma': \Delta_1 \cap \Delta'' = (\Delta' - (\Delta_1 \cap \Delta''))\) as \(\sigma' = \sigma|\Delta_1 \cap \Delta''\). By Lemma 2.1, \(\sigma' E_1 \vdash e : \tau \rightarrow \sigma' E_1\). But \(\sigma' E_1 = \sigma E_1\) from its definition, and, as the typing can only introduce fresh variables, \(\sigma' E_1' = \sigma E_1'\). Thus, \(\sigma E \geq \sigma E_1 \vdash e : \tau \rightarrow \sigma E_1' \geq \sigma E'\).

Lemma 2.8. The following rules hold:
\[
E \equiv E \quad \Delta; \Pi \geq \Delta; \emptyset \quad \Delta; \emptyset \equiv \Delta; \text{true}
\]
\[
\Delta; \Gamma \land \Gamma' \equiv \Delta; \Gamma, \Gamma' \quad \Delta; \Gamma \equiv \Delta; \land \Gamma
\]
\[
\Delta; t(p_1, \ldots, p_n) \equiv \Delta; [r_1 \rightarrow p_1, \ldots, r_n \rightarrow p_n] T(t)
\]
\[
\Delta; \Pi \geq \Delta; \Gamma \Rightarrow (\Delta; \Pi \equiv \Delta; \Pi, \Gamma) \quad \Delta; \Gamma \equiv \Delta; \Gamma \lor \Pi
\]
\[
\Delta; \Gamma \land \Delta; \Gamma' \equiv \Delta; \Gamma \land \Delta; \Gamma'\quad \Delta; \Gamma \equiv \Delta; \Gamma \land \Pi
\]
\[
\Delta; \emptyset \equiv \Delta; B \rightarrow B
\]
\[
\Delta, \Gamma \equiv \Delta; \Gamma \lor \Pi \quad \Delta; \Pi \equiv \Delta; \emptyset \rightarrow \Pi
\]
\[
\Delta; \emptyset \equiv \Delta; B \rightarrow B
\]

Reduce:
\[
\Delta, \Gamma \equiv \Delta; \Gamma \lor \Pi \quad \Delta; \Pi \equiv \Delta; \emptyset \rightarrow \Pi
\]

Carve-Out:
\[
\Gamma \equiv k : \tau \rightarrow k' \land k \neq k_1 \land \ldots \land k \neq k_n
\]
\[
B = \{k_1 : \tau, \ldots, k_n : \tau\}
\]
\[
\Delta; k' \rightarrow \tau \land \{k : \tau, B\}, \Gamma \equiv \Delta; k' \rightarrow \tau \land \{k : \tau, B\}, \Gamma
\]

Pack:
\[
(\Delta; k : \text{ptr}(\rho). [r \rightarrow \rho] \Pi) \equiv (\Delta; k : \exists r. \text{ptr}(r) \Pi)
\]

Unpack:
\[
r' \text{ is fresh}
\]
\[
(\Delta; k : \exists r. \text{ptr}(r) \Pi) \equiv (\{r\}' \Delta; k : \text{ptr}(r'). [r \rightarrow r'] \Pi)
\]

Proof. In the proof, we use \(A_T' = A_T, \sigma' = \sigma\) except when otherwise stated.

\[
E \equiv E
\]
Trivial.

\[
\Delta; \Pi \geq \Delta; \emptyset
\]
This case follows immediately since \(\mu; A, \emptyset \vdash \emptyset \downarrow \{\} \) is an axiom.

\[
\Delta; \emptyset \equiv \Delta; \text{true}
\]
The \(\leq\) direction is already established by the previous case. The \(\geq\) direction is established by the permission consistency rule for \(\Gamma = \text{true}\).

\[
\Delta; \Gamma \land \Gamma' \equiv \Delta; \Gamma \land \Gamma'
\]
This case works because both sides require \(A \vdash \Gamma = \text{true}\) and \(A \vdash \Gamma' = \text{true}\).
\(\Delta; \Gamma \equiv \Delta; \neg \Gamma\)

This case works because both sides require \(A \vdash \Gamma = \text{true}\).

\(\Delta; t(p_1, \ldots, p_n) \equiv \Delta; [r_1 \rightarrow p_1, \ldots, r_n \rightarrow p_n]T(t)\)

For the \(\geq\) direction: the left side can achieve consistency only if \(t(\sigma p_1, \ldots, \sigma p_n) \in AT\), which requires in turn that \(A \vdash \sigma r_1 \rightarrow p_1, \ldots, \sigma r_n \rightarrow p_n]T(t) = \text{true}\) which (since we assume the correct number of parameters to \(t\)) is equivalent to \(A \vdash \sigma r_1 \rightarrow \sigma p_1, \ldots, \sigma r_n \rightarrow \sigma p_n]T(t) = \text{true}\) which is precisely what is needed to prove consistency of the right-hand side.

For the \(\leq\) direction, let \(A'_T = AT \cup \{t(\sigma p_1, \ldots, \sigma p_n)\}\), which is no obstacle to consistency because the right-hand side requires this fact. With this addition, proving consistency of the right-hand side is straightforward.

\((\Delta; \Pi \geq \Delta; \Gamma) \Rightarrow (\Delta; \Pi \equiv \Delta; \Pi, \Gamma)\)

The \(\leq\) direction of the equivalence to prove follows immediately since removing \(\Gamma\) imposes no obstacle to consistency. For the \(\geq\) direction, the antecedent shows that \(\Gamma\) can be found consistent, possibly with a new \(AT\) that we shall call \(A'_T\). Now let \(A'_T = AT \cup A'_T\). The new permissions \(\Pi, \Gamma\) are consistent as before and the expanded \(A'_T\) will still be checkable since it is composed of parts that were checkable previously.

