## Soundness proof of type checking

Dhruv C. Makwana

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>Commentary</td>
<td>2</td>
</tr>
<tr>
<td>B2</td>
<td>Typing Judgements</td>
<td>4</td>
</tr>
<tr>
<td>B3</td>
<td>Operational Semantics Judgements</td>
<td>7</td>
</tr>
<tr>
<td>B4</td>
<td>Proof Judgements</td>
<td>8</td>
</tr>
<tr>
<td>B5</td>
<td>Groups of Rules</td>
<td>10</td>
</tr>
<tr>
<td>B5.1</td>
<td>Typing rules with an $\text{smt} (\Phi \Rightarrow qterm)$ premise</td>
<td>10</td>
</tr>
<tr>
<td>B5.2</td>
<td>Typing rules which change the context</td>
<td>10</td>
</tr>
<tr>
<td>B5.3</td>
<td>Value typing rules</td>
<td>11</td>
</tr>
<tr>
<td>B6</td>
<td>Weakening</td>
<td>12</td>
</tr>
<tr>
<td>B7</td>
<td>Substitution</td>
<td>13</td>
</tr>
<tr>
<td>B7.1</td>
<td>Substitutions preserve SMT results</td>
<td>13</td>
</tr>
<tr>
<td>B7.2</td>
<td>Substitutions can be split up</td>
<td>13</td>
</tr>
<tr>
<td>B7.3</td>
<td>Substitution</td>
<td>13</td>
</tr>
<tr>
<td>B8</td>
<td>Resource Term Lemmas</td>
<td>17</td>
</tr>
<tr>
<td>B8.1</td>
<td>Definition: Normalised contexts</td>
<td>17</td>
</tr>
<tr>
<td>B8.2</td>
<td>Resource contexts typing closed terms must be normalised</td>
<td>17</td>
</tr>
<tr>
<td>B8.3</td>
<td>Non-conditional resources determine context and values</td>
<td>17</td>
</tr>
<tr>
<td>B8.4</td>
<td>Normalised resource context determines structure of heap</td>
<td>18</td>
</tr>
<tr>
<td>B8.5</td>
<td>Well-typed resource value determines its footprint</td>
<td>19</td>
</tr>
<tr>
<td>B8.6</td>
<td>Progress and type preservation for resource terms</td>
<td>19</td>
</tr>
<tr>
<td>B8.7</td>
<td>Resource term reduction is deterministic</td>
<td>24</td>
</tr>
<tr>
<td>B8.8</td>
<td>Resource term reduction is isolated</td>
<td>24</td>
</tr>
<tr>
<td>B9</td>
<td>Progress</td>
<td>25</td>
</tr>
<tr>
<td>B9.1</td>
<td>$\Phi \vdash res \sim res'$ is an equivalence relation</td>
<td>25</td>
</tr>
<tr>
<td>B9.2</td>
<td>$\Phi \vdash res \sim res'$ is preserved by substitution</td>
<td>25</td>
</tr>
<tr>
<td>B9.3</td>
<td>Well-typed spines produce substitutions and the same return type</td>
<td>25</td>
</tr>
<tr>
<td>B9.4</td>
<td>Well-typed values pattern-match successfully</td>
<td>26</td>
</tr>
<tr>
<td>B9.5</td>
<td>$\Phi \vdash \text{to}_f \text{un} ret \sim ret$</td>
<td>32</td>
</tr>
<tr>
<td>B9.6</td>
<td>Statement and proof</td>
<td>32</td>
</tr>
<tr>
<td>B10</td>
<td>Type Preservation</td>
<td>38</td>
</tr>
<tr>
<td>B10.1</td>
<td>$\text{Owned} \langle \tau \rangle$ resource output values have type $\beta_\tau$</td>
<td>38</td>
</tr>
<tr>
<td>B10.2</td>
<td>Type Preservation Statement and Proof</td>
<td>38</td>
</tr>
</tbody>
</table>
### B1 Commentary

Since Core is a first-order language, and we require that all functions and labels are annotated with the correct type, it suffices to only use purely syntactic techniques to prove soundness. This remains true despite the addition of linear types, systems with which are normally proved using logical relations. There are three main components to this: a joint progress-and-type-preservation proof for resource term reduction, a progress theorem and a type-preservation theorem.

Let a resource be called **normalised** if it is either a `pred`, `qpred` or an under-determined conditional resource. Let a resource context `R` be called normalised if it contains only normalised resources. Even though the grammar of resources is richer, we can, in all the proofs relating to well-typed closed resource terms, assume the resource context to be normalised. This is fine because of the following lemma: if a well-typed resource term is closed, then the context in which it is well-typed must be normalised.

Operational semantics for resource term happens to be defined using big-step style; this makes its definition concise and modular, at the cost of making the proof of soundness of resource term reduction more complicated since it requires joint progress and type preservation. The configuration for the operational semantics is a pair of a heap and an annotated and let-normalised Core program.

Heaps only contain normalised resources. Predicates in heaps are optionally tagged with their “definition” `def` (a resource value of the type of the predicate body) and a sub-heap (of the resources used by the definition). This is to support folding and unfolding predicates in the operational semantics, and to capture the idea that predicate encapsulate their contents until opened.

The types of heaps are normalised contexts; the rules for these are straightforward, except the fact a heap with a folded predicate requires there exists a context for which the resource value `def` and `heap` is well-typed. This becomes necessary for proving the progress of pattern-matching for the whole of the annotated and let-normalised Core.

#### Theorem 1 (Progress and type preservation for resource terms)

For all closed resource terms `res` which type check or synthesise `($) Φ; R ⊢ res ⇐ ret` and all well-typed heaps `(Φ ⊢ h ⇐ R)` there exists a resource value `res_val`, context `R'` and heap `(h')`, such that: the value is well-typed `($) Φ; R' ⊢ res_val ⇐ res`; the heap is well-typed `($) Φ ⊢ h' ⇐ R'`; for all frame-heaps `(f)`, the resource term reduces to the resource value without affecting the frame-heap `($) Φ; res_term ⊢ h' + f; res_val`).

The interesting case in the proof of this is folding a predicate; proving this case requires a notion of `footprint` of a resource value: the subheap containing the resources referred to by the value.

#### Theorem 2 (Progress for the annotated and let-normalised Core)

If a top-level expression `texpr` is well-typed `($) Φ; R ⊢ texpr ⇐ ret` and all computational patterns in it are exhaustive, then either it is a value `tval`, or it is unreachable, or for all heaps `(h)`, if the heap is well-typed `($) Φ ⊢ h ⇐ R` then there exists another heap `(h')` and expression `(texpr')` which is stepped to `($) Φ; texpr' ⊢ h' + f; res_val)`. 
The assumption that all computational patterns are exhaustive is justified because they are generated by Cerberus. As one might expect, proving progress requires well-typed patterns successfully produce substitutions. However, this complicated by two things, the solution to which requires the introduction of a relation on SMT terms and resource types, $\Phi \vdash res \sim res'$ (to be read “under constraints $\Phi$, $res$ is related to $res'$”).

The first is that the constraint term generated when typing a computational pattern (this is required to record, in the constraint context, which branch the type system is assuming it is in) is not exactly equal to the values it can match in the operational semantics (nor would we want it to be: the pattern $\text{Cons}(x_1, x_2)$ should match the value $\text{Cons}([\text{pval}_1, \text{Cons}([\text{pval}_2, \text{Nil}_{\beta}]))]$). Hence, we must weaken the notion of equality on types to $\sim$ relatedness, which links the two, so that during the proof can substitute the constraint term $x_1 :: x_2$ at the type-level, and maintain a link to the corresponding value. The second is that the conditions of related conditional resource must remain SMT-equivalent (with reference to a constraint context), so that pattern-match typing and resource term typing are consistent.

**Theorem 3 (Type preservation for the annotated and let-normalised Core)** For all closed and well-typed top-level expressions ($\cdot;\cdot;\Phi; R \vdash texpr \Leftarrow \text{ret}$), well typed heaps ($\Phi \vdash h \Leftarrow R$), frame-heaps ($f$), new heaps ($\text{heap}$), and new top-level expressions ($texpr'$), which are connected by a step in the operational semantics ($\langle h + f; texpr \rangle \rightarrow \langle \text{heap}; texpr' \rangle$), if all top-level functions are annotated correctly, there exists a constraint context ($\Phi'$), sub-heap ($h'$), and resource context ($R'$), such that the constraint context is $\Phi$ extended, the frame is unaffected ($\text{heap} = h' + f$), the sub-heap is well-typed ($\Phi' \vdash h' \Leftarrow R'$), and the top-level expression too ($\cdot;\cdot;\Phi'; R' \vdash texpr' \Leftarrow \text{ret}$).

A few things are noteworthy about the proof. First is that a frame-heap has to be explicitly passed around. Whilst this is inconvenient, it becomes necessary in the EXPL_TOP_SEQ_LETT case. The next is that proof that well-typed spines produce well-typed substitutions require quantifying over the substitutions done so far, so that the inductive case matches up and the substitution so far $\psi$ shows up in the conclusion, ‘closing’ otherwise ‘open’ substitution and terms. One more is in the proof that well-typed patterns produce well-typed substitutions: unfortunately quantifying over the substitutions done so far is not helpful because even the substitution itself can be ‘open’ (refer to free variables). Hence the peculiar typing of pattern-matching, so that all terms are well-scoped. This allows us to induct usefully, and get the required substitution in the output substitution and its type, making way to apply the substitution lemma afterwards. Lastly, we gather constraints throughout the proof, since these are accumulated by the typing rules, during pattern-matching, case and if. Given the constraint context is always well-formed (w.r.t. to the empty contexts), this means that all the constraints must be trivial (though extra effort would be required to show that they are trivially true, for example, showing that default bool cannot occur.
B2 Typing Judgements

In this document, \( \mathcal{C}; \mathcal{L}; \Phi; \mathcal{R} \vdash J \) stands for all defined judgements, listed in the remainder of this section after this paragraph. In particular, it does not stand for \( \mathcal{C} \vdash \text{mem}\_\text{val} \Rightarrow \beta \) or \( \mathcal{C}; \mathcal{L} \vdash \text{term} \Rightarrow \beta \). Furthermore, I assume that lemma B6 (Weakening) and lemma B7.3 (Substitution) (proven for the defined judgements in the referenced sections) hold for these \( (\mathcal{C} \vdash \text{mem}\_\text{val} \Rightarrow \beta, \) and \( \mathcal{C}; \mathcal{L} \vdash \text{term} \Rightarrow \beta) \) judgements.

\[
\text{res\_judge} ::= \\
| \Phi \vdash \text{cmp\_min}(\text{iguard}, \text{iguard}') \leadsto \text{opt\_cmp\_term} \\
\text{given constraints } \Phi, \text{ iguard is potentially included in iguard'} \text{ (or vice-versa) with ordering and minimum opt\_cmp\_term} \\
| \Phi \vdash \text{qpred\_term} \sqsubseteq \text{? qpred\_term}' \leadsto \text{opt\_cmp} \\
\text{given constraints } \Phi, \text{ qpred\_term is potentially included in qpred\_term'} \text{ (or vice-versa) with ordering opt\_cmp} \\
| \Phi \vdash \text{res\_req } \equiv \text{ res\_req}' \leadsto \text{bool} \\
\text{return type equality: given constraints } \Phi, \text{ res is equal to res'} \\
| \Phi \vdash \text{res } \equiv \text{ res}' \\
\text{resource equality: given constraints } \Phi, \text{ res } \text{ is equal to res'} \\
| \Phi \vdash \text{simp\_rec}(\text{res}) \leadsto \text{res}', \text{bool} \\
\text{partial-simplification of resources: given constraints } \Phi, \text{ res partially simplifies (strips ifs) to res'} \\
| \Phi \vdash \text{simp}(\text{res}) \leadsto \text{opt\_res} \\
\text{partial-simplification of resources: given constraints } \Phi, \text{ res attempts a partial simplification (strips ifs) to opt\_res} \\
\]

\[
\text{ret\_judge} ::= \\
| \Phi \vdash \text{ret } \equiv \text{ ret}' \\
\text{return type equality: given constraints } \Phi, \text{ ret is equal to ret'} \\
\]

\[
\text{pat\_judge} ::= \\
| \text{pat};\beta \leadsto \mathcal{C} \text{ with term} \\
\text{computational pattern to context: pat and type } \beta \text{ produces context } \mathcal{C} \text{ and constraint term} \\
| \text{ident\_or\_pat};\beta \leadsto \mathcal{C} \text{ with term} \\
\text{identifier-or-pattern to context: ident\_or\_pat and type } \beta \text{ produces context } \mathcal{C} \text{ and constraint term}
\]
expl_pure ::=  
| \[C ; \Phi \vdash \text{object.value} \Rightarrow \beta\]  
object value synthesises: given \(C\), \text{object.value} synthesises type \(\beta\)  
| \[C ; \Phi \vdash \text{pval} \Rightarrow \beta\]  
pure value synthesises: given \(C\), \text{pval} synthesises type \(\beta\)  
| \[C ; \Phi \vdash \text{pexpr} \Rightarrow \text{pure.ret}\]  
pure expression synthesises: given \(C;\Phi\), \text{pexpr} synthesises a pure (non-resourceful) return type \text{pure.ret}  
| \[C ; \Phi \vdash \text{tpval} \Leftarrow \text{pure.ret}\]  
pure top-level value checks: given \(C;\Phi\), \text{tpval} checks against \text{pure.ret}  
| \[C ; \Phi \vdash \text{tpexpr} \Leftarrow \text{pure.ret}\]  
pure top-level expression checks: given \(C;\Phi\), \text{tpexpr} checks against \text{pure.ret}  

expl_res ::=  
| \[C ; \Phi ; \kappa \vdash \text{pred.ops} \Rightarrow \text{res}\]  
resource \((q)\)predicate operation term synthesis: given \(C;\Phi;\kappa\), \text{pred.ops} synthesises resource \text{res}\]  
| \[C ; \Phi ; \kappa \vdash \text{res.term} \Rightarrow \text{res}\]  
resource term synthesises: given \(C;\Phi;\kappa\), \text{res.term} synthesises resource \text{res}\]  
| \[C ; \Phi ; \kappa \vdash \text{res.term} \Leftarrow \text{res}\]  
resource term checks: given \(C;\Phi;\kappa\), \text{res.term} checks against resource \text{res}\]  

expl_spine ::=  
| \[C ; \Phi ; \kappa \vdash \text{spine :: fun} \Rightarrow \text{ret}\]  
function call spine checks: given \(C;\Phi;\kappa\), compatible \text{spine}, \text{fun} produces an \text{ret}\]  