\(\Delta; \neg \Gamma \geq \Delta; \Gamma \rightarrow \Pi\)

This case follows immediately because the left-hand side requires \(A \vdash \Gamma = \text{false}\) which enables the right-hand-side to be proved.

\(\Delta; \Pi \equiv \Delta; \Gamma \rightarrow \Pi\)

The \(\geq\) direction is similar to the last case, the result follows since the left-hand side requires \(A \vdash \Gamma = \text{true}\). The \(\leq\) direction is also similar: the right side requires \(A \vdash \Gamma = \text{true}\) and thus the proof of \(\Gamma \rightarrow \Pi\) requires that we have \(\Pi\), and the consistency of the left-hand side is easily established.

\(\Delta; \Pi \equiv \Delta; 0 \rightarrow \Pi\)

This case follows immediately from the consistency axiom for \(B_2 \rightarrow \Pi\) specialized for the case \(B_2 = \emptyset\):

\[
\nu; A; B_1 \vdash \Pi \downarrow \tilde{\Pi} \\
\mu; A; B_1 \vdash 0 \rightarrow \Pi \downarrow \tilde{\Pi}
\]

\(\Delta; 0 \equiv \Delta; B \rightarrow B\)

The \(\leq\) direction is a special case of the second case. The \(\geq\) direction is handled by constructing a proof for consistency with one copy of CP-IMP (with \(B_1 = 0, B_2 = \sigma B\)) and multiple copies of CP-UNION until we have split the permissions to single keys and apply CP-IDENTITY to cancel each \(\sigma\beta\) against itself.

\(\Delta; B_1 \rightarrow B_2, (B_2, B_3) \rightarrow \Pi_4 \geq \Delta; (B_1, B_3) \rightarrow \Pi_4\)

This rule is a generalization of the linear modens poenens rule, which is achieved when \(B_1 = B_3 = \emptyset\).

When proving this rule, we may without loss of generality assume all variables have been substituted away (\(\Delta = \emptyset\) and \(\sigma\) is the empty substitution) because if there are variables, the consistency rule will immediately substitute them away.

Now let \(B_2 = \{\beta_1, \ldots, \beta_n\}\), and thus from consistency on the left we have the following facts:

\[
\nu; A; B_1 \vdash \beta_1 \downarrow \tilde{\Pi}_2 \quad B_1 = \sum_{1 \leq i \leq n} B_{1i}
\]

\[
\mu; A; \beta_1, \ldots, \beta_n, B_3 \vdash \Pi_4 \downarrow \tilde{\Pi}_4
\]

If \(B_2\) is empty \((n = 0)\), then so must be \(B_1\) and the result is trivial. The case for \(n > 1\) can be handled by repeatedly applying the \(n = 1\) case. Thus without loss of generality, we may assume \(n = 1\) and thus \(B_2 = \{\beta\}\). So to summarize, we have the situation:

\[
\mu; A; B_1 \vdash \beta \downarrow \tilde{\Pi}_2 \quad \mu; A; \beta, B_3 \vdash \Pi_4 \downarrow \tilde{\Pi}_4
\]

\[
\tilde{\Pi} = \tilde{\Pi}_2 \uplus \tilde{\Pi}_4
\]

Now we prove by induction over the derivation of the second rule that whenever we have these three rules, we also have

\[
\mu; A; (B_1, B_3) \vdash \Pi_4 \downarrow \tilde{\Pi}
\]

Proving this “mini-lemma” requires that we examine the following cases for the last step in the derivation of \(\mu; A; \beta_1, B_3 \vdash \Pi_4 \downarrow \tilde{\Pi}\):

CP-Empty \(\Pi_4 = \emptyset\) (Impossible)

CP-True \(\Pi_4 = \Gamma\) (Impossible)

CP-FalseImp \(\Pi_4 = \Gamma \rightarrow \Pi\) where \(A \vdash \Gamma = \text{false}\) (Impossible)

CP-ImpTrue \(\Pi_4 = \Gamma \rightarrow \Pi'\) (Follows immediately by induction)

CP-Union \(\Pi_4 = \Pi_1, \Pi_2\)

Here \(\beta\) must appear on the left-hand side of one of the two facts above the line; we use induction on that one, and the other remains as before, and the desired result follows.

CP-Identity \(\beta, B_3 = \Pi_4 = \{I : \tau\}\)

In this case, \(\Pi_4 = \emptyset\). Also, we must have \(\beta = (I : \tau), B_3 = \emptyset\), which means the required fact is already established.

CP-Field Here we have

\[
B'_1 = \sum_{(t', \sigma' \epsilon) \in A} B'_{t', \sigma'}
\]

\[
\mu; A; B'_{t', \sigma'} \vdash t' \downarrow \tilde{\Pi}'_{t', \sigma'}
\]

\[
\tilde{\Pi}'_{t', \sigma'} = \biguplus_{(t', \sigma' \epsilon) \in A} \tilde{\Pi}'_{t', \sigma'}
\]

\[
\mu; A; B'_2 \vdash \mu(I : \tau \downarrow \tilde{\Pi}'_2 : t_{atom})
\]

we have two cases, either \(\beta \in B'_1\) or \(\beta \in B'_2\) (or both). In the former case, that means that \(\beta \in\)
For some \( l' : \tau' : l \in A._a \). Then since \( \hat{\Pi}_1 \) is disjoint with \( \Pi_4 = \hat{\Pi}_1 \sqcup \Pi_2 \sqcup \{ l : \tau_{\text{atom}} \} \) then it is certainly disjoint with all the \( \hat{\Pi}_{\nu.,\tau'} \) that partition \( \hat{\Pi}_1 \). Then by induction for the \( l' : \tau' : l \) case, and the associativity of \( \sqcup \), we achieve the desired result.

If on the other hand \( \beta \in B_1^1 \), then if \( \tau \) is atomic, we are done since \( \hat{\Pi}_1' = \{ \} \). Otherwise if \( \tau \) is existential, then we can use induction on the permissions packed into the existential and again achieve our result.

Thus with this mini-lemma (which is also used in the following case), we can now state

\[
\mu; A; \emptyset \vdash (B_1, B_3) \Rightarrow \Pi_4 \sqsubseteq \hat{\Pi}
\]

which enables us to prove consistency of the right-hand side.

**Carve-Out** As before, we ignore variables. Now the expansion of the \( \cdot : \}\backslash \{ . . . \} \) adds adoption facts to a linear implication.

Thus \( k' : \tau' \backslash \{ k_1 : \tau_1, \ldots, k_n : \tau_n \} \), \( k : \tau : \tau' \), \( k \neq k_1, \ldots, k \neq k_n \) where each \( k \) has no variables (is a \( l \)) in a consistent state \( \mu; A \) using \( \{ A_\nu, A_\tau \} \) requires

\[
\mu; A; l_1 : \tau_1, \ldots, l_n : \tau_n \vdash k' : \tau' \sqsubseteq \hat{\Pi}
\]

\[
(l_1 : \tau_1 : l' \in A_\nu \quad (l : \tau : l' \in A_\nu \quad l \neq l_i
\]

Now let \( \{ l'' : \tau'' : l \in A_\nu \} \) be enumerated in the following order for simplicity (where \( (l_{n+1} : \tau_{n+1}) = (l : \tau) \)):

\[
\{ l_1 : \tau_1, \ldots, l_n : \tau_n, l_{n+1} : \tau_{n+1}, \ldots, l_m : \tau_m \}
\]

Now since adoption is acyclic \( l' \neq l_i \) for any \( 1 \leq i \leq m \), and so the last consistency rule for permissions must be the one to prove the consistency of \( k' : \tau \), which means we have the following facts:

\[
\mu; A; B_i \vdash l_i : \tau_i \sqsubseteq \hat{\Pi}_i
\]

\[
(l_1 : \tau_1, \ldots, l_n : \tau_n) = B_0 + \sum_{1 \leq l \leq m} B_i
\]

\[
\mu; A; B_0 \vdash \mu(l') : \tau' \sqsubseteq \hat{\Pi}_0 \sqcup \tau_{\text{atom}}
\]

\[
\hat{\Pi} = \{ l' : \tau' \} \sqcup \hat{\Pi}_0 \sqcup \biguplus_{1 \leq l \leq m} \hat{\Pi}_i
\]

Of interest here is \( B_{n+1} \). If it is empty, we can create \( B_1 = B_i \) except for \( B_{n+1}' = \{ l : \tau \} \). Now we can easily prove \( \mu; A; l : \tau \vdash l : \tau \sqsubseteq \{ \} = \hat{\Pi}_i \), and keeping all other \( \hat{\Pi}'_i = \hat{\Pi}_i \), we can prove

\[
\mu; A; B'_i \vdash l_i : \tau_i \sqsubseteq \hat{\Pi}'_i
\]

\[
(l_1 : \tau_1, \ldots, l_n : \tau_n, l_{n+1} : \tau_{n+1}) = B_0 + \sum_{1 \leq l \leq m} B'_i
\]

\[
\mu; A; B_0 \vdash \mu(l') : \tau' \sqsubseteq \hat{\Pi}_0 \sqcup \tau_{\text{atom}}
\]

\[
\hat{\Pi}' = \{ l' : \tau' \} \sqcup \hat{\Pi}_0 \sqcup \biguplus_{1 \leq l \leq m} \hat{\Pi}'_i
\]

\[
\mu; A; l : \tau, l_1, \ldots, l_n : \tau_n \vdash l' : \tau' \sqsubseteq \hat{\Pi}'
\]

\[
\mu; A; l : \tau, l_1, \ldots, l_n \vdash \rightarrow l' : \tau' \sqsubseteq \hat{\Pi}'
\]

which permits us to prove consistency of the right-hand side.

But suppose \( B_{n+1} \) is not empty: \( B_{n+1} = l_j : \tau_j, B'' \). In that case, we use the mini-lemma from the previous case to convert

\[
\mu; A; B_j \vdash l_j : \tau_j \sqsubseteq \hat{\Pi}_j
\]

\[
\mu; A; l_j : \tau_j, B_{n+1}' \vdash l : \tau \sqsubseteq \hat{\Pi}_{n+1}
\]

\[
\hat{\Pi}_{n+1}' = \hat{\Pi}_j \sqcup \hat{\Pi}_{n+1}
\]

into

\[
\mu; A; B_j, B'' \vdash l : \tau \sqsubseteq \hat{\Pi}_{n+1}'
\]

to which we add the axiom

\[
\mu; A; l_j : \tau_j \vdash l_j : \tau_j \sqsubseteq \{ \} = \hat{\Pi}_j
\]

and produce a new proof of our original fact using \( B_{n+1}' = B_j, B'' \). If \( \Pi_j \vdash \{ l : \tau_j \} \) and all the other \( B_i = B_i, \hat{\Pi}_i = \hat{\Pi}_i \). If this new \( B_{n+1}' \) is still not empty we can repeat this process. Eventually because of linearity (we have a partition) and finiteness, we end up with a \( B_{n+1}' \) which is empty and can proceed.