\(\mathcal{L};\Phi \vdash \text{res.pat}: \text{res} \rightsquigarrow \mathcal{L}';\Phi';\kappa'\)  
resources pattern to context: given constraints \(\Phi\), \text{res.pat} of type \text{res} produces contexts \(\mathcal{L}';\Phi';\kappa'\)  
\(\mathcal{C};\mathcal{L};\Phi \vdash \text{ret.pat}: \text{ret} \rightsquigarrow \mathcal{C}';\mathcal{L}';\Phi';\kappa'\)  
return pattern to context: given context \(\mathcal{C};\mathcal{L};\Phi\), \text{ret.pat} and return type \text{ret} produces contexts \(\mathcal{C}';\mathcal{L}';\Phi';\kappa'\)  
\(\Phi \vdash \text{ret.pat}: \text{ret} \rightsquigarrow \mathcal{C}';\mathcal{L}';\Phi';\kappa'\)  
return pattern to context: given constraints \(\Phi\), \text{ret.pat} and return type \text{ret} produces contexts \(\mathcal{C}';\mathcal{L}';\Phi';\kappa'\)  

\(\text{expl} \text{ pure} \) 
\(\text{expl} \text{ res} \) 
\(\text{expl} \text{ spine} \)
expl_is_expr ::= 
  | C; L; Φ; R ⊢ action ⇒ ret
  memory action syntheses: given C; L; Φ; R, action synthesises return type ret
  | C; L; Φ; R ⊢ memop ⇒ ret
  memory operation syntheses: given C; L; Φ; R, memop synthesises return type ret
  | C; L; Φ; R ⊢ is_expr ⇒ ret
  indet. seq. expression syntheses: given C; L; Φ; R, is_expr synthesises return type ret

expl_seq_expr ::= 
  | C; L; Φ; R ⊢ seq_expr ⇒ ret
  seq. expression syntheses: given C; L; Φ; R, seq_expr synthesises return type ret

expl_top ::= 
  | C; L; Φ; R ⊢ tval ⇐ ret
  top-level value checks: given C; L; Φ; R, tval checks against return type ret
  | C; L; Φ; R ⊢ seq_texpr ⇐ ret
  top-level seq. expression checks: given C; L; Φ; R, seq_texpr checks against return type ret
  | C; L; Φ; R ⊢ is_texpr ⇐ ret
  top-level indet. seq. expression checks: given C; L; Φ; R, is_texpr checks against return type ret
  | C; L; Φ; R ⊢ texpr ⇐ ret
  top-level expression checks: given C; L; Φ; R, texpr checks against return type ret
B3  Operational Semantics Judgements

\[ \text{subs} \_judge ::= \]
\[ \quad | \ \text{pat} = \text{pval} \sim \sigma \]
\[ \quad \text{computational value deconstruction: pat deconstructs pval to produce substitution } \sigma \]
\[ \quad | \ \text{ident} \_\text{or} \_\text{pat} = \text{pval} \sim \sigma \]
\[ \quad \text{computational value deconstruction: ident_or_pat deconstructs pval to produce substitution } \sigma \]
\[ \quad | \ \langle \text{h}; \text{res} \_\text{pat} = \text{res} \_\text{val} \rangle \sim \langle \text{h}'; \sigma \rangle \]
\[ \quad \text{resource term deconstruction: res_pat deconstructs res_val to produce substitution } \sigma \]
\[ \quad | \ \langle \text{h}; \text{ret} \_\text{pat}_i = \text{ret}_\text{term}_i \rangle \sim \langle \text{h}'; \sigma \rangle \]
\[ \quad \text{return value deconstruction: ret_pat_i deconstructs ret_val_i to produce substitution } \sigma \]
\[ \quad | \ \langle \text{h}; \text{x}_i = \text{spine}_\text{elem}_i \rangle :: \text{fun} \gg \langle \text{h}'; \sigma; \text{ret} \rangle \]
\[ \quad \text{function call spine: heap } \text{h} \text{ and formal parameters } \text{x}_i \text{ assigned to spine_elem_i for function of type fun, produce new heap } \text{h}' \text{ substitution } \sigma \text{ and result type } \text{ret} \]

\[ \text{pure_opsem_defns} ::= \]
\[ \quad | \ \langle \text{pexpr} \rangle \rightarrow \langle \text{texpr} : \text{pure} \_\text{ret} \rangle \]
\[ \quad | \ \langle \text{texpr} \rangle \rightarrow \langle \text{texpr}' \rangle \]

\[ \text{opsem_defns} ::= \]
\[ \quad | \ \langle \text{h}; \text{pred} \_\text{ops} \rangle \downarrow \langle \text{h}'; \text{res} \_\text{val} \rangle \]
\[ \quad \text{big-step resource (q)points-to operation reduction: } \langle \text{h}; \text{pred} \_\text{ops} \rangle \text{ reduces to } \langle \text{h}'; \text{res} \_\text{val} \rangle \]
\[ \quad | \ \text{footprint of res_val in } \text{h} \sim \text{h}_1 \text{ rem } \text{h}_2 \]
\[ \quad \text{footprint of res_val in heap } \text{h} \text{ is } \text{h}_1 \text{ with } \text{h}_2 \text{ remainder/frame} \]
\[ \quad | \ \langle \text{h}; \text{res} \_\text{term} \rangle \downarrow \langle \text{h}'; \text{res} \_\text{val} \rangle \]
\[ \quad \text{big-step resource term reduction: } \langle \text{h}; \text{res} \_\text{term} \rangle \text{ reduces to } \langle \text{h}'; \text{res} \_\text{val} \rangle \]
\[ \quad | \ \langle \text{h}; \text{action} \rangle \rightarrow \langle \text{h}'; \text{is} \_\text{expr} \rangle \]
\[ \quad | \ \langle \text{h}; \text{memop} \rangle \rightarrow \langle \text{h}'; \text{is} \_\text{expr} \rangle \]
\[ \quad | \ \langle \text{h}; \text{is} \_\text{expr} \rangle \rightarrow \langle \text{h}'; \text{is} \_\text{expr}' \rangle \]
\[ \quad | \ \langle \text{h}; \text{seq}_\text{expr} \rangle \rightarrow \langle \text{h}'; \text{texpr} : \text{ret} \rangle \]
\[ \quad | \ \langle \text{h}; \text{seq}_\text{texpr} \rangle \rightarrow \langle \text{h}'; \text{texpr} \rangle \]
\[ \quad | \ \langle \text{h}; \text{is}_\text{texpr} \rangle \rightarrow \langle \text{h}'; \text{texpr} \rangle \]
\[ \quad | \ \langle \text{h}; \text{texpr} \rangle \rightarrow \langle \text{h}'; \text{texpr}' \rangle \]
B4 Proof Judgements

Note that the definition of \( \sim \) is omitted/assumed. It simply means that \( \text{term} \) and \( \text{term}' \) can be unified. Informally, \( \text{term} \sim \text{term}' \) are defined recursively over the structure of SMT terms, using the standard definition of unification: variables unify with anything (modulo an occurs check), atoms unify if they are identical, compound terms unify if their constructors (except for Specified) and arity are identical, and their arguments unify recursively.

To clarify the Specified exception: \( \text{term} \sim \text{Specified}(pval) \) (and \( \text{Specified}(pval) \sim \text{term} \)) iff \( \text{term} \sim pval \).

\( \sim \) is additionally assumed to be an equivalence relation and preserved by substitution: if \( \text{term} \sim \text{term}' \) and \( x \sim y \) in \( \text{term}_1 \sim \text{term}'_1 \) then \( \text{term}/x(\text{term}_1) \sim \text{term}'/y(\text{term}'_1) \).

Note: \( \sim \) is only used in the proof of soundness, and not in the explicit CN type system. There is no unification required in the type system, but the notion of related terms is required to argue for the soundness of pattern-matching (Section B9.4 Well-typed values pattern-match successfully).
misc_extra ::= extra judgements for proof-related definitions
    | ∀ x. iguard ⇒ C; L; Φ ⊢ h ⇐ R
      meta-logical quantification over heap-typing
    | ∀ term ∼ term’. Φ ⊢ fun ∼ ret
      meta-logical quantification over related fun and ret
    | ∀ term ∼ term’. Φ ⊢ res ∼ res’
      meta-logical quantification over related res and res’
    | term ∼ term’
      omitted/assumed defintion: SMT terms term and term’ are related

proof_defs ::= 
    | π_i :: fun ∼ C; Φ; R | ret
      matching π_i and fun produces contexts C; L; Φ; R and return type ret
    | C; L; Φ; R ⊑ C’; L’; Φ’; R’
      context weakening: C; L; Φ; R is stronger than C’; L’; Φ’; R’
    | C; L; Φ; R ⊢ σ ⇐ (C; L; R)
      well-typed substitution: given C; L; Φ; R, σ checks against type (C; L; R). It is complicated by the fact that substitutions are assumed to be sequential/telescoping.
    | C; L; Φ ⊢ h ⇐ R
      heap typing: under context C; L; Φ, heap h checks against context/type R
    | Φ ⊢ h ⇐ R
      heap typing: under context Φ, heap h checks against context/type R
    | Φ ⊢ res ∼ res’
      res is related to res’
    | Φ ⊢ fun ∼ ret
      fun is related to ret
B5 Groups of Rules

B5.1 Typing rules with an \(\text{smt}(\Phi \Rightarrow q\text{term})\) premise


B5.2 Typing rules which change the context

B5.2.1 Rules which add constraints

EXPL_TOP_SEQ_IF.

B5.2.2 Rules which add constraints and computational or logical variables

EXPL_TOP_SEQ_LETP, EXPL_TOP_SEQ_LETP, EXPL_TOP_SEQ_CASE.

B5.2.3 Rules which restrict the resource context

No-resource / “pure” rules: IG_CMP_EQ, IG_CMP_LT, IG_CMP_GT, IG_CMP_NONE, Q_CMP_NAME_NEQ, Q_CMP_PTR_STEP_NEQ, Q_CMP_COMPARABLE, Q_CMP_IARG_NEQ, Q_CMP_COMPARABLE, REQ_EQ_PP_NAME_NEQ, REQ_EQ_PP_IARG_NEQ, REQ_EQ_PP_EQ, REQ_EQ_QQ_EQ, REQ_EQ_QQ_NEQ, RES_EQ_EMP, RES_EQ_PHI, RES_EQ_PRED, RES_EQ_QPRED, RES_EQ_SEPCONJ, RES_EQ_EXISTS, RES_EQ_ORD_DISJ, RES_SIMP_REC_IF_TRUE, RES_SIMP_REC_IF_FALSE, RES_SIMP_REC_EXISTS, RES_SIMP_REC_NOCHANGE, SIMP_NO_SIMP, SIMP_SEP_CONJ, RET_EQ_END, RET_EQ_COMP, RET_EQ_LOG, RET_EQ_PHI, RET_EQ_RES, PAT_COMP_NO_SYM_ANNOT, PAT_COMP_SYM_ANNOT, PAT_COMPNIL, PAT_COMP_CONS, PAT_COMP_TUPLE, PAT_COMPARRAY, PAT_COMP_SPECIFIED, PAT_SYM_OR_PAT_SYM, PAT_SYM_OR_PAT_PAT, PAT_RES_MATCHEMP, PAT_RES_MATCH_PHI, PAT_RES_MATCH_IF_TRUE, PAT_RES_MATCH_IF_FALSE, PAT_RES_MATCH_VAR, PAT_RES_MATCH_SEP_CONJ, PAT_RES_MATCH_PACK, PAT_RES_MATCH_FOLD, PAT_RET_EMPTY, PAT_RET_COMP, PAT_RET_LOG, PAT_RET_RES, PAT_RET_PHI, PAT_RET_AUX, PURE_VAL_OBJ_INT, PURE_VAL_OBJ_PTR, PURE_VAL_OBJ_ARR, PURE_VAL_OBJ_STRUCT, PURE_VAL_VAR, PURE_VAL_OBJ, PURE_VAL_LOADED, PURE_VAL_UNIT, PURE_VAL_TRUE, PURE_VAL_FALSE, PURE_VAL_LIST, PURE_VAL_TUPLE, PURE_VALCTORNIL, PURE_VALCTORCONS, PURE_VALCTOR_TUPLE, PURE_VALCTORARRAY, PURE_VALCTOR_SPECIFIED, PURE_VAL_STRUCT, PURE_EXPR_VAR, PURE_EXPR_ARRAY_SHIFT, PURE_EXPR_MEMBER_SHIFT, PURE_EXPR_NOT, PURE_EXPR_ARITH_BINOP, PURE_EXPR_REL_BINOP, PURE_EXPR_BOOL_BINOP, PURE_EXPR_CALL, PURE_EXPR_ASSERT_UNDEF, PURE_EXPR_BOOL_TO_INTEGER, PURE_EXPR_WrapI, PURE_TOP_VAL_UNDEF, PURE_TOP_VAL_ERROR, PURE_TOP_VAL_DONE.

Resource-mentioning rules: RES_SYN_EMP, RES_SYN_VAR, RES_SYN_VAR_SIMP, RES_SYN_PRED, RES_SYN_Q_PRED, RES_SYN_SEP_CONJ, RES_CHK_PHI, RES_CHK_SEP_CONJ, EXPL_SPIRE_RET,
B5.2.4 Rules which add constraints and restrict the resource context

 Pure_Top_if.

B5.2.5 Rules which add constraints and variables, and restrict the resource context

 Pure_Top_let, Pure_Top_letT, Pure_Top_case, Expl_Top_seq_let, Expl_Top_seq_letT, Expl_Top_is_letS.

B5.3 Value typing rules

 Pure_val_obj_int, Pure_val_obj_ptr, Pure_val_obj_arr, Pure_val_obj_struct,
 Pure_val_var, Pure_val_obj, Pure_val_loaded, Pure_val_unit, Pure_val_true,
 Pure_val_false, Pure_val_tuple, Pure_val_ctor_nil, Pure_val_ctor_cons, Pure_val_ctor_tuple,
 Pure_val_ctor_array, Pure_val_ctor_specified, Pure_val_struct,
 Pure_top_val_done, Res_syn_emp, Res_syn_var, Res_syn_var_simp, Res_syn_pred,
B6  Weakening

If $C; L; \Phi; R \sqsubseteq C'; L'; \Phi'; R'$ and $C; L; \Phi; R \vdash J$ then $C'; L'; \Phi'; R' \vdash J$.

**Assume:** 1. $C; L; \Phi; R \sqsubseteq C'; L'; \Phi'; R'$.
   2. $C; L; \Phi; R \vdash J$

**Prove:** $C'; L'; \Phi'; R' \vdash J$.

(1) 1. **Case:** Pure_Val_Var.

   **Proof:** By Weak_Cons_Comp, if $x: \beta \in C$ then $x: \beta \in C'$.

(1) 2. **Case:** Typing rules with an $\text{smt} (\Phi \Rightarrow \text{qterm})$ premise (see B5.1).

   **Assume:** $\text{smt} (\Phi \Rightarrow \text{qterm})$.
   **Prove:** $\text{smt} (\Phi' \Rightarrow \text{qterm})$.

   (2) 1. For all $\text{term}$, if $\text{term} \in \Phi$ then $\text{term} \in \Phi'$.

   **Proof:** By Weak_Cons_Phi.

   (2) 2. Any extra constraints in $\Phi'$ (by Weak_Skip_Phi) would either be irrelevant, redundant, or inconsistent.

   (2) 3. In all cases, $\text{smt} (\Phi' \Rightarrow \text{qterm})$ as required.

(1) 3. **Case:** All remaining rules.

   (2) 1. $R = R'$.

   **Proof:** Only Weak_Cons_Res exists.

   (2) 2. All remaining rules are functorial in $C; L; \Phi$, so one can proceed by straightforward induction.

   (2) 3. So $C'; L'; \Phi'; R' \vdash J$ as required.
B7 Substitution

B7.1 Substitutions preserve SMT results

If \( \text{smt}(\Phi \Rightarrow q\text{term}) \) and \( C; L; \sigma(\Phi); R \vdash \sigma \Leftarrow (C'; L'; R') \), then \( \text{smt}(\sigma(\Phi) \Rightarrow \sigma(q\text{term})) \).

Proof: By the first assumption, \( q\text{term} \) holds for all (well-typed, ensured by the second assumption) instantiations of its free variables.

B7.2 Substitutions can be split up

If \( C; L; \Phi; R \vdash \sigma \Leftarrow (C'; L'; R') \) then
\[ \exists R_1, R_2, \sigma_1, \sigma_2. C; L; \Phi; R_1 \vdash \sigma_1 \Leftarrow (C'; L'; R'_1) \land C; L; \Phi; R_2 \vdash \sigma_2 \Leftarrow (C'; L'; R'_2). \]

Proof sketch: By induction on the substitution. If \( \sigma = \lbrack \text{res\_term}/r, \sigma' \rbrack \) where \( r:\text{res} \):

1. Let \( R' \) be such that \( C; L; \Phi; R' \vdash \text{res\_term} \Leftarrow \text{res} \).

2. Recursively split \( \sigma' \) into \( \sigma'_1 \) and \( R''_1; \sigma'_2 \) and \( R''_2 \).

3. If \( r \in R'_1 \), let \( \sigma_1 = \lbrack \text{res\_term}/r, \sigma'_1 \rbrack \) and \( R_1 = R', R''_1 \).

4. If \( r \in R'_2 \), let \( \sigma_2 = \lbrack \text{res\_term}/r, \sigma'_2 \rbrack \).

5. For other cases, both are treated exactly the same.

B7.3 Substitution

If \( C'; L'; \Phi; R' \vdash J \), then \( \forall C, L, R, \sigma. (C; L; \sigma(\Phi); R \vdash \sigma \Leftarrow (C'; L'; R')) \Rightarrow C; L; \sigma(\Phi); R \vdash \sigma(J) \).

For types, substitutions are only defined over resource types \( \text{res} \), and return types \( \text{res} \), not base types \( \beta \). Similarly, for terms, substitutions are only defined over expressions (including SMT terms \( \text{term} \)), but not (computational, resource or return) patterns.

Since \( \Phi \) is scoped to \( C'; L' \), we must substitute over it as well as all the usual suspects on the right.

Substitution of contexts is defined by substituting over each constraint in \( \Phi \). As a result, \( \sigma(\Phi_1, \Phi_2) = \sigma(\Phi_1), \sigma(\Phi_2) \), and if \( \sigma(\Phi) = \Phi'_1, \Phi'_2 \) then \( \exists \Phi_1, \Phi_2. \sigma(\Phi_1, \Phi_2) = \sigma(\Phi_1), \sigma(\Phi_2) \).

Proof sketch: Induction over the typing judgements.

1. Variable rules: \text{PURE\_VAL\_VAR}, \text{RES\_SYN\_VAR\_SIMP}, \text{RES\_SYN\_VAR}.

2. \text{EXPL\_TOP\_VAL\_DONE}: prove that \( \text{to.fun} \) commutes with substitution.

3. Typing rules which change the context (see B5.2).


Assume: 1. \( C'; L'; \Phi; R' \vdash J \).

2. Arbitrary \( C, L, R, \sigma. \)

3. \( C; L; \sigma(\Phi); R \vdash \sigma \Leftarrow (C'; L'; R') \).
PROVE: $C; L; \sigma(\Phi); R \vdash \sigma(J)$.

(1)1. CASE: Pure_Val_Var.

\[ C'; L' \vdash x \Rightarrow \beta \]
\[ C; L; \sigma(\Phi); R \vdash \sigma \Leftarrow (C' ; L' ; \cdot) . \]

(2)1. $x: \beta \in C'$. 

PROOF: By inversion on assumption 1.

(2)2. $R$ is empty.

PROOF: SUBS_CHK_Res is the only rule which could require a non-empty resource context, and it is never used because $R'$ is empty.

(2)3. $\exists \sigma_1,\text{pval},\sigma_2,\beta,C_1,C_2,L_1,L_2$.

1. $\sigma = [\sigma_1,\text{pval}/x,\sigma_2]$
2. $C; L; \Phi; \cdot \vdash \sigma_1 \Leftarrow (C_1; L_1; \cdot)$
3. $C; L; \Phi; \cdot \vdash \sigma_1(\text{pval}/x) \Leftarrow (x; \beta; \cdot; \cdot)$
4. $C \vdash \sigma_1(\text{pval}) \Rightarrow \beta$
5. $C; L; \Phi; \cdot \vdash \sigma_1(\text{pval}/x(\sigma_2)) \Leftarrow (C_2; L_2; \cdot)$.

PROOF: Repeated inversion on assumption 3 until the SUBS_CHK_COMP responsible for adding $x$ (by (2)1, there must be at least one).

(2)4. Since $\sigma(x) = \sigma_1(\text{pval})$, we are done.

PROOF: By $C; L \vdash \sigma(x) \Rightarrow \beta$.

(1)2. CASE: Res_Syn_VarSimp.

\[ C'; L'; \Phi; r: \text{res} \vdash r \Rightarrow \text{res}' \]
\[ C; L; \sigma(\Phi); R \vdash \sigma \Leftarrow (C' ; L' ; r: \text{res}') . \]

(2)1. $\exists \sigma_1,\text{res_term},\sigma_2,C_1,C_2,L_1,L_2$.

1. $\sigma = [\sigma_1,\text{res_term}/r,\sigma_2]$
2. $C; L; \sigma(\Phi); \cdot \vdash \sigma_1 \Leftarrow (C_1; L_1; \cdot)$
3. $\cdot; \cdot; \sigma(\Phi); R \vdash \sigma_1(\text{res_term}/r) \Leftarrow (\cdot; \cdot; r; \sigma_1(\text{res}') )$
4. $C; L; \sigma(\Phi); R \vdash \sigma_1(\text{res_term}) \Leftarrow \sigma_1(\text{res}')$
5. $C; L; \sigma(\Phi); \cdot \vdash \sigma_1(\text{res_term}/r(\sigma_2)) \Leftarrow (C_2; L_2; \cdot)$.

PROOF: Repeated inversion on assumption 3 until the SUBS_CHK_Res responsible for adding $r$ (there must be exactly one).

(2)2. Suffices: 1. $\sigma(r) = \sigma_1(\text{res_term})$

2. $\sigma(\text{res}') = [\sigma_1,\text{res_term}/r,\sigma_2](\text{res}') = \sigma_1(\text{res}')$.

(2)3. $\sigma(r) = \sigma_1(\text{res_term})$.

PROOF: $\sigma_2(\cdot) = r$, because $\sigma_2$ does not mention any resource variables.

(2)4. $\sigma(\text{res}') = [\sigma_1,\text{res_term}/r,\sigma_2](\text{res}') = \sigma_1(\text{res}')$.

(3)1. $[\sigma_1,\text{res_term}/r,\sigma_2](\text{res}') = [\sigma_1,\sigma_2](\text{res}')$.

PROOF: Resource types do not refer to resource variables.
\(3\). \([\sigma_1, \sigma_2](res') = \sigma_1(res)\).

**Proof:** By \(\vdash \sigma(\Phi); \mathcal{R} \vdash \sigma_1(res_{\text{term}}) \iff \sigma_1(res')\), we know that \(res'\) only refers to variables in \(\mathcal{C}, \mathcal{C}_1; \mathcal{L}, \mathcal{L}_1\).

\(1.\) **Case:** \texttt{Res_Syn_Var}.
\(\mathcal{C}'; \mathcal{L}'; \Phi; r: res' \vdash r \Rightarrow res\)

**Proof:** Similar to \texttt{Res_Syn_VarSimp}, but with \(res' = res\).

\(1.\) **Case:** \texttt{Expl_Top_Val.Done}.

**Proof sketch:** \texttt{to_fun} recursively maps \(\Sigma\) to \(\Pi\), \(\exists\) to \(\forall\), \(\land\) to \(\lor\) and \(*\) to \(\Rightarrow\), and otherwise keeps any \texttt{term} and \(res\) the same. Hence, \(\sigma(\texttt{to_fun ret}) = \texttt{to_fun} \sigma(\texttt{ret})\), and the case proceeds by induction straightforwardly.

\(1.\) **Case:** Typing rules which change the context (see B5.2), except for \texttt{Pure_Val_Var}, \texttt{Res_Syn_Var}, and \texttt{Res_Syn_VarSimp}.

For brevity, I shall only go over \texttt{Expl_Top_Seq_Let}, as it is one of the most general rules; one which adds constraints and variables, and restricts the resource context.

**Proof sketch:** The key idea is to apply lemma B7.2 (Substitutions can be split up) as required by the restrictions on the resource context. If a rule has a \texttt{smt} (\(\Phi \Rightarrow \texttt{qterm}\)) premise, then apply lemma B7.1 (Substitutions preserve SMT results).

\(2.\) 1. \(\exists \texttt{ret}_1, \mathcal{C}_3, \mathcal{L}_3, \Phi_3, \mathcal{R}_3\).
   1. \(\mathcal{C}'; \mathcal{L}'; \Phi; \mathcal{R}_1', \mathcal{R}_2' \vdash \texttt{let ret\_pat = seq\_expr in} \texttt{texpr} \iff \texttt{ret}_2\)
   2. \(\mathcal{C}; \mathcal{L}; \sigma(\Phi); \mathcal{R} \vdash \sigma \leftarrow (\mathcal{C}'; \mathcal{L}'; \mathcal{R}_1', \mathcal{R}_2')\).
   3. \(\mathcal{C}', \mathcal{C}_3; \mathcal{L}_3, \Phi, \mathcal{R}_3, \mathcal{R}_2', \mathcal{R}_3 \vdash \texttt{texpr} \iff \texttt{ret}_2\).

**Proof:** Inversion on assumption 1.

\(2.\) 1. \(\forall \mathcal{C}, \mathcal{L}, \mathcal{R}_1, \sigma_1\).
   1. \(\mathcal{C}; \mathcal{L}; \sigma_1(\Phi); \mathcal{R}_1 \vdash \sigma_1 \iff (\mathcal{C}'; \mathcal{L}'; \mathcal{R}_1')\) \(\Rightarrow \)
   2. \(\mathcal{C}; \mathcal{L}; \sigma_1(\Phi); \mathcal{R}_1 \vdash \sigma_1(\texttt{seq\_expr}) \Rightarrow \texttt{ret}_1\).
   2. \(\forall \mathcal{C}, \mathcal{L}, \mathcal{R}_2, \sigma_2\).
   1. \(\mathcal{C}; \mathcal{L}; \sigma_2(\Phi); \mathcal{R}_2 \vdash \sigma_2 \iff (\mathcal{C}'; \mathcal{L}'; \cdot)\) \(\Rightarrow \)
   2. \(\mathcal{C}; \mathcal{L}; \sigma_2(\Phi); \mathcal{R}_2 \vdash \sigma_2(\texttt{ret}_1) \Rightarrow \texttt{ret}_2\).
   3. \(\forall \mathcal{C}, \mathcal{L}, \mathcal{R}_3, \sigma_3\).
   1. \(\mathcal{C}; \mathcal{L}; \sigma_3(\Phi, \Phi_3); \mathcal{R}_3 \vdash \sigma_3 \iff (\mathcal{C}', \mathcal{C}_3; \mathcal{L}', \mathcal{L}_3; \mathcal{R}_2', \mathcal{R}_3)\) \(\Rightarrow \)
   2. \(\mathcal{C}; \mathcal{L}; \sigma(\Phi, \Phi_3); \mathcal{R}_2 \vdash \sigma(\texttt{texpr}) \iff \texttt{ret}_2\).

**Proof:** By induction on (2)1.

\(2.\) 3. \(\sigma\) and \(\mathcal{R}\) can be split up into \(\sigma_1\) and \(\mathcal{R}_1; \sigma_2;\) and \(\sigma_3\) and \(\mathcal{R}_2\) such that:
   1. \(\mathcal{R} = \mathcal{R}_1, \mathcal{R}_2\)
   2. \(\mathcal{C}; \mathcal{L}; \sigma(\Phi); \mathcal{R}_1 \vdash \sigma_1 \iff (\mathcal{C}'; \mathcal{L}'; \mathcal{R}_1')\)
   3. \(\mathcal{C}; \mathcal{L}; \sigma(\Phi); \mathcal{R}_2 \vdash \sigma_2 \iff (\mathcal{C}'; \mathcal{L}'; \cdot)\)
   4. \(\mathcal{C}; \mathcal{L}; \sigma(\Phi); \mathcal{R}_3 \vdash \sigma_3 \iff (\mathcal{C}'; \mathcal{L}'; \mathcal{R}_3')\).
(2.4) 1. \( \sigma(\Phi) = \sigma_1(\Phi) = \sigma_2(\Phi) = \sigma_3(\Phi) \)
2. \( \sigma(\Phi_3) = \sigma_2(\Phi_3) = \sigma_3(\Phi_3) \)
3. \( \sigma(R_3) = \sigma_2(R_3) = \sigma_3(R_3) \).