**Pack** \( \Delta ; k : \text{ptr}(\rho), [r \rightarrow \rho] \Pi \geq (\Delta ; k : \exists r. \text{ptr}(r) \sqsubseteq \Pi) \)

Let \( l = sk \). Associativity of upthus and consistency on the left requires that we have the following facts:

\[
\mu; A; \emptyset \vdash l' : \tau' \sqsubseteq \hat{\Pi}_{l', \tau'}
\]

\[
\hat{\Pi}_1 = \bigcup_{l' : \tau' \sqsubseteq \hat{\Pi}} \hat{\Pi}_{l', \tau'}
\]

\[
\mu; A; \emptyset \vdash \mu(l) : \text{ptr}(\sigma l) \sqsubseteq \{ \} : \text{ptr}(\sigma l)
\]

\[
\mu; A; \emptyset \vdash \sigma[r \rightarrow \rho] \Pi \sqsubseteq \hat{\Pi}_3
\]

\[
\hat{\Pi} = \hat{\Pi}_1 \sqcup \{ l : \text{ptr}(\sigma l) \} \sqsubseteq \hat{\Pi}_3
\]

Furthermore, this element of the flattened permissions \( l : \text{ptr}(\sigma l) \) ensures that \( \mu(l) = \sigma l \), some object reference, we call \( \nu \).

Now since \( r \) is fresh, \( \sigma[r \rightarrow \rho] \Pi = [r \rightarrow \sigma r] \Pi = [r \rightarrow \nu] \sigma \Pi \), where \( \sigma, r, r' \neq r \Rightarrow \sigma r' = \sigma \nu \), and thus

\[
\mu; A; \emptyset \vdash \nu[s] : \Pi \sqsubseteq \hat{\Pi}_3
\]

\[
\mu; A; \emptyset \vdash \nu : (\exists r. \text{ptr}(\nu r) \sqsubseteq \Pi) \sqsubseteq \hat{\Pi}_3
\]
which permits us to prove the consistency of the right-hand side.

Unpack

\[ r' \text{ is fresh} \]
\[ (\Delta, k : \exists r.\text{ptr}(r) \text{ with } \Pi) \supseteq \{ \{ r' \} \cup \Delta; k : \text{ptr}(r'), [r \mapsto r'] \Pi \} \]

Let \( l = \sigma k \). For consistency on the left we must have used \( \text{CP=FIELD} \) and thus have the following facts:

\[ \mu; A; \emptyset \vdash \mu(l) : \sigma(\exists r.\text{ptr}(r) \text{ with } \Pi) \Downarrow \Pi_1 \Downarrow r' \]
\[ \Pi_1 = \bigcup_{\nu : r' \in l} \Pi_{1, r'} \]
\[ \mu; A; \emptyset \vdash \mu(l) : \{ l : \text{ptr}(\sigma r) \} \]
\[ \Pi = \Pi_1 \sqcup \Pi_2 \sqcup \{ l : \text{ptr}(\sigma r) \} \]

Furthermore, this element of the flattened permissions \( l : \text{ptr}(\sigma r) \) ensures that \( \mu(l) = \sigma r \), and both equal some object reference, \( \nu \).

Let \( \sigma' = \sigma[r' \mapsto \nu] \). Trivially \( \sigma' \supseteq \sigma \). Then, \( \mu; A; \emptyset \vdash \mu(\sigma' k) : \sigma(\text{ptr}(r')) \Downarrow \{ \nu \} : \text{ptr}(\nu) \). We can use this with the first two facts above to get:

\[ \mu; A; \emptyset \vdash \sigma'(k : \text{ptr}(r')) \Downarrow \Pi_1 \sqcup \{ l : \text{ptr}(\sigma r) \} \]

\((\text{As } \sigma \text{ and } \sigma' \text{ differ only on } r' \text{ (fresh) } \sigma r = \sigma' r.\)\)

The third fact, in turn, requires that

\[ \mu; A; \emptyset \vdash [r \mapsto \nu] \sigma \Pi_1 \Downarrow \Pi_2 \]

But this is identical to

\[ \mu; A; \emptyset \vdash \sigma'([r \mapsto r'] \Pi) \Downarrow \Pi_2 \]

The facts * and *' can be combined using \( \text{CP-UNION} \) to get:

\[ \mu; A; \emptyset \vdash \sigma'(k : \text{ptr}(r'), [r \mapsto r'] \Pi) \Downarrow \Pi_1 \sqcup \Pi_2 \sqcup \{ l : \text{ptr}(\sigma r) \} \]

The union of disjoint sets is well-defined as it duplicates that used to define \( \Pi \) above.

\[ \square \]

**Lemma 2.9.** Given a type-checked program \( g (\vdash g : \omega) \), an expression \( e \) that type-checks in a variable-free environment \( E = (\emptyset; \Pi) \), \( E \vdash_\omega e : \tau \rightarrow E'' \), \( E'' = (\Delta''; \Pi'') \), we do not require \( \Delta'' = \emptyset \) and a memory and adoption information consistent with \( E \) (\( \mu; A \vdash E \) consistent), then either \( e \) is a value or there exists an evaluation step \((\mu; a; e) \rightarrow_g (\mu; a'; e') \) and for any such evaluation step, there exists a substitution (with absolute addresses) on some of the new type variables \( \sigma : \Delta \rightarrow \emptyset \), where \( \Delta \subseteq \Delta'' \) such that the resulting expression type-checks in a new environment \( E' = (\emptyset; \Pi') \) with the substituted result \( E' \vdash_\omega e' : \sigma \tau \rightarrow \Delta'' - \Delta; \sigma \Pi'' \). In addition, the witness \( A' \) for the new consistency will include everything in the witness of the original consistency \( A \) (\( A_\omega \subseteq A', A_\tau \subseteq A'_\tau \)), and furthermore the new flattened permissions will only include locations from the old permissions or newly allocated memory:

\( \text{Dom}(\Pi') \cap \text{Dom}(\mu) \subseteq \text{Dom}(\Pi) \).