Proof: All the substitutions differ only the resource-variable substitutions, but \textit{term} and \textit{res} (and so \textit{ret} and \Phi) do not mention resource variables.

(2.5) Suffices: \( \exists R_1, R_2, \text{ret}_1, C_3, L_3, \Phi_3, R_3 \).
1. \( R = R_1, R_2 \)
2. \( C; L; \sigma(\Phi); R_1 \vdash \sigma(\text{seq_expr}) \Rightarrow \text{ret}_1 \)
3. \( \sigma(\Phi) \vdash \text{ret\_pat}:\sigma(\text{ret}_1) \leadsto C_3; L_3; \Phi_3; R_3 \)
4. \( C, C_3; L, L_3; \sigma(\Phi), \Phi_3; R_2, R_3 \vdash \sigma(\text{texpr}) \Leftarrow \sigma(\text{ret}_2) \).

Proof: By \textsc{Expl\_Top\_Seq\_Let}.

(2.6) Let: \( R_1; R_2; \sigma(\text{ret}_1); C_3; L_3; \sigma(\Phi_3); \sigma(R_3) \) be the witnesses for (2.5).

Suffices: 1. \( R = R_1, R_2 \)
2. \( C; L; \sigma(\Phi); R_1 \vdash \sigma(\text{seq_expr}) \Rightarrow \sigma(\text{ret}_1) \)
3. \( \sigma(\Phi) \vdash \text{ret\_pat}:\sigma(\text{ret}_1) \leadsto C_3; L_3; \sigma(\Phi_3); \sigma(R_3) \)
4. \( C, C_3; L, L_3; \sigma(\Phi), \sigma(\Phi_3); R_2, \sigma(R_3) \vdash \sigma(\text{texpr}) \Leftarrow \sigma(\text{ret}_2) \).

(2.7) We are done.

Proof: Apply (2.2) with (2.3) and (2.4).

(1.6) Case: All remaining rules.

Proof sketch: By straightforward induction. If the rule has a \texttt{smt}(\Phi \Rightarrow qterm) premise, apply lemma B7.1 (Substitutions preserve SMT results).
B8 Resource Term Lemmas

B8.1 Definition: Normalised contexts

A resource context is *normalised* if it contains only predicates, quantified predicates and under-determined conditional resources.

B8.2 Resource contexts typing closed terms must be normalised

**Assume:**
1. Arbitrary res  
2. Closed (no free-variables) res_term  
3. ·; ·; Φ; R ⊢ res_term ⇐ res (or synthesising)

**Prove:** ∃R. R = R.

**Proof sketch:** By induction on the typing judgement.

1.1. **Case:** Res_Syn_Emp, Res_Syn_Pred, Res_Syn_QPred, Res_chk_Phi.  
   **Proof:** R = R (the context is already normalised).

1.2. **Case:** Res_Syn_Var, Res_Syn_VarSimp  
   **Proof:** Impossible cases (res_terms are not closed).

   **Proof:** By induction.

B8.3 Non-conditional resources determine context and values

This is a simple inversion lemma.

**Assume:**
1. Arbitrary res_val  
2. res ≠ if term then res₁ else res₂.  
3. ·; ·; Φ; R ⊢ res_val ⇐ res (or synthesising)

   **Proof:** res_terms in these rules are not values.

1.2. If res = emp, then R = · and res_val = emp.  
   **Proof:** By inversion, the assumption must be Res_Syn_Emp (and optionally Res_chk_
Switch).

(1.3) If \( \text{res} = \text{term} \), then \( \mathcal{R} = \cdot \) and \( \text{res\_val} = \text{term} \).
Proof: By inversion, the assumption must be \( \text{RES\_CHK\_PHI} \).

(1.4) If \( \text{res} = \text{pred\_term}(\text{oarg}) \), then \( \mathcal{R} = \cdot \), \( \text{res\_val} = \text{pred\_term} \).
Proof: By inversion, the assumption must be \( \text{RES\_SYN\_PRED} \) (and optionally \( \text{RES\_CHK\_SWITCH} \)).

(1.5) If \( \text{res} = \text{qpred\_term}(\text{oarg}) \), then \( \mathcal{R} = \cdot \), \( \text{res\_val} = \text{qpred\_term} \).
Proof: By inversion, the assumption must be \( \text{RES\_SYN\_QPRED} \) (and optionally \( \text{RES\_CHK\_SWITCH} \)).

(1.6) If \( \text{res} = \text{res}_1 \ast \text{res}_2 \), then \( \mathcal{R} = \mathcal{R}_1, \mathcal{R}_2 \) and \( \text{res\_val} = \langle \text{res\_val}_1, \text{res\_val}_2 \rangle \).
Proof: By inversion, the assumption must be \( \text{RES\_SYN\_SEPCONJ} \) (and optionally \( \text{RES\_CHK\_SEPCONJ} \)).

B8.4 Normalised resource context determines structure of heap

This is as simple inversion lemma.

Assume: \( \Phi \vdash h \leftrightarrow \mathcal{R} \).

(1) If \( \mathcal{R} = \cdot \), then \( h = \cdot \).
Proof: By inversion, the assumption must be \( \text{HEAP\_EMPTY} \).

(2) If \( \text{pred} = \text{ptr} \rightarrow^\text{init} \text{value} \), \( \mathcal{R} = \cdot \), then \( h = \{ \text{pred}' \& \text{None} \} \) for \( \Phi \vdash \text{pred} \equiv \text{pred}' \).
Proof: By inversion, the assumption must be \( \text{HEAP\_PRED\_OWNED} \).

(3) If \( \mathcal{R} = \cdot \), then \( h = \{ \text{pred}' \& \text{def} \& h' \} \).
for \( \Phi \vdash \text{pred} \equiv \text{pred}' \).
Proof: By inversion, the assumption must be \( \text{HEAP\_PRED\_OTHER} \).

(4) If \( \text{qpred} = \# x. \text{iguard} \Rightarrow \text{ptr} + x \times \text{size\_of}(\tau) \rightarrow^\text{oarg[x].init} \text{oarg[x].value} \), \( \mathcal{R} = \cdot \), then \( h = \{ \text{qpred}' \& \cdot \} \) for \( \Phi \vdash \text{qpred} \equiv \text{qpred}' \).
Proof: By inversion, the assumption must be \( \text{HEAP\_QPRED\_OWNED} \).

(5) If \( \mathcal{R} = \cdot \), then \( h = \{ \text{qpred}' \& \text{arr\_def\_heap} \} \) for \( \Phi \vdash \text{qpred} \equiv \text{qpred}' \).
Proof: By inversion, the assumption must be \( \text{HEAP\_QPRED\_OTHER} \).

(6) If \( \mathcal{R} = \mathcal{R}_1, \mathcal{R}_2 \), then \( h = h_1 + h_2 \), where \( \Phi \vdash h_1 \leftrightarrow \mathcal{R}_1 \) and \( \Phi \vdash h_2 \leftrightarrow \mathcal{R}_2 \).
Proof: By inversion, the assumption must be \( \text{HEAP\_CONCAT} \).
B8.5 Well-typed resource value determines its footprint

ASSUME: · · Φ; R ⊢ res_val ⇐ res (or synthesising)
         Φ ⊢ h ⇐ R.
PROVE: ∀ f. footprint_of res_val in h + f ⇝ h rem f.

PROOF SKETCH: By induction on the typing judgement.

⟨1⟩1. CASE: Res_Syn_Emp or Res_CHK_Phi
   R = · and so h = · by lemma B8.4.
   PROOF: Footprint_Emp or Footprint_TERM respectively.

⟨1⟩2. CASE: Res_Syn_PRED or Res_SYN_QPred
   R = prod_term(oarg) or qprod_term(oarg), and so
   h = {prod_term(oarg) & opt_def_heap} or {qprod_term(oarg) & arr_def_heap} by
   lemma B8.4.
   PROOF: Footprint_PRED or Footprint_QPred respectively.

⟨1⟩3. CASE: pack(oarg, res_val').
   PROOF: By induction.

⟨1⟩4. CASE: Footprint_SepPair.
   res_val = ⟨res_val1, res_val2⟩,
   R = R1, R2, and so
   h = h1 + h2 where Φ ⊢ h1 ⇐ R1 and Φ ⊢ h2 ⇐ R2 by lemma B8.4.
   ⟨2⟩1. footprint_of res_val1 in h1 + h2 + f ⇝ h1 rem h2 + f.
       PROOF: Instantiate inductive hypothesis with h2 + f.
   ⟨2⟩2. footprint_of res_val1 in h2 + f ⇝ h2 rem f.
       PROOF: Instantiate inductive hypothesis with f.

B8.6 Progress and type preservation for resource terms

ASSUME: 1. Closed (no free-variables) res_term
         2. · · Φ; R ⊢ res_term ⇐ res (or synthesising)
         3. Φ ⊢ h ⇐ R

PROVE: ∃ res_val, R', h'.
1. · · Φ; R' ⊢ res_val ⇐ res (or synthesising respectively)
2. Φ ⊢ h' ⇐ R'
3. ∀ f. (h + f; res_term) ↓ (h' + f; res_val).

PROOF SKETCH: Induction on the resource term typing assumption. The type dictates the value
and context, the latter of which dictates the shape of the heap.

Because of this direction of information, you cannot prove that
∀\mathcal{R}' . (Φ ⊢ h' ⇐ \mathcal{R}') ⇒ (\cdots ; \mathcal{R}' \vdash res\_val' ⇐ res) . The converse is already true by the composition of lemmas B8.3 and B8.4. You need the existential, so that you can provide it as a witness when proving heap typing for folded predicates, which you need to use in proving unfolding predicates in pattern-matching.

(1.1) **CASE: Res_SynPredOps_Iterate**

**LET:** res\_term = iterate(res\_term', n)
qpred\_term = (x; 0 ≤ x ∧ x ≤ n − 1){Owned(τ)(ptr + x \times size_{o}(τ))}
res = qpred\_term(oarg)
pred\_term = Owned(array n τ)(ptr)
res' = pred\_term(oarg').

(2.1) \(\cdots ; Φ; \mathcal{R} \vdash res\_term' \Rightarrow res'.\)
**PROOF:** By inversion on the typing assumption.

(2.2) \(∃h'', \mathcal{R}'', res\_val'.\)
1. \(\cdots ; \mathcal{R}'' \vdash res\_val' \Rightarrow res'.\)
2. \(Φ \vdash h'' \Leftarrow \mathcal{R}''\)
3. \(∀f . (h + f; res\_term') \Downarrow (h'' + f; res\_val').\)
**PROOF:** By the induction hypothesis.

(2.3) res\_val' = pred\_term and \(\mathcal{R}'' = \_;res'.\)
**PROOF:** By (2.2) and lemma B8.3 (Non-conditional resources determine context and values).

(2.4) h'' = \{pred\_term(oarg')\} & None.
**PROOF:** By (2.3) and lemma B8.4 (Normalised resource context determines structure of heap).

(2.5) **LET:** res\_val = (x; 0 ≤ x ∧ x ≤ n − 1){Owned(τ)(ptr + x \times size_{o}(τ))}
\(\mathcal{R}' = \_;qpred\_term(oarg)\) and \(h' = \{qpred\_term(oarg)\} & \_.\)
**PROOF:** Prove value typing using Res_SynQPred; heap typing using Heap_QPred_Owned; reduction using PredOps_ResV_Iterate.

(1.2) **CASE: Res_SynPredOps_Congeal**
**PROOF:** Like Res_SynPredOps_Iterate, but with:
res\_term = congeal(res\_term', n)
res = pred\_term(oarg) where pred\_term = Owned(array n τ)(ptr)
res' = qpred\_term(oarg) where qpred\_term = (x; iqguard){Owned(τ)(ptr + x \times size_{o}(τ))}
res\_val' = qpred\_term and \(\mathcal{R}' = \_;res',\) by lemma B8.3

Let res\_val = pred\_term, \(\mathcal{R}' = \_;qpred\_term(oarg)\) and \(h' = \{qpred\_term(oarg)\} & None\} to prove: value typing using Res_Syn_Pred; heap typing using Heap_Pred_Owned; reduction using PredOps_ResV_Congeal.

(1.3) **CASE: Res_SynPredOps_Explode**
**PROOF:** Like Res_SynPredOps_Iterate, but with:
res\_term = explode(res\_term')
res = * (pred_term_i(oarg_i))^i where pred_term_i = Owned (τ_i)(ptr + ptr offset_of_tag(member_i))

res' = pred_term(oarg) where pred_term = Owned (struct tag)(ptr)
res_val' = pred_term and R'' = _pred_term(oarg), by lemma B8.3

Let res_val = ⟨pred_term_i^i⟩, R' = _pred_term(oarg_i)^i and
h' = ⟨pred_term_i(oarg_i) & None⟩^i, to prove: value typing using Res_Syn_Pred and Res_Syn_SepConj; heap typing using Heap_Concat and Heap_Pred_Owned; reduction using PredOps_ResV_Explode.

(1.4) CASE: Res_Syn_PredOps_Explode
Proof: Like Res_Syn_PredOps_Iterate, but with:
res_term = implode (res_term', tag)
res = pred_term(oarg) where pred_term = Owned (struct tag)(ptr)
res_term' = * (pred_term_i(oarg_i))^i where pred_term_i = Owned (τ_i)(ptr + ptr offset_of_tag(member_i))
res_val' = pred_term_i^i and R'' = _pred_term(oarg_i)^i, by lemma B8.3

Let res_val = Owned (struct tag)(ptr), R' = _pred_term(oarg), and
h' = ⟨pred_term(oarg) & None⟩, to prove: value typing using Res_Syn_Pred; heap typing using Heap_Pred_Owned; reduction using PredOps_ResV_Explode.

(1.5) CASE: Res_Syn_PredOps_Break
Proof: Like Res_Syn_PredOps_Iterate, but with:
res_term = break (res_term', term)
res = qpred_term(oarg) * pred_term(oarg[term]) where
qpred_term = (x; iguard ∧ (x ≠ term)){α(ptr + x×step, iargs)} and
pred_term = α(ptr + (term × step), term/x(iargs))

res' = qpred_term'(oarg) where qpred_term' = (x; iguard){α(ptr + x×step, iargs)}
res_val' = qpred_term', and R'' = _qpred_term'(oarg), by lemma B8.3.

If predicate is Owned (τ), h'' = {qpred_term'(oarg) & ·} (by lemma B8.4), so let
h' = {qpred_term(oarg) & ·} + {pred_term(oarg[term]) & None} (by [term] = None).
Otherwise, h'' = {qpred_term(oarg) & arr_def_heap}, (again by lemma B8.4), so let
h' = {qpred_term(oarg) & arr_def_heap} + {pred_term(oarg[term]) & arr_def_heap[term]}.

Let res_val = {qpred_term, pred_term} and
R' = _qpred_term(oarg), _pred_term(oarg[term]) to prove: value typing using Res_Syn_QPred, Res_Syn_Pred, Res_Syn_SepConj; heap typing using Heap_Concat, Heap_QPred_Owned / Heap_QPred_Other, and Heap_Pred_Owned / Heap_Pred_Other (with witness _pred_term(oarg[term])); reduction using PredOps_ResV_Break.

(1.6) CASE: Res_Syn_PredOps_Glue
Proof: Like Res_Syn_PredOps_Iterate, but with:
res_term = glue (res_term')

21
\[ res = qpred\_term(oarg_1[\text{term}] := oarg_2) \text{ where} \\
qpred\_term = (x; iguard \lor x = \text{term})\{\alpha(ptr_1 + x \times \text{step}, i\text{arg}_1^i)\} \]

\[ res' = qpred\_term_1(oarg_1) \ast qpred\_term_2(oarg_2) \text{ where} \\
qpred\_term_1 = (x; iguard)\{\alpha(ptr_1 + x \times \text{step}, i\text{arg}_1^i)\} \text{ and } qpred\_term_2 = \alpha(ptr_2, i\text{arg}_2^i). \\
res\_val' = (qpred\_term_1, qpred\_term_2), \text{ and } R'' = \_\_qpred\_term_1(oarg_1), \_\_qpred\_term_2(oarg_2), \text{ by lemma B8.3.} \]

If predicate is \textbf{Owned}\(\tau\), \(h'' = \{qpred\_term_1(oarg_1) & \cdot\} + \{qpred\_term_2(oarg_2) & \textbf{None}\} \text{ (by lemma B8.4), so let } h' = \{qpred\_term(oarg) & \cdot\} \text{ (by \{\text{term}\} := \textbf{None} = \cdot). Otherwise,} \]

\[ h'' = \{qpred\_term_1(oarg_1) & \text{arr\_def\_heap}\} + \{qpred\_term_2(oarg_2) & \text{def} & \text{heap}\} \text{ (again by lemma B8.4), so let } h' = \{qpred\_term(oarg) & \text{arr\_def\_heap}[\text{term}] := \text{def} & \text{heap}\}. \]

Let \(res\_val = qpred\_term\) and \(R = \_\_qpred\_term(oarg_1[\text{term}] := oarg_2)\), to prove: value typing using \textbf{Res\_Syn\_QPred}; heap typing using \textbf{Heap\_QPred\_Owned} / \textbf{Heap\_QPred\_Other}; reduction using \textbf{PredOps\_ResV\_Glue}.

\textbf{(1.7). CASE: Res\_Syn\_PredOps\_Inj} \\
\textbf{PROOF:} Like Res\_Syn\_PredOps\_Iterate, but with: \\
\(res\_term = \text{inj}(res\_term', ptr_1, \text{step}, x, i\text{arg}_1^i)\) \\
\[ res = qpred\_term((\text{default array } \beta)[\text{term}] := oarg) \text{ where} \\
qpred\_term = (x; x = \text{term})\{\alpha(ptr_1 + x \times \text{step}, i\text{arg}_1^i)\} \]

\[ res' = qpred\_term(oarg) \text{ where } qpred\_term = \alpha(ptr_2, i\text{arg}_2^i) \]

\[ res\_val' = qpred\_term, \text{ and } R'' = \_\_qpred\_term(oarg), \text{ by lemma B8.3.} \]

If predicate is \textbf{Owned}\(\tau\), \(h'' = \{qpred\_term(oarg) & \textbf{None}\} \text{ (by lemma B8.4), so let } h' = \{qpred\_term((\text{default array } \beta)[\text{term}] := oarg) & \cdot\} \text{ (by \{\text{term}\} := \textbf{None} = \cdot). Otherwise,} \]

\[ h'' = \{qpred\_term(oarg) & \text{def} & \text{heap}\} \text{ (again by lemma B8.4), so let } h' = \{qpred\_term((\text{default array } \beta)[\text{term}] := oarg) & \{\text{term}\} := \text{def} & \text{heap}\}. \]

Let \(res\_val = qpred\_term\) and \(R = \_\_qpred\_term((\text{default array } \beta)[\text{term}] := oarg)\), to prove typing using \textbf{Heap\_QPred\_Owned} / \textbf{Heap\_QPred\_Other}, and reduction using \textbf{PredOps\_ResV\_Inj}.

\textbf{(1.8). CASE: Res\_Syn\_PredOps\_Split} \\
\textbf{PROOF:} Like Res\_Syn\_PredOps\_Iterate, but with: \\
\(res\_term = \text{split}(res\_term', iguard)\) \\
\[ res = qpred\_term_1(oarg) \ast qpred\_term_2(oarg) \text{ where} \\
qpred\_term_1 = (x; iguard)\{\alpha(ptr_1 + x \times \text{step}, i\text{args}\}) \text{ and} \\
qpred\_term_2 = (x; iguard_2)\{\alpha(ptr_1 + x \times \text{step}, i\text{args}\}) \]

\[ res' = qpred\_term(oarg) \text{ where } qpred\_term = (x; iguard')\{\alpha(ptr_1 + x \times \text{step}, i\text{args}\}) \]

\[ res\_val' = qpred\_term, \text{ and } R'' = \_\_qpred\_term(oarg), \text{ by lemma B8.3.} \]

If predicate is \textbf{Owned}\(\tau\), \(h'' = \{qpred\_term(oarg) & \cdot\} \text{ (by lemma B8.4), so let } h' = \{qpred\_term_1(oarg) & \cdot\} + \{qpred\_term_2(oarg) & \cdot\}. Otherwise,}
\[ h'' = \{ qpred\_term(oarg) \& arr\_def\_heap \} \] (again by lemma B8.4), so let
\[ h' = \{ qpred\_term_1(oarg) \& arr\_def\_heap \} + \{ qpred\_term_2(oarg) \& arr\_def\_heap \} \]

Let \( \text{res\_val} = \{\text{qpred\_term}_1, \text{qpred\_term}_2\} \), and
\( R = \{ qpred\_term_1(oarg), qpred\_term_2(oarg) \} \), to prove: value typing using \text{RES\_SYN\_QPRED} and \text{RES\_SYN\_SEP\_CONJ}; heap typing using \text{HEAP\_CONCAT} and \text{HEAP\_QPRED\_OWNED} / \text{HEAP\_QPRED\_OTHER}; reduction using \text{PREDOPS\_RES\_V\_SEP\_PAIR}.

(1.9) CASE: \text{RES\_SYN\_EMP}, \text{RES\_SYN\_PRED}, \text{RES\_SYN\_QPRED}, \text{RES\_CHK\_PHI}.
PROOF: In these cases, \( h = h' \), \( R = R' \) and \( \text{res\_term} = \text{res\_val} \).
Typing holds by assumption; prove reduction using \text{REST\_RES\_V\_SEP\_PAIR}.

(1.10) CASE: \text{RES\_SYN\_PREDOPS}
PROOF: Both typing and reduction (using \text{REST\_RES\_V\_PREDOPS}) hold by induction.

(1.11) CASE: \text{RES\_SYN\_SEP\_CONJ}, \text{RES\_CHK\_SEP\_CONJ}.
\[ \text{res} = \text{res}_1 \ast \text{res}_2, \]
\[ \text{res\_term} = \langle \text{res\_term}_1, \text{res\_term}_2 \rangle, \]
\[ h = h_1 + h_2, \text{ so } R = R_1, R_2, \]
\[ \Phi \vdash h_1 \iff R_1 \text{ and } \Phi \vdash h_2 \iff R_2. \]

(2.1) \( \exists h_1', R'_1, \text{res\_val}_1 \ldots \land (\forall f_1 \ldots) \)
\[ \exists h_2', R'_2, \text{res\_val}_2 \ldots \land (\forall f_2 \ldots) \]
PROOF: By induction.

(2.2) \( \langle h_1 + h_2 + f; \text{res\_term}_1 \rangle \triangleright \langle h_1' + h_2 + f; \text{res\_val}_1 \rangle. \)
\( \langle h_1' + h_2 + f; \text{res\_term}_2 \rangle \triangleright \langle h_1' + h_2 + f; \text{res\_val}_2 \rangle. \)
PROOF: Instantiate \( f_1 \) with \( h_2 + f \), and \( f_2 \) with, \( h_1' + f \).

(2.3) LET: \( \text{res\_val} = \langle \text{res\_val}_1, \text{res\_val}_2 \rangle, R' = R'_1, R'_2, \text{ and } h' = h_1' + h_2'. \)
Prove value typing using \text{RES\_SYN\_SEP\_CONJ} / \text{RES\_CHK\_SEP\_CONJ}; heap typing using \text{HEAP\_CONCAT}; reduction using (2.2) and \text{REST\_RES\_V\_SEP\_PAIR}.

(1.12) CASE: \text{RES\_CHK\_PACK}
PROOF: Like \text{RES\_SYN\_PREDOPS\_ITERATE}, but with:
\[ \text{res\_term} = \text{pack}(oarg, \text{res\_term}'_1), \text{ res} = \exists y : \beta. \text{ res}'' , \text{ res}' = oarg / y(\text{res}'') \]
\[ \text{res\_val} = \text{pack}(oarg, \text{res\_val}') \]. Value and heap typing hold by induction; prove reduction using \text{REST\_RES\_V\_PACK}.

(1.13) CASE: \text{RES\_SYN\_FOLD}
PROOF: Like \text{RES\_SYN\_PREDOPS\_ITERATE}, but with:
\[ \alpha \equiv x_p : \ldots, x_i : x_i', y : \ldots \rightarrow \text{res}'' \in \text{Globals} \]
\[ \text{res\_term} = \text{fold res\_term}'_\alpha(\text{ptr}, \text{iar}_i)'(oarg) \]
\[ \text{res} = \alpha(\text{ptr}, \text{iar}_i)'(oarg) \]
\[ \text{res}' = [oarg / y, \text{iar}_i / x_i', \text{ptr} / x_p](\text{res}''). \]
\[ \exists h_1, R', \text{res\_val}'. \]
1. \( \vdash \Phi; R' \vdash \text{res\_val}' \iff \text{res}' \).
2. $\Phi \vdash h_1 \Leftarrow R'$
3. $\forall f. \langle h + f; \text{res\_term} \rangle \Downarrow \langle h_1 + f; \text{res\_val}' \rangle$
(by induction).

Let $\text{res\_val} = \alpha(ptr, iarg, i), \; R' = \vdash \alpha(ptr, iarg, i)(oarg)$ and
$h' = \{\alpha(ptr, iarg, i)(oarg) \& \text{res\_val}' \& h_1\}$, to prove: value typing using \text{Res\_Syn\_Pred};
heap typing using \text{Heap\_Pred\_Other}.
Since footprint of $\text{res\_val}'$ in $h_1 + f \leadsto h_1$ rem $f$ by lemma B8.5 (Well-typed resource value determines its footprint), prove reduction using \text{ResT\_ResV\_Fold}.

(1)14. \textbf{Case: Res\_Chk\_If\_True, Res\_Chk\_If\_False}
\textbf{Proof:} By induction with $res'$ as $res_1$ or $res_2$ respectively. This is exhaustive because only variables can synthesise under-determined conditional resources and those are excluded by assumption of $\text{res\_term}$ being closed.

(1)15. \textbf{Case: Res\_Chk\_Switch}
\textbf{Proof:} By induction on the synthesising judgement.

\textbf{B8.7 Resource term reduction is deterministic}

\textbf{Proof sketch:} Induction over the definition: it is syntax directed.

\textbf{B8.8 Resource term reduction is isolated}

If $\text{res\_term}$ is closed, $\vdash \cdot; \cdot; \Phi; R \vdash \text{res\_term} \Leftarrow res \; \Phi \vdash h \Leftarrow R$ and $\langle h + f; \text{res\_term} \rangle \Downarrow \langle \text{heap}; \text{res\_val} \rangle$
then $\exists h', R', \text{heap} = h' + f \land (\Phi \vdash h' \Leftarrow R') \land (\cdot; \cdot; \Phi; R' \vdash \text{res\_val} \Leftarrow res)$.

\textbf{Proof:} This simply the composition of lemma B8.7 (Resource term reduction is deterministic) and lemma B8.6 (Progress and type preservation for resource terms).
B9  Progress

B9.1  \( \Phi \vdash res \sim res' \) is an equivalence relation

Proof sketch: By induction and term \( \sim \) term' assumed to be an equivalence relation (see section B4 Proof Judgements).

B9.2  \( \Phi \vdash res \sim res' \) is preserved by substitution

If \( x \sim y \) in \( \Phi \vdash res \sim res' \) and term \( \sim \) term' then \( \Phi \vdash \text{term}/x(res) \sim \text{term}/y(res') \).

Proof sketch: By induction and term \( \sim \) term' assumed to be preserved by substitution (see section B4 Proof Judgements).

B9.3  Well-typed spines produce substitutions and the same return type

Assume: \( :: \Phi; \overline{\text{spine}_1} :: \psi_1(\text{fun}_1) \Rightarrow \text{ret}_1 \)

\( \Phi \vdash h \Leftarrow \overline{\text{R}} \) and \( \psi_1(\text{fun}_1) = \psi_2(\text{fun}_2) = \psi_2(\text{fun}_3) \)

\( \langle h + f; \overline{x_i} = \text{spine}_1 \rangle :: \psi_2(\text{fun}_2) \Rightarrow \langle \text{heap}; \sigma_2; \text{ret}_2 \rangle \)

\( \overline{x_i} :: \text{fun}_3 \sim \overline{\text{C}}; \overline{\text{L}}; \overline{\Phi}; \overline{\text{R}}' | \text{ret}_3. \)

Prove: \( \psi_1(\text{ret}_1) = \psi_2(\text{ret}_2) = [\psi_2, \sigma_2](\text{ret}_3) \)

\( \exists h', \overline{\text{R}}'. \text{heap} = h' + f, \Phi \vdash h' \Leftarrow \overline{\text{R}}' \) and

\( :: \Phi; \overline{\text{R}}' \vdash \psi_2(\sigma_2) \Leftarrow (\overline{\text{C}}; \overline{\text{L}}; \psi_2(\text{fun}_2)(\overline{\text{R}}')). \)

(1) Case: Expl_Spine_Ret

\( :: \Phi; \overline{\text{R}} \vdash \psi_1(\text{ret}) \Rightarrow [\psi_1, \cdot](\text{ret}) \) (by assumption)

Let: \( h' = \cdot, \overline{\text{R}}' = \cdot. \)

\( f = h' + f \) trivially.