**Proof.** Let \( \sigma \) and \( A \) be the (original) witnesses of consistency. We prove the result by induction over the typing derivation, with the following case analysis:

**Unit, Num, True, False, Address**

For all these cases, \( e \) is a value, and thus the result is vacuously satisfied.

**Plus** \( e_1 + e_2 \)

If both \( e_1 \) and \( e_2 \) are values, then they must be integer constants and evaluation to another integer constant is immediate. The memory hasn’t changed and typing is assured and thus we choose the same environment \( E' = E \), an empty substitution \( \sigma = \emptyset \) and use the same witnesses for consistency.

If \( e_1 \) is a value but \( e_2 \) is not, then we get an evaluation and the new environment and witnesses by induction. Since \( e_1 \) is a value, its typing is assured and we can form a typing of all of \( e' \) with the new environment.

If \( e_1 \) is not a value, then by induction, we get a new environment \( E' \) and substitution \( \sigma \) with witnesses for the consistency of the memory after the evaluation of \( e_1 \). Thus we have evaluation of \( e \) and using the substitution lemma can type \( e_2 \) in the (substituted) output environment from \( e' \).

**Equal** \( e_1 = e_2 \)

Analogous to the case for **Plus**. Here we use the fact that substitutions on atomic types always yield atomic types, in fact types that are storage compatible, thus enabling us to preserve the typing \( \tau_1 \sim \tau_2 \).

**Read** \( e_1.f \)

If \( e_1 \) is a value, then it must be an object \( \nu \), and \( \tau_1 = \text{ptr}(\nu) \). We must have therefore \( \Pi = v.f : \tau \setminus \{ B \}, \Pi_1 \) where \( \tau \) is an atomic type. By consistency, we know that \( \mu; A; B \vdash v.f : \tau \Downarrow \Pi_1 \) where \( \Pi_1 \subseteq \Pi \) used check \( \mu \) for consistency, and where every \( B \in B \), we have \( (\beta_i \prec \nu.f) \in A_\omega \) and thus for every \( B_i = B \), \( \beta_i \in \tau_1 \). Since we assume \( \beta_i \prec \nu.f \) and thus the only rule for consistency is the final one, which (since \( \tau \) is already atomic) requires that \( \Pi_1 \supseteq \nu.f : \tau \). Thus the rule for consistency requires that \( \vdash \mu(\nu.f) : \tau \). Given these facts, we can determine that evaluation to \( (\mu; a.e') \) is assured where \( e' = \mu(\nu.f) \).

Now let \( E' = E, \sigma = \emptyset \), and so \( E \vdash_\omega e' : \tau \rightarrow E'' \) is true. Since we have \( \mu' = \mu, a' = a \), consistency is checking the same values.

Otherwise if \( e_1 \) is not a value, then the result follows by induction.

**New** \( \text{new}\{ f_i \mid 1 \leq i \leq n \} \)

In this case, we have \( \tau = \text{ptr}(r) \) for a fresh variable \( r \). Since we assume \( \mu \) is finite while the set of addresses is infinite, evaluation is assured to a new state \( (\mu'; a'; e') \) where \( a' = a, e' = \nu \). Now let \( \sigma = [r \mapsto \nu] \) be a substitution that replaces the fresh variable with \( \nu \), and thus \( \sigma \tau = \text{ptr}(\nu) \). Now let \( E' = \sigma E'' \), it is clear that \( E' \vdash_\omega e' : \tau \rightarrow E' \), and thus all we need to prove is the preservation of consistency. The new set of permissions \( \Pi' \) has additions for the newly allocated fields: \( \{ \nu.f_1 : \tau_{f_1}, \ldots, \nu.f_n : \tau_{f_n} \} \). Now the evaluation ensures us that \( \nu.f_i \notin \text{Rng}(a = a') \), and thus
Let $\mu' \triangleright; a': 0 \vdash \nu, f_i : \tau_i, \downarrow \{\nu, f_i : \tau_i\}$. Next we check that these flattened sets are disjoint with the flattened permissions from the original $\Pi$. Since these permissions were consistent with $\mu$, and because that means $\mu$ must be defined on the locations in these flattened sets, and because the evaluation rule ensures that $\mu f_i$ was not defined for any $f_i$ we get the desired result. We note that this meets the requirements on the domain of the new flattened permissions. Finally, we need to check the full flattened permission set against $\mu'$. It is only changed for the new locations, so it remains consistent with the original flattened permissions, and the changes of evaluation gives it exactly the correct values that correspond to the fields’ initial types and thus we are done.

Write $e_1.f := e_2$

For this type rule, the type of the pointer $e_1$ is $\text{ptr}(\rho)$ and the type $\tau'$ of new value is storage compatible with $\tau'$ the type recorded for the field in the permission. If $e_1$ and $e_2$ are both values, then $e_1$ must be an object reference $\nu$. As with the rule for READ, consistency requires that $\vdash \mu(\nu, f) : \tau'$ and $\tau'$ is atomic. We also have that $e_2 : \tau$, the value is of the required (atomic) type. Since the type rule requires $\tau \sim \tau_i$, we have $e_2 \sim \nu f_i$ and thus we have progress to $(\mu; a; \emptyset)$. Let $E' \equiv E''$, and thus $E'' \vdash \omega (\emptyset) : \text{unit} \vdash E''$ follows immediately. Now, because of the union of disjoint sets computing $\Pi, \Pi_1$ must not flatten to include a requirement that $\mu(\nu, f)$ has type $\tau'$, and thus changing the type of this field will not cause problems, and indeed the new type now matches what is in the store, and so we have consistency. The domain of the flattened permissions remains the same as before.