\( \Phi \vdash \cdot \Leftarrow \cdot \) by Heap_Empty.

\( :: \Phi; \overline{\text{R}} \vdash \cdot \Leftarrow (\cdot; \cdot) \) by Subs_CHK_Empty.

(1)2 Case: Expl_Spine_Comp

\( :: \Phi; \overline{\text{R}} \vdash \psi_1(\text{ret}) \Rightarrow [\psi_1, \cdot](\text{ret}) \)

\( \cdot \vdash \text{pval} \Rightarrow \beta \)

\( \langle h + f; \overline{x_i} = \text{spine}_1 \rangle :: [\text{pval}/x, \psi_1](\text{fun}_1) \Rightarrow \text{ret}_1 \)

(by inversion, Subs_SPINE_Comp) \( x, \overline{x_i} :: \Pi x; \beta. \text{fun}_3 \sim x; \beta, \overline{\text{C}}; \overline{\text{L}}; \overline{\Phi}; \overline{\text{R}}' | \text{ret}_3 \) (by inversion, Fun_ENV_Ret).

\( [\text{pval}/x, \psi_1](\text{ret}_1) = [\text{pval}/x, \psi_2](\text{ret}_2) = [\text{pval}/x, \psi_2, \sigma_2](\text{ret}_3) \)

\( \exists h', \overline{\text{R}}'. \text{heap} = h' + f, \Phi \vdash h' \Leftarrow \overline{\text{R}}' \) and

\( :: \Phi; \overline{\text{R}}' \vdash [\text{pval}/x, \psi_2](\sigma_2) \Leftarrow (\overline{\text{C}}; \overline{\text{L}}; [\text{pval}/x, \psi_2](\text{R}')). \) (by induction).
[ψ₂, pval/x, σ₂′][ret₃] = [pval/x, ψ₂, σ₂′] and
::; Φ; R₀ ⊢ ψ₂([pval/x, σ₂′]) ⇐ (C′, x; β; L; ψ₂(R′)),
by SUBSCHK_COMP and SUBSCHK_CONCAT (because pval is closed, we have
[pval/x, ψ₂(σ₂′)] = ψ₂([pval/x, σ₂′])).

(1.3) **CASE: EXPL_endian**

Similar to EXPL_endian but with SUBSCHK_LOG.

(1.4) **CASE: EXPL_endian**

By induction (does not affect substitution).

(1.5) **CASE: **EXPL_endian**

By induction (does not affect substitution).

ψ₁(ret₁) = [res_val/x, ψ₂](ret₂) = [res_val/x, ψ₂, σ₂′](ret₃)
(by resources variables not in types)
∀h′, R₀. heapₗ = h′ + h + f and res_val = res_val′
(by lemma B8.8 (Progress and type preservation for resource terms)).

Let: h′ = h′ + h and R₀ = R₀, R₀.

Hence Φ ⊢ h′ ⇐ R₀, R₀ (by HEAP_CONCAT),
[res_val/x, ψ₂, σ₂′](ret₃) = [pval, res_val/x, σ₂′](ret₃) and
::; Φ; R₀ ⊢ ψ₂([res_val/x, σ₂′]) ⇐ (C; L; ψ₂(x; res, R′)), by SUBSCHK_RES and SUBSCHK_CONCAT (because res_val is closed, we have [res_val/x, ψ₂(σ₂′)] = ψ₂([res_val/x, σ₂′])).

**B9.4 Well-typed values pattern-match successfully**

Note that the definition of term ∼ term′ is not explicitly stated; see section B4 (Proof Judgements) for more details.

**Assume:**
1. C; L; Φ ⊢ ret_pat₁ :ret ∼ C′; L′; Φ′; R′
2. ret_pat₁ is exhaustive
3. Φ ⊢ fun ∼ ret
4. ::; Φ; R ⊢ ret_term₁ : fun ∼ 1
5. Φ ⊢ h ⇐ R
PROVE: \( \exists h', \sigma. \forall f. \langle h + f; \text{ret\_pat}_i = \text{ret\_term}_i \rangle \leadsto \langle h' + f; \sigma \rangle \)

\( \exists R'. \)

\( \mathcal{C}; \mathcal{L}; \Phi \vdash h' \leftarrow R' \land \mathcal{C}; \mathcal{L}; \Phi; R' \vdash \sigma \leftarrow (\mathcal{C}'; \mathcal{L'}; \mathcal{R}'). \)

PROOF SKETCH: Induction over the pattern-matching judgement.

\( \top \)1. CASE: \textbf{Pat\_Ret\_Empty}

\( \mathcal{C}; \mathcal{L}; \Phi \vdash \text{comp} \text{ident\_or\_pat}, \text{ret\_pat}_j: \Sigma y; \beta. \text{ret} \leadsto C_1; C_2; \mathcal{L}_2; \Phi_2; R_2 \)

which means \( \text{fun} = \text{I} \) (by inversion, \text{Rel\_Ret\_I})

and so \( \mathcal{C}; \mathcal{L}; \Phi; \vdash :\text{I} \gg \text{I} \) (by inversion, \text{Expl\_Spine\_Ret}),

and \( h = \cdot \) (by lemma B8.4).

Let \( h' = \cdot \), to step with \text{Subs\_Pat\_Ret\_Empty}.

Let \( R' = \cdot \), to type \( h' \) with \text{Heap\_Empty} and \( \sigma \) with \text{Subs\_CHK\_Empty}.

\( \top \)2. CASE: \textbf{Pat\_Ret\_Comp}

\( \mathcal{C}; \mathcal{L}; \Phi \vdash \text{comp} \text{ident\_or\_pat}, \text{ret\_pat}_j: \Sigma y; \beta. \text{ret} \leadsto C_1; C_2; \mathcal{L}_2; \Phi_2; R_2 \)

which means \( \text{fun} = \Pi x; \beta. \text{fun}' \) (by inversion, \text{Rel\_Ret\_Comp}),

and so \( \mathcal{C}; \mathcal{L}; \Phi; R \vdash \text{pval}, \text{ret\_term}_j :: \Sigma x; \beta. \text{fun}' \gg \text{I} \) (by inversion, \text{Expl\_Spine\_Comp}).

\( \text{ident\_or\_pat}; \beta \leadsto C_1 \text{ with } \text{term}_1 \) (from the pattern-matching assumption),

\( \text{ident\_or\_pat} \) is exhaustive (from the exhaustive assumption),

and \( \cdot \vdash \text{pval} \Rightarrow \beta \) (from the spine typing assumption),

imply \( \text{term}_1 \sim \text{pval}, \text{ident\_or\_pat} = \text{pval} \leadsto \sigma_1 \)

and \( \cdot ; \cdot ; \cdot ; \vdash \sigma_1 \leftarrow (C_1; \cdot ; \cdot ) \) (by the nested proof below).

\( \mathcal{C}, C_1; \mathcal{L}; \Phi \vdash \text{ret\_pat}_j : \text{term}_1 / y(\text{ret}') \leadsto C_2; \mathcal{L}_2; \Phi_2; R_2 \) (from the pattern-matching assumption),

\( \forall \text{term}_1 \sim \text{pval}. \Phi \vdash \text{pval} / x(\text{fun}') \sim \text{term}_1 / y(\text{ret}') \) (from the related assumption),

\( \cdot ; \cdot ; \cdot ; R \vdash \text{ret\_term}_j :: \text{pval} / x(\text{fun}') \gg \text{I} \) (from the spine typing assumption),

and \( \cdot \vdash h \leftarrow R, \) imply \( \langle h + f; \text{ret\_pat}_j = \text{ret\_term}_j \rangle \leadsto \langle h'' + f; \sigma_2 \rangle \)

and that \( \exists R'' \) such that \( \mathcal{C}, C_1; \mathcal{L}; \Phi \vdash h'' \leftarrow R'' \) and \( \mathcal{C}, C_1; \mathcal{L}; \Phi; R'' \vdash \sigma_2 \leftarrow (C_2; \mathcal{L}_2; \mathcal{R}_2) \) (by induction).

Since \( \mathcal{C}; \mathcal{L}; \sigma_1(\Phi); \sigma_1(\mathcal{R}') \vdash [\text{id}, \sigma_1] \leftarrow (C, C_1; \mathcal{L}; \mathcal{R}') \), and \( \sigma_1(\Phi) = \Phi \) (because \( \Phi \) is well-scoped / does not contain any variables from \( C_1 \)) we have \( \mathcal{C}; \mathcal{L}; \Phi \vdash \sigma_1(h'') \leftarrow \sigma_1(\mathcal{R}'') \) and \( \mathcal{C}; \mathcal{L}; \Phi; \sigma_1(\mathcal{R}') \vdash \sigma_1(\sigma_2) \leftarrow (C_2; \mathcal{L}_2; \sigma_1(\mathcal{R}_2)) \) (by lemma B7.3 (Substitution)).

LET: \( h' = \sigma_1(h''), \sigma = [\sigma_1, \sigma_2] \) to step with \text{Subs\_Pat\_Ret\_Comp}.

\( \mathcal{R}' = \sigma_1(\mathcal{R}''). \)

So \( \mathcal{C}; \mathcal{L}; \Phi \vdash h' \leftarrow \mathcal{R}' \)

and \( \mathcal{C}; \mathcal{L}; \Phi; \mathcal{R}' \vdash \sigma \leftarrow (C_1, C_2; \mathcal{L}_2; \mathcal{R}_2) \) hold by lemma B6 (Weakening) and \text{Subs\_CHK\_Concat}.

ASSUME: 1. \text{ident\_or\_pat}; \beta \leadsto C_1 \text{ with } \text{term}_1

2. \text{ident\_or\_pat} \) is exhaustive

27
3. \( \cdot \vdash pval \Rightarrow \beta \)

PROVE: 1. \( \text{term}_1 \sim pval \)
2. \( \exists \sigma. \text{ident_or_pat} = pval \leadsto \sigma \) and \( \cdot ; ; ; ; \vdash \sigma \rightleftharpoons (C_1; ; ; ) \).

(2)1. CASE: \text{Pat\_Comp\_No\_Sym\_Annot}
   Proof: \( \text{term}_1 \) is a wildcard (fresh variable) which would unify with \( pval \); let \( \sigma = \cdot \) for \text{Subs\_Pat\_Value\_No\_Sym\_Annot} / \text{Subs\_Chk\_Empty}.

(2)2. CASE: \text{Pat\_Comp\_Sym\_Annot, Pat\_Sym\_Or\_Pat\_Sym}
   Proof: \( \text{term}_1 = x \), a fresh pattern variable, so would unify with \( pval \);
   let \( \sigma = pval \leadsto x \) for \text{Subs\_Pat\_Value\_Sym\_Annot} / \text{Subs\_Chk\_Comp} (using \( \cdot \vdash pval \Rightarrow \beta \)).

(2)3. CASE: \text{Pat\_Comp\_Nil}
   Proof: \( \text{term}_1 = \text{nil} \), and by inversion on the typing assumption, and then by exhaustiveness, \( pval = \text{Nil}(\cdot) \), so would unify; let \( \sigma = \cdot \) for \text{Subs\_Pat\_Value\_Nil} / \text{Subs\_Chk\_Empty}.

(2)4. CASE: \text{Pat\_Comp\_Cons}
   Proof: \( \text{term}_1 = \text{term}_{11} :: \text{term}_{12} \), and by inversion on the typing assumption, and then by exhaustiveness, \( pval = \text{Cons}(pval_1, pval_2) \). By induction (1) they would unify
   (2) let \( \sigma = [\sigma_1, \sigma_2] \) for \text{Subs\_Pat\_Value\_Cons} / \text{Subs\_Chk\_Comp} and \text{Subs\_Chk\_Concat} (both are independent).

(2)5. CASE: \text{Pat\_Comp\_Tuple}
   Proof: \( \text{term}_1 = (\text{term}_{1}^{\dagger}) \), and by inversion on the typing assumption,
   \( pval = \text{Tuple}(pval_{1}^{\dagger}) \). By induction (1) they would unify
   (2) let \( \sigma = [\sigma_1^{\dagger}] \) for \text{Subs\_Pat\_Value\_Tuple} / \text{Subs\_Chk\_Comp} and \text{Subs\_Chk\_Concat}.

(2)6. CASE: \text{Pat\_Comp\_Array}
   Proof: Similar to \text{Pat\_Comp\_Tuple}, but with \text{Subs\_Pat\_Value\_Array}.

(2)7. CASE: \text{Pat\_Comp\_Specified}
   Proof: By induction we have (1) \( \text{term}_1 \sim pval \), and by the \text{Specified} exception (see Section B4, Proof Judgements) \( \text{term}_1 \sim \text{Specified}(pval) \);
   \( \sigma \) for \text{Subs\_Pat\_Value\_Specified}, typing by induction.

(2)8. CASE: \text{Pat\_Sym\_Or\_Pat\_Pat}
   Proof: By induction.

(1)3. CASE: \text{Pat\_Ret\_Log}
\( C; L; \Phi \vdash \log y', \overline{\text{ret\_pat}^{\dagger}}; ; \exists y: \beta. \text{ret} \sim C_2; y'; \beta, L_2; \Phi_2; R_2 \)
which means \( \text{fun} = \forall x: \beta. \text{fun}' \) (by inversion, \text{Rel\_Ret\_Log})
and so \( ; ; ; ; ;; \Phi; R \vdash \text{oarg}\_\text{ret\_term}_{j} \leadsto \forall x; \beta. \text{fun}' \gg I \) (by inversion, \text{Expl\_Spine\_Log}).