If $e_1$ is a value, but not $e_2$, the desired result follows immediately by induction. If $e_1$ is not a value, again we use induction, and then also the substitution lemma to achieve the required result.

Seq $e_1; e_2$

If $e_1$ is a value, then since it must have unit type, it can only be $\emptyset$ and thus we have one step of evaluation to $(\mu; a; e_2)$. The type rule permits us to write $E \vdash \omega e_1 \vdash_1 \omega e_2 : \tau \vdash \omega e_2$ and since $e_1 = \emptyset$, we have $E_1 = E$. Let $E'' = E = E_1$ and then typing is preserved trivially. Consistency is also preserved since no part of the relation is affected by evaluation.

If $e_1$ is not a value, then the result follows through application of induction and the substitution lemma.

If $\text{if } e_0 \text{ then } e_1 \text{ else } e_2$

If $e_0$ is a value, then since the type is boolean, it must be true or false. In the former case, we have evaluation immediately to $(\mu; a; e_2)$. Now we know that $E \vdash \omega e_0 : \text{bool} \vdash E' \vdash \omega e_1 : \text{unit} \vdash E_1$ and $E' = E$, and also that $E'' = E_1 \lor E_2$, which means $E_1 = \sigma_1 E''$, and thus the substitution we use is $\sigma = \sigma_1$. Thus we have the typing result needed, and since memory and environment are unchanged, consistency is also as needed. The case for $e_0 = \text{false}$ is analogous.

Now if $e_0$ is not a value, we can use induction and the substitution lemma (including the part that says that substitution carries over $\lor$) to achieve the desired result.

IfEqual if $e_0 = e_1$ then $e_2$ else $e_3$

If both $e_0$ and $e_1$ are values, then given their types, they must be $\nu_1$ and $\nu_2$ respectively, and their types must be $\text{ptr}(\nu_1)$ and $\text{ptr}(\nu_2)$. Now the type rule ensures that $e_2$ type checks in an environment in which the equality is assumed and $e_3$ in the environment where they are assumed not equal:

$\Delta; \nu_1 = \nu_2, \Pi \vdash e_2 : \text{unit} \vdash E_2 \Delta; \nu_1 \neq \nu_2, \Pi \vdash e_3 : \text{unit} \vdash E_3 E'' = E_2 \lor E_3$

If the objects are indeed equal, we have immediate progress to if true then $e_2$ else $e_3$. Furthermore, (it can be easily shown), the equality fact adds no information and can be dropped, and thus we have $E \vdash \omega e_2 : \text{unit} \vdash E_2$. By the definition of $\lor$, we have $E_2 = \sigma_2 E''$ and thus $E \vdash \omega e_2 : \text{unit} \vdash \sigma E''$ which is precisely what we need to use IfTrue to type-check the new program. Since the store and environment are unchanged, consistency also follows. Thus we have achieved the desired result. The case for inequality is completely analogous.

If either $e_0$ or $e_1$ is not a value, then we can use the inductive hypothesis and straightforward reasoning afterwards.

IfTrue if true then $e_1$ else $e_2$

Trivial.

IfFalse if false then $e_1$ else $e_2$

Trivial.

Call call $p$

The type rule, Proc and Program, ensure we have the following facts:

$E = \emptyset; \sigma_1 \Pi_1, \Pi_3$

$E'' = \Delta'; \sigma_1 \Pi_2, \Pi_3$

$\omega(p) = \forall \Delta_1, \Pi_1 \rightarrow \exists \Delta_2, \sigma_2 \Pi_2$

$\sigma_1 : \Delta_1 \rightarrow \emptyset$

$\Delta_1$ fresh

$\sigma_2 : \Delta_2 \rightarrow \Delta_2$

$\Delta_1; \Pi_1 \vdash_\omega g(p) : \text{unit} \vdash \Delta_1'; \sigma_1 \sigma_2 \Pi_2$

$\Delta_1' \cap \Delta_2 = \emptyset$

$\sigma_3 : \Delta_2 \rightarrow \Delta_1'$

In particular, we know that $g(p)$ is defined and thus we have immediate progress to $(\mu; a; g(p))$. By application of the substitution lemma (for $\sigma_1$) and the widening lemma (adding $\Pi_3$ to both sides), we achieve

$E \vdash_\omega g(p) : \text{unit} \vdash \sigma_1 \Delta_1'; \sigma_1 \sigma_3 \sigma_2 \Pi_2, \Pi_3$

Since $\sigma_1$ is idempotent and neither $\sigma_2$ nor $\sigma_3$ have any effect on variables in $\Delta_1 (\Delta_1$ must be disjoint with
both $\Delta_2$ and with fresh variables $\Delta_1$), $\sigma_1 \sigma_1 \sigma_2 \Pi_2 = \sigma_1 \sigma_1 \sigma_2 (\sigma_1 \Pi_2)$. Let $\sigma = \sigma_1 \sigma_2$. Now $\sigma + \Delta = \sigma_1 \sigma_1 \sigma_2 \Delta_1 = \sigma_1 \sigma_2 \Delta_2 = \sigma_1 \Delta_1$ and thus since $\Pi_1$ has no free variables, the previous typing can be written $E \vdash_\omega \phi(p)$: unit $\vdash \sigma E''$ which is the required type preservation. Consistency is preserved trivially since the input environment $E' = E$, memory and adoption are all unchanged.

Nest $\text{nest } e_0 \cdot f_0$ in $\epsilon_1 \cdot f_1$

If both $e_0$ and $e_1$ are values, then the typing rules require (using the unstated canonical forms lemma) that they both be objects $\nu$ and $\nu'$ respectively, and thus we have immediate progress to $(\mu; a'; \emptyset)$ where $a' = a \cup \{(a, f) \prec (a', f')\}$. Typing is preserved too using $E' = E''$ and Unit: $E' \vdash_\omega \emptyset$ : unit $\vdash E''$. Preservation of consistency is more interesting. The starting permissions are

$$\Pi = \{(l_1: \tau_1 \mid l_2: \tau_2) : (l_1 \neq l_2), l_1: \tau_1 \vdash l_2: \tau_2\}, \Pi_1$$

Consistency in the original state gives inequalities (which we do not need for this lemma) and also the following (we may need to use the associativity and commutativity of $+$ and $\ominus$ to rearrange the consistency proof to have this shape):

$$\mu; A; \emptyset \vdash l: \tau \ominus \Pi_2$$

$$\mu; A; l_1: \tau_1, \ldots, l_n: \tau_n \vdash l': \tau' \ominus \Pi_3$$

$$\mu; A; \emptyset \vdash \Pi_1 \ominus \Pi_2 \ominus \Pi_3$$

Furthermore, we have $a \ominus (l_1 \prec l_2)$ for all $1 \leq i \leq n$. Since the adoption relation cannot be cyclic the consistency rule for the second consistency fact above cannot be the self-canceling one, and must instead use adoption. Now if $l' \prec l$ using our original adoption relation, then the consistency proof for the first consistency fact above would include a subrule $\mu; A; \emptyset \vdash l': \tau' \ominus \Pi_4$, but then $\Pi_4$ would include $l': \tau_3'$ atom whereas $\Pi_3 \ni l': \tau_3'$ atom. If these atomic types are the same, then the union of disjoint sets operator will fail at some point. If they are different, then $\mu(l')$ will be required to match two different atomic types, which is not possible. Thus the new addition to $a$ cannot cause a cycle. Now define $A' = A \cup \{l: \tau \prec l'\}$ and thus matches $a'$ as required for consistency. It remains only to show that we can form the new set of flattened permissions and that this set has no more requirements upon $\mu$ than did the previous set.

We consider two cases: whether or not the location was already adopted with this type: $l: \tau \prec l' \in A_\prec$. If it was already adopted, we have $A_\prec = A_\prec, a' = a$ and thus there is no change in the consistency proof and thus we have $\mu; A; l_1: \tau_1, \ldots, l_n: \tau_n \vdash l': \tau' \ominus \Pi_3$ and thus, the consistency proof simply produces a smaller set than it did before, and thus consistency is preserved.

Otherwise, first we note that $\mu; A; \emptyset \vdash \Pi_1 \ominus \Pi_3$ cannot depend on the absence of the new adoption fact. This is due to the ways in which adoption facts are used. In particular, a fact permission $\pi = \Gamma$ cannot depend on the absence of an adoption fact. Similarly for the case $\pi = \Gamma \ominus \Pi$. And the only other place adoption is used is when we use the consistency rule CP-Field. If the consistency proof for $\Pi_1$ uses this rule and depends on the presence or absence of $l: \tau \prec l'$ then the resulting flattened permission sets must include an element of the form $l': \tau_3''$ atom which would have caused consistency problems as described previously. Thus we conclude $\mu; A; \emptyset \vdash \Pi_1 \ominus \Pi_3$.

Next, the new application of CP-Field for $l': \tau'$ is completed by adding the subgoal $\mu; A; \emptyset \vdash l: \tau \ominus \Pi_3$, which means that we have:

$$\mu; A; l_1: \tau_1, \ldots, l_n: \tau_n \vdash l': \tau' \ominus \Pi_3 \ominus \Pi_2$$

$$\mu; A; \emptyset \vdash \Pi_1 \ominus \Pi_3 \ominus \Pi_2$$

This set is the same as before and thus consistency is preserved.

Transform $E \geq E_0 \vdash_\omega e: \tau \ominus E_2 \geq E''$

If $e$ is a value, we are done. Otherwise, we have progress by induction $(\mu; a; e) \rightarrow_\mu (\mu'; a'; e')$, and also have $E'_1$ and $\sigma$ where $E'_1 \vdash e' : \tau \ominus \sigma E_2$ and $\mu; a' \vdash \sigma E'_1$ consistent. Now by the substitution lemma, $\sigma E_2 \geq \sigma E_2''$ and thus we can apply Transform again to achieve:

$$E'_1 \geq E'_1 \vdash e': \tau \ominus \sigma E_2 \geq \sigma E_2''$$

which is the required result. The set of flattened permissions is always a subset of the original ones (Lemma 2.4) and thus the domain result is met.

Lemma 2.10. Given a type-checked program $g (\vdash g: \omega)$, an expression $e$ that type-checks in a variable-free environment $\emptyset; \Pi \vdash_\omega e : \tau \ominus E''$ and a memory and adoption consistent in an environment with more permissions $(\mu; a; \emptyset; \Pi; \Pi_1, \Pi_2, \text{consistent})$, then the evaluation of the expression (if any) will not read or write any field mentioned in the flattening of the extra permissions $(\Pi_1)$.

Proof. First we note that consistency requires first that Dom($\Pi$) $\cup$ Dom($\Pi_2$) $\subseteq$ Dom($\mu$), and second that the two sets of (flattened) permissions refer to different segments of memory: Dom($\Pi$) $\cap$ Dom($\Pi_2$) = $\emptyset$. Otherwise, either they would have overlap (and have the union of disjoint sets be undefined) or they would have conflicting requirements on the store which would make consistency impossible.

Now the statement is trivial if evaluation is already complete ($e$ is a value). Otherwise, we proceed with induction over the length of the evaluation sequence, by first showing the result for one step of evaluation and then showing that we can re-establish the conditions of the lemma.