\( C; L, y; \beta; \Phi \vdash \overline{\text{ret\_pat}^{\dagger}}; ; \text{ret}' \sim C_2; L_2; \Phi_2; R_2 \) (from the pattern-matching assumption),
∀ oarg ∼ oarg'. Φ ⊢ oarg/x(fun') ∼ oarg'/y'(ret') (from the related assumption),

\[ \vdash \Phi; \mathcal{R} \vdash \text{ret} \text{term}^j : \text{term} \land \text{fun} \Rightarrow \mathcal{I} \] (from the spine typing assumption)

and Φ ⊢ h ≈ \mathcal{R}, imply \langle h + f; \text{ret} \text{pat}_i = \text{ret} \text{term}^j \rangle \simeq \langle h'' + f; \sigma \rangle

and ∃ \mathcal{R}' such that \mathcal{C}; \mathcal{L}, y'; Φ ⊢ h'' ≈ \mathcal{R}'' and \mathcal{C}; \mathcal{L}, y'; Φ; \mathcal{R}'' ⊢ σ_2 ≈ (\mathcal{C}_2; \mathcal{L}_2; \mathcal{R}_2) (by induction).

Since ; ; ⊢ oarg ⇒ Φ, and oarg/y'(Φ) = Φ (because it is well-scoped / doesn’t refer to y')

and C; L; oarg/y'(Φ); oarg/y'(R') ⊢ [id, oarg/y'] ≈ (C; L, y'; Φ; R')

we have C; L; Φ ⊢ oarg/y'(h'') ≈ oarg/y'(R''), and

C; L; Φ; oarg/y'(R'') ⊢ oarg/y'(σ_2) ≈ (C_2; L_2; oarg/y'(R_2)) (by lemma B7.3 (Substitution)).

LET: h' = oarg/y'(h''), σ = [oarg/y', σ_2] to step with SUBS_PAT_RET_LOG.

\[ \mathcal{R}' = oarg/y'(R''). \]

So \mathcal{C}; \mathcal{L}; Φ ⊢ h' ≈ \mathcal{R}' and \mathcal{C}; \mathcal{L}; Φ; \mathcal{R}' ⊢ σ ≈ (\mathcal{C}_2; y'; Φ; \mathcal{L}_2; \mathcal{R}_2) by SUBS_CHK_CONCAT.

(1.4) CASE: PAT_RET_PHI
\[ \mathcal{C}; \mathcal{L}; Φ ⊢ \text{ret} \text{pat}^j : \text{term} \land \text{ret} \Rightarrow \mathcal{C}'; \mathcal{L}'; Φ'; \text{term}'; \mathcal{R}' \]

which means fun = term ⊃ fun' (by inversion, REL_RET_PHI),

and so ; ; ; Φ; \mathcal{R} ⊢ \text{ret} \text{term}^j : \text{term} → fun' ⇒ \mathcal{I} (by inversion, EXPL_SPINE_RET).

\[ \mathcal{C}; \mathcal{L}; Φ ⊢ \text{fun}' \sim \text{ret}' \] (from the pattern-matching assumption)

Φ ⊢ fun' ∼ ret' (from the related assumption),

\[ ; ; ; Φ; \mathcal{R} ⊢ \text{ret} \text{term}^j : \text{fun}' ⇒ \mathcal{I} \] (from the spine typing assumption)

imply \langle h + f; \text{ret} \text{pat}_i = \text{ret} \text{term}^j \rangle \simeq \langle h' + f; \sigma \rangle and the heap and substitution typings (by induction).

(1.5) CASE: PAT_RET_RES
\[ \mathcal{C}; \mathcal{L}; Φ ⊢ \text{res} \text{pat}, \text{ret} \text{pat}: \text{res}' ∼ \text{ret}' \Rightarrow \mathcal{C}_4; \mathcal{L}_4; Φ_3; Φ_4; \mathcal{R}_3; \mathcal{R}_4 \]

which means fun = res → fun' (by inversion, REL_RET_RES),

and so ; ; ; Φ; \mathcal{R}_1; \mathcal{R}_3 ⊢ \text{res} \text{term}, \text{spine} : \text{res} → \text{fun} ⇒ \mathcal{I} (by inversion, EXPL_SPINE_RET),

and h = h_1 + h_2 where Φ ⊢ h_1 ≈ \mathcal{R}_1 and Φ ⊢ h_2 ≈ \mathcal{R}_2 (by lemma B8.4).

; ; ; ; Φ_1 ⊢ \text{res} \text{term} ≈ \text{res} (from the spine typing assumption)

and Φ ⊢ h_1 ≈ \mathcal{R}_1, imply ∃ \text{res} \text{val}, \mathcal{R}_1', h_1' such that Φ ⊢ h_1' ≈ \mathcal{R}_1', ; ; ; Φ_1; \mathcal{R}_1' ⊢ \text{res} \text{val} ≈ \text{res}'

and \langle h_1 + h_2 + f; \text{res} \text{term} \rangle ⊥ \langle h_1' + h_2 + f; \text{res} \text{val} \rangle (by lemma B8.6 (Progress and type preservation for resource terms)).

\[ \mathcal{C}; \mathcal{L}; Φ ⊢ \text{res} \text{pat}: \text{res}' \Rightarrow \mathcal{C}_3; \mathcal{L}_3; Φ_3; \mathcal{R}_3 \] (from the pattern matching assumption),

Φ ⊢ res ∼ res' (from the related assumption),

; ; ; Φ; \mathcal{R}_1' ⊢ \text{res} \text{val} ≈ \text{res} and Φ ⊢ h_1' ≈ \mathcal{R}_1', imply

∃ h''_1, \mathcal{R}_1'' such that \langle h_1' + h_2 + f; \text{res} \text{pat} = \text{res} \text{val} \rangle \Rightarrow \langle h''_1 + h_2 + f; σ_1 \rangle \mathcal{C}; \mathcal{L}; Φ ⊢ h''_1 ≈ \mathcal{R}_1'',

and \mathcal{C}; \mathcal{L}; Φ; \mathcal{R}_1'' ⊢ σ_1 ≈ (; ; \mathcal{L}_3; \mathcal{R}_3) (by the nested proof below).

\[ \mathcal{C}; \mathcal{L}; Φ ⊢ \text{ret} \text{pat}^j : \text{ret} \Rightarrow \mathcal{C}_4; \mathcal{L}_4; Φ_4; \mathcal{R}_4 \] (from the pattern matching assumption),

Φ ⊢ fun' ∼ ret' (from the related assumption),

29
\[; ; \Phi; R_2 \vdash \text{ret\_term}^j \; :: \; \text{fun} \Rightarrow I \] (from the spine typing assumption),
and \( \Phi \vdash h_2 \leq R_2 \), imply \( \langle h_2 + h'_2 + f; \text{ret\_pat} = \text{ret\_term}^j \rangle \sim \langle h'_2 + h'' + f; \sigma_2 \rangle \) and \( \exists R_2' \) such that \( C; L; \Phi; h_2' \leq R_2', \) and \( C; L; \Phi; R_2' \vdash \sigma_2 \leq (C_4; L_4; R_4) \) (by induction).

Let: \( h' = h'' + h_2' \)
\( \sigma = [\sigma_1, \sigma_2] \) to step with \text{Subs\_Pat\_Ret\_Res}.
\( R' = R_1', R_2' \).
So \( C; L; \Phi; h' \leq R' \) by \text{Heap\_Concat}
and \( C; L; \Phi; R' \vdash \sigma \leq (C_4; L_3; L_4; R_3; R_4) \) by \text{Subs\_chk\_Concat}
(because \( R_2' \) is well-formed w.r.t. \( C; L \), it does not contain any variables from \( L_3; R_3 \) so \( \sigma_1(R_2') = R_2' \)).

Assume: 1. \( L; \Phi \vdash \text{res\_pat}; \text{res} \sim \langle L'; \Phi' \rangle; R' \)
2. \( \Phi \vdash \text{res} \sim \text{res}' \)
3. \( \exists R. (\langle ; ; ; \Phi; R \vdash \text{res\_val} \leq \text{res} \rangle \land (C; L; \Phi \vdash h \leq R) \)

Prove: \( \exists h', \sigma \).
\( \forall f. \langle h + f; \text{res\_pat} = \text{res\_val} \rangle \sim \langle h' + f; \sigma \rangle \)
\( \exists R'. C; L; \Phi \vdash h' \leq R' \land C; L; \Phi; R' \vdash \sigma \leq (\langle ; ; ; \rangle; R') \).

(2) Case: \text{Pat\_Res\_Match\_Fold}
\( L; \Phi \vdash \text{fold}(\text{res\_pat}); \alpha(\text{ptr}, \text{iar}_{i\_1}^{-1})(\text{oarg}) \sim \langle L'; \Phi'; R' \rangle \)
which means \( \text{res} = \alpha(\text{ptr}, \text{iar}_{i\_1}^{-1})(\text{oarg}) \) (by inversion, \text{Rel\_Res\_Pred})
and so \( R = \alpha(\text{ptr}, \text{iar}_{i\_1}^{-1})(\text{oarg}) \) and \( \text{res\_val} = \alpha(\text{ptr}^2, \text{iar}_{i\_2}^{-1}) \) (by lemma B8.3).

(3) 1. \( h = \{ \alpha(\text{ptr}, \text{iar}_{i\_1}^{-1})(\text{oarg}) \} \) & def & heap
Proof: \( \alpha \neq \text{O\_n\_med}(\tau) \) (from the pattern-matching assumption), and lemma B8.4 (Normalised resource context determines structure of heap).

(3) 2. \( \exists R_4' \).
1. \( \alpha \equiv x_p; \text{pointer}, \bar{x}_i; \beta_i^{-1}, y; \text{record} \tag{j}; \beta_{j'}^{-1} \Rightarrow \text{res}\_\text{p} \in \text{Globals} \)
2. \( C; L; \Phi; R_4' \vdash \text{def} \leq [\text{oarg}/y, [\text{iar}_{i\_1}^{-1}], \text{ptr}/x_p](\text{res''}) \)
3. \( C; L; \Phi \vdash \text{heap} \leq R_4' \)
Proof: By inversion, \( C; L; \Phi \vdash h \leq R \) is \text{Heap\_Pred\_Other}.

(3) 3. \( L; \Phi \vdash \text{res\_pat}; \{ \text{oarg}/y, [\text{iar}_{i\_1}^{-1}], \text{ptr}/x_p \}(\text{res''}) \sim \langle L; \Phi; R \rangle \)
Proof: By inversion on the pattern-matching assumption.

(3) 4. \( \Phi \vdash [\text{oarg}/y, [\text{iar}_{i\_1}^{-1}], \text{ptr}/x_p](\text{res''}) \sim [\text{oarg}/y, [\text{iar}_{i\_1}^{-1}], \text{ptr}/x_p](\text{res''}) \)
Proof: By lemma B9.2, using \( \Phi \vdash \text{res''} \sim \text{res''} \) (by lemma B9.1) and \( \text{ptr} \sim \text{ptr}' \),
\( \text{iar}_i \sim \text{iar}_{i\_1}^{-1} \) and \( \text{oarg} \sim \text{oarg}' \).

(3) 5. \( \langle \text{heap} + f; \text{res\_pat} = \text{res\_val} \rangle \sim \langle h' + f; \sigma \rangle \)
Proof: By induction, using (3)2, (3)3 and (3)4.

(3) 6. Step with \text{Subs\_Pat\_Res\_Fold}.

(3) 7. \( h', R' \) as given by induction.
(2.2) Case: \texttt{Pat_Res_Match_Emp / Pat_Res_Match_Phi} \\
res = \texttt{emp} or \texttt{term} (by inversion, REL_Res_Emp / REL_Res_Phi) and so 
res \_val = \texttt{emp} or \texttt{term} and \texttt{R} = \cdot (by lemma B8.3), meaning \( h = \cdot \) (by lemma B8.4). 
Proof: Let \( h' = \cdot \) to step with \texttt{Subs_Pat_Res_Emp / Subs_Pat_Res_Phi}. 
\( \texttt{R}' = \cdot \), so \texttt{Heap_Empty} and \texttt{Subs_Chk_Empty} suffice.

(2.3) Case: \texttt{Pat_Res_Match_If_True / Pat_Res_Match_If_False} \\
Only showing true case, false case is symmetric.
res' = if \texttt{term}' then res'_1 else res'_2 so 
res = if \texttt{term} then res_1 else res_2 (by inversion, REL_Res_IF).

Since \texttt{smt} (\( \Phi \Rightarrow \texttt{term}' \)) (from the pattern-matching assumption) and 
\texttt{smt} (\( \Phi \Rightarrow \texttt{term} \Leftrightarrow \texttt{term}' \)), we can conclude the typing assumption must be \texttt{Res_Chk_If_True}.

From there, we proceed by induction.

(2.4) Case: \texttt{Pat_Res_Match_Var} \\
Proof: Let \( h' = h \) to step with \texttt{Subs_Pat_Res_Var}. 
\( \texttt{R}' = \texttt{R} \) so \texttt{Subs_Chk_Res}.

(2.5) Case: \texttt{Pat_Res_Match_SepConj} \\
\( \mathcal{L}; \Phi \vdash \langle \texttt{res\_pat}_1, \texttt{res\_pat}_2 \rangle : \texttt{res}'_1 * \texttt{res}'_2 \Rightarrow \mathcal{L}_1, \mathcal{L}_2; \Phi_1, \Phi_2; \texttt{R}_1, \texttt{R}_2 \)
res = res_1 * res_2 (by inversion, REL_Res_SepConj) and 
\vdash; \cdot; \Phi; \texttt{R}_1, \texttt{R}_1 \vdash \langle \texttt{res\_val}_1, \texttt{res\_val}_2 \rangle \Leftarrow res_1 * res_2 (by lemma B8.3), 
so \( h = h_1 + h_2 \) where \( \Phi \vdash h \Leftarrow \texttt{R}_1 \) and \( \Phi \vdash h \Leftarrow \texttt{R}_2 \).

By induction, obtain \( h'_1 \) and \( h'_2 \), and then let \( h' = h'_1 + h'_2 \). Instantiate the frame, from 
the inductive hypothesis with \( h_2 + f \) and then \( h'_1 + f \) to conclude 
\langle h_1 + h_2 + f; \texttt{res\_pat}_1 = \texttt{res\_val}_1 \rangle \Rightarrow \langle h'_1 + h_2 + f; \sigma_1 \rangle \) and 
\langle h_2 + h'_1 + f; \texttt{res\_pat}_2 = \texttt{res\_val}_2 \rangle \Rightarrow \langle h'_2 + h'_1 + f; \sigma_2 \rangle \) to step with \texttt{Subs_Pat_Res_Pair}. 
Let: \( \texttt{R}' = \texttt{R}'_1, \texttt{R}'_2 \) (obtained from induction).
We then have and \( \mathcal{C}_1 \mathcal{L}; \Phi \vdash h'_1 + h'_2 \Rightarrow \texttt{R}' \) and (since \( \sigma_1(\texttt{R}_2) = \texttt{R}_2 \) because it can not 
refer to \( \mathcal{L}_1 \)) \( \mathcal{C}_1 \mathcal{L}; \Phi; \texttt{R}' \vdash [\sigma_1, \sigma_2] \Leftarrow (\cdot; \mathcal{L}_1, \mathcal{L}_2; \texttt{R}_1, \texttt{R}_2) \).