First, however, we observe that if $E \vdash_\omega v : \tau \ominus E'$ then the resulting flattened set of permissions for any consistent memory will be a (possible improper) subset of the original flattened permissions $\Pi'' \subseteq \Pi$. This follows because the five rules specifically for values Unit/Num/True/False/Address make no change in the environment at all, and the only other one Transform (which may be applied an arbitrary number of times) can only produce a subset of the flattened permissions (by Lemma 2.4).
If evaluation proceeds for one step $\text{Eval}_\mu; ac\mu'; a'e'$ let $\Pi$ be the flattened permissions before evaluation. We make a simple proof by induction over the typing of $e$ that any location $l$ read or written in the evaluation will be present in the flattened permissions: $l \in \text{Dom}(\Pi)$:

**Unit/Num/True/False/Address** No evaluation possible.

**Plus** $e_1 + e_2$

If $e_1$ is not a value, then the next step of evaluation will evaluate $e_1$. The environment used in the typing of $e_1$ is same as the environment for $e$ and thus the result follows immediately by induction. Otherwise, if it is a value, but $e_2$ is not, then we see that the flattened permissions for the environment used to check $e_2$ are a subset, and thus the result follows by induction. If both $e_1$ and $e_2$ are values, then they must be integer constants, and thus the evaluation does not access memory at all.

**Equal** $e_1 = e_2$

Analogous to **Plus**

**Read** $e_1.f$

If $e_1$ is not a value, then the result follows by induction (there is no change in environment). Otherwise, $e_1$ must be a value of type $\text{ptr}(v)$ (given that the environment is empty). Now $E_1$ (the environment after typing the object) must either be the same as $E$, or must be a transformed version of $E$. In either case, it must have an empty $\Delta$, and its permissions $\Pi_1$ must include $\nu.f: \tau \setminus \{\ldots\}$. Using a similar process as was used for the proof of progress, we can show that the flattened permissions $\hat{\Pi}_1$ must include $\nu.f: \tau$. Thus it must be in the flattened permissions for the original environment $\Pi$.

**New** new { ... }

This construct does not access any existing memory locations when evaluated.

**Write** $e_1.f := e_2$

Analogous to **Read**

**Seq** $e_1; e_2$

If $e_1$ is a value, then it must be $()$, and thus evaluation goes forward without any read or write effects. Otherwise, the result follows by induction.

**If/Equality if** $e_0$ then $e_1$ else $e_2$

If $e_0$ is a value, then it must be a boolean constant and evaluation proceeds without affecting memory. Otherwise the result holds by induction.

**IfTrue/IfFalse/Call** Evaluation doesn’t use memory.

**Nest** $e_0.f_0$ in $e_1.f_1$

If $e_0$ is not a value, the result follows by induction. Otherwise, it $e_1$ is not a value, the result follows by induction once we take into account possible application of **Transform**, which as we have seen allow us to to achieve our result still.

The interesting case is the one in which both are values. Now, the permission required (possibly after transformation that only reduce the flattened permissions) ensures that the flattened permissions will include the two locations of the nesting expression, and thus the lemma is proved.

**Transform** $E \geq E_1 \vdash_\omega e : \tau \vdash E_2 \geq E''$

By induction the result is true for $E_1$ and then because of Lemma 2.4, must be true for $E$ as well.

Now we use the preservation aspect of Lemma 2.9 to get $\emptyset; \Pi' \vdash_\omega e' : \tau \vdash_\sigma E''$, where $\text{Dom}(\hat{\Pi}') \cap \text{Dom}(\mu) \subseteq \text{Dom}(\Pi)$. Now since $\text{Dom}(\Pi_1) \subseteq \text{Dom}(\mu)$ and as shown above $\text{Dom}(\hat{\Pi}) \subseteq \text{Dom}(\mu)$. As a result, we get that $\Pi' \uplus \Pi_1$ is well defined. Finally this union of disjoint sets is consistent with $\mu'$ because the first part is already consistent (by preservation), and the second part (which was consistent with $\mu$) only puts requirements on locations that are not allowed to be modified in evaluation (by this lemma). Furthermore, evaluation only adds things to $a$, never removes them, and because consistency of permissions never depends on the absence of adoption information, we have $\mu'; a' \vdash \Pi'; \Pi_1$ consistent, which preserves the conditions of this lemma for another evaluation.

**Theorem** 2.11. Given a type-checked program $g \vdash \omega \mu : \omega$, two expressions $e_1, e_2$ each of which type-checks in a variable-free environment $\emptyset; \Pi_1, \Pi_2$ and a memory and adoption consistent with the combined environments: $(\mu; a \vdash \emptyset; \Pi_1, \Pi_2$ consistent), then when evaluating the expressions in sequence $e_1; e_2$ no state will be accessed by both expressions.

**Proof.** Preservation and widening ensures that when we finish evaluating $e_1$, we have a substitution $\sigma_1 : \Delta_1' \rightarrow \emptyset$ for which $\emptyset; \sigma_1 \Pi_1', \Pi_2 \vdash_\omega e_2 : \text{unit} \vdash E_2'$ and a memory and adoption consistent with the combined environments: $(\mu; a \vdash \emptyset; \Pi_1, \Pi_2$ consistent). Then we apply Lemma 2.10 to ensure that evaluating $e_2$ does not access anything in $\Pi_1'$ (the flattening of $\sigma\Pi_1'$). Furthermore, this set does not include any values for locations in $\text{Dom}(\Pi_2)$. On the other hand, if we follow preservation step by step, we see that the evaluation $e_1$ only accessed locations in $\text{Dom}(\Pi_1')$. Thus the two expressions are separate.