(2.6) Case: \texttt{Pat_Res_Match_Pack} \\
\( \mathcal{L}; \Phi \vdash \texttt{pack}(x, \texttt{res\_pat}') : \exists y; \beta. \texttt{res}'_1 \Rightarrow x; \beta, \mathcal{L}', \Phi'; \texttt{R}' \)
res = \exists y; \beta. \texttt{res}_1 (by inversion, REL_Res_Exists) and 
\vdash; \cdot; \Phi; \texttt{R} \vdash \texttt{pack}(\texttt{oarg}, \texttt{res\_val}') \Leftarrow \exists y; \beta. \texttt{res}_1 (by lemma B8.3).
\( \mathcal{L}, x; \beta; \Phi \vdash \texttt{res\_pat}' : x/y'(\texttt{res}'_1) \Rightarrow \mathcal{L}', \Phi'; \texttt{R}' \) (from the pattern-matching assumption), 
\vdash; \cdot; \Phi; \texttt{R} \vdash \texttt{res\_val}' \Leftarrow \texttt{oarg}/y(\texttt{res}_1) \) (from the typing assumption), 
\( \forall \texttt{term} \sim \texttt{term}'. \Phi \vdash \texttt{term}/y(\texttt{res}_1) \sim \texttt{term}'/y'(\texttt{res}'_1) \) (from the related assumption), 
\texttt{oarg} \sim x \text{ imply } \exists h'', \sigma'. \forall f \ldots 
and \( \exists \texttt{R}'. \mathcal{C}, \mathcal{L}, x; \beta; \Phi \vdash h' \Leftarrow \texttt{R}' \land \mathcal{C}; x; \beta, \mathcal{L}, \Phi; \texttt{R}'' \vdash \sigma' \Leftarrow (\cdot; \mathcal{L}', \texttt{R}') \)
Since \( \vdash; \cdot \vdash \texttt{oarg} \Rightarrow \beta \), and \texttt{oarg}/x(\Phi) = \Phi (because it is well-scoped / doesn’t refer to
\( x \) and \( \mathcal{C}; \mathcal{L}; oarg/x(\Phi); oarg/x(\mathcal{R}') \vdash [id, oarg/x] \Leftarrow (\mathcal{C}; x; \beta, \mathcal{L}'; \mathcal{R}'), \)
we have \( \mathcal{C}; \mathcal{L}; \Phi \vdash oarg/x(h'') \equiv oarg/x(\mathcal{R}'') \), and
\( \mathcal{C}; \mathcal{L}; \Phi; oarg/x(\mathcal{R}'') \vdash oarg/x(\sigma') \Leftarrow (\cdot; \mathcal{L'}; oarg/x(\mathcal{R}_2)) \) (by lemma B7.3 (Substitution)).

Let: \( h' = oarg/x(h'') \)
\( \sigma = [oarg/x, \sigma'] \) to step with Subs_Pat_Res_Pack.
\( \mathcal{R}' = oarg/x(\mathcal{R}'') \) so \( \mathcal{C}; \mathcal{L}; \Phi; oarg/x(\mathcal{R}'') \vdash [oarg/x, \sigma'] \Leftarrow (\cdot; \mathcal{L'}; \mathcal{R}_2) \) by Subs_Chk_Concat.

B9.5 \( \Phi \vdash \text{to\_fun } \text{ret} \sim \text{ret} \)

**Proof sketch:** Induction over \( \text{ret} \).

**B9.6 Statement and proof**

**Assume:**
1. Closed (no free-variables) expression \( \text{texpr} \).
2. \( \cdot; \cdot; \Phi; \mathcal{R} \vdash \text{texpr} \Leftarrow \text{ret} \)
3. All patterns in \( \text{texpr} \) are exhaustive.

**Prove:** Either \( \text{texpr} \) is a value \( \text{tval} \), or it is unreachable, or
\( \forall h, \mathcal{R}. (\Phi \vdash h \Leftarrow \mathcal{R}) \Rightarrow \exists h', \text{texpr}'. \langle h; \text{texpr} \rangle \rightarrow \langle h'; \text{texpr}' \rangle. \)

**Proof sketch:** Induction over the typing rules.

\( \langle 1 \rangle \) **Case:** Value typing rules (see B5.3).
**Proof:** All these judgements/rules give types to syntactic values; and there are no operational rules corresponding to them (see Section B3).

\( \langle 1 \rangle \) **Case:** \text{Pure\_Top\_Val\_Undef}, \text{Pure\_Top\_Val\_Error}, \text{Expl\_Top\_Val\_Undef}, \text{Expl\_Top\_Val\_Error}.
**Proof:** All these rules require inconsistent constraint context, and so would be unreachable.

\( \langle 1 \rangle \) **Case:** \text{Pure\_Expr\_Array\_Shift}.
**Proof:** By inversion on \( \cdot \vdash pval_l \Rightarrow \text{pointer}, pval_1 \) must be a \text{mem\_ptr} (\text{Pure\_Val\_Obj\_Ptr}). Similarly \( pval_2 \) must be a \text{mem\_int}, so step with PE\_TP\_Array\_Shift.

\( \langle 1 \rangle \) **Case:** \text{Pure\_Expr\_Member\_Shift}.
**Proof:** \( pval \) must be a \text{mem\_ptr} so step with PE\_TP\_Member\_Shift.

\( \langle 1 \rangle \) **Case:** \text{Pure\_Expr\_Not}.
**Proof:** \( pval \) must be a \text{bool\_value} so step with PE\_TP\_Not\_True or PE\_TP\_Not\_False.

\( \langle 1 \rangle \) **Case:** \text{Pure\_Expr\_Arith\_Binop}, \text{Pure\_Expr\_Rel\_Binop}.
**Proof:** \( pval_l \) and \( pval_2 \) must be \text{mem\_ints}, so step with PE\_TP\_Arith\_Binop or PE\_TP\_Rel\_Binop respectively.
1.7. CASE: Pure.Expr_Bool_Binop.
   Proof: \( pval_1 \) and \( pval_2 \) must be bool_values, so step with PE_TP_Bool_Binop.

1.8. CASE: Pure.Expr_Call.

   (2)1. name: \( \text{pure}_i \equiv \overline{x_i} \mapsto \text{tpexpr} \in \text{Globals} \).
        \( \vdash; \Phi; \vdash \overline{pval}_i : \text{pure}_i \gg \Sigma y; \beta. \text{term} \land I \).
        Proof: By inversion on the assumption.

   (2)2. \( \langle ; x_i = pval_i \rangle : \text{pure}_i \gg \langle ; \sigma; \Sigma y; \beta. \text{term} \land I \rangle \).
        Proof: By lemma B9.3.

   (2)3. Thus it can step with PE_TP_Call.

1.9. CASE: Pure.Expr_Assert_Undef.

   (2)1. \( pval \) must be a bool_value

   (2)2. \( \text{smt}(\Phi \Rightarrow \Phi) \). Proof: By inversion on the assumption.

   (2)3. If it is False, then by the latter, we have an inconsistent constraints context, meaning
       the code is unreachable.

   (2)4. If it is True, we may step with PE_TP_Assert_Undef.

1.10. CASE: Pure.Expr_Bool_To_Integer.
      Proof: \( pval \) must be a bool_value (Pure.Val_True, Pure.Val.False) and so step with
            PE_TP_Bool_To_Integer_True, PE_TP_Bool_To_Integer.False respectively.

1.11. CASE: Pure.Expr_WrapI.
      Proof: \( pval \) must be a mem_int (Pure.Val_Obj_Ptr) and so step with PE_TP_WrapI.

1.12. CASE: Pure_Top_IF, Pure_Top_CASE, Pure_Top_Let, Pure_Top_LetT.
      Proof: See Expl_Top_SEQ_IF, Expl_Top_SEQ_CASE, Expl_Top_SEQ_Let, Expl_Top_SEQ_LetT, case for more general proofs.


   (2)1. \( pval \) must be a mem_int.

   (2)2. \( h \) must be \( \cdot \) (empty).
       Proof: By Heap.Empty.

   (2)3. Step with Action.Is_Create.
       Proof: \( \text{mem_ptr} \) is free in the premises and so can be constructed to satisfy the
       requirements.

1. \(pval_0\) must be a `mem_ptr`.
   \(\text{Proof: By Pure.Val_Obj_Ptr.}\)

2. \(\vdash \Phi; R' \vdash res\_term \Rightarrow term \mapsto \tau \ pval_1\).
   \(\text{smt}(\Phi \Rightarrow (term = \text{mem\_ptr}) \land (init = \text{const.true})).\)
   \(\text{Proof: By inversion on the typing assumption and (2)1.}\)

3. \(\exists h', R', res\_val.\)
   1. \(\Phi \vdash h' \leftarrow R'\)
   2. \(\langle h; res\_term \rangle \downarrow \langle h'; res\_val \rangle\)
   3. \(\vdash \Phi; R' \vdash res\_val \Rightarrow term \mapsto \tau \ pval_1\)
   \(\text{Proof: By (2)2 and lemma B8.6 (Progress and type preservation for resource terms).}\)

4. \(res\_val = \text{Owned}(\tau)(term)\).
   \(\text{Proof: By lemma B8.3 (Non-conditional resources determine context and values).}\)

5. \(h' = \{term \mapsto \tau \ pval_1 \& \text{None}\}\).
   \(\text{Proof: By inversion on the term typing assumption in (2)3 using (2)4, } \Phi \vdash h' \leftarrow R'\)
   \(\text{and lemma B8.4 (Normalised resource context determines structure of heap).}\)


1. \(pval_0\) must both be a `mem_ptr`.
   \(\text{Proof: By Pure.Val_Obj_Ptr.}\)

2. \(\vdash \Phi; R' \vdash res\_term \Rightarrow term \mapsto \tau \_\_\_\_\_\_\_.\)
   \(\text{smt}(\Phi \Rightarrow term = \text{mem\_ptr}).\)
   \(\text{Proof: By inversion on the typing assumption and (2)1.}\)

3. \(\exists h', R', res\_val.\)
   1. \(\Phi \vdash h' \leftarrow R'\)
   2. \(\langle h; res\_term \rangle \downarrow \langle h'; res\_val \rangle\)
   3. \(\vdash \Phi; R' \vdash res\_val \Rightarrow term \mapsto \tau \_\_\_\_\_\_.\)
   \(\text{Proof: By (2)2 and lemma B8.6 (Progress and type preservation for resource terms).}\)

4. \(res\_val = \text{Owned}(\tau)(term)\).
   \(\text{Proof: By lemma B8.3 (Non-conditional resources determine context and values).}\)

5. \(h' = \{term \mapsto \tau \_\_\_\_\_\_\_\_\& \text{None}\}\).
   \(\text{Proof: By inversion on the term typing assumption in (2)3 using (2)4, } \Phi \vdash h' \leftarrow R'\)
   \(\text{and lemma B8.4 (Normalised resource context determines structure of heap).}\)

6. Step with `Action_Save`.

16. Case: `Expl\_Is\_Action\_Kill\_Static`
(2.1) \textit{pval} must be a \textit{mem\_ptr}.
Proof: By Pure\_Val\_Obj\_Ptr.

(2.2) \textit{\cdot\cdot\cdot}; Φ; \mathcal{R} \vdash \text{res}\_term \Rightarrow \text{term} \mapsto term^\tau.
Proof: By inversion on the typing assumption and (2.1).

(2.3) \exists h', \mathcal{R}', \text{res\_val}.
1. \Phi \vdash h' \leftarrow \mathcal{R}'
2. ⟨h; \text{res}\_term⟩ ⇓ ⟨h'; \text{res}\_val⟩
3. \textit{\cdot\cdot\cdot}; Φ; \mathcal{R}' \vdash \text{res}\_val \Rightarrow \text{term} \mapsto term^\tau.
Proof: By (2.2) and lemma B8.6 (Progress and type preservation for resource terms).

(2.4) \text{res\_val} = \text{Owned} (\tau)(\text{term}).
Proof: By lemma B8.3 (Non-conditional resources determine context and values).

(2.5) h' = \{\text{term} \mapsto term^\tau \& \text{None}\}.
Proof: By inversion on the typing assumption in (2.3), \Phi \vdash h' \leftarrow \mathcal{R}' and lemma B8.4 (Normalised resource context determines structure of heap).

(2.6) Step with Action\_Is\_Kill\_Static.

(1.17) Case: Expl\_Is\_Memop\_Rel\_Binop.
Proof: Similar to Pure\_Expr\_Rel\_Binop, but step with Memop\_Is\_Rel\_Binop.

(1.18) Case: Expl\_Is\_Memop\_IntFromPtr.
Proof: \textit{pval} must be a \textit{mem\_ptr}, so step with Memop\_Is\_IntFromPtr.

(1.19) Case: Expl\_Is\_Memop\_PtrFromInt.
Proof: \textit{pval} must be a \textit{mem\_int}, so step with Memop\_Is\_PtrFromInt.

(1.20) Case: Expl\_Is\_Memop\_PtrValidForDeref.

(2.1) \textit{pval} must be a \textit{mem\_ptr}.
Proof: By Pure\_Val\_Obj\_Ptr.

(2.2) \textit{\cdot\cdot\cdot}; Φ; \mathcal{R} \vdash \text{res}\_term \Rightarrow \text{term} \mapsto term^\tau.
Proof: By inversion on the typing assumption and (2.1).

(2.3) \exists h' \mathcal{R}', \text{res\_val}.
1. \Phi \vdash h' \leftarrow \mathcal{R}'
2. ⟨h; \text{res}\_term⟩ ⇓ ⟨h'; \text{res}\_val⟩
3. \textit{\cdot\cdot\cdot}; Φ; \mathcal{R}' \vdash \text{res}\_val \Rightarrow \text{term} \mapsto term^\tau.
Proof: By (2.2) and lemma B8.6 (Progress and type preservation for resource terms).

(2.4) \text{res\_val} = \text{Owned} (\tau)(\text{term}).
Proof: By lemma B8.3 (Non-conditional resources determine context and values).

\( h' = \{ \text{term} \mapsto \text{init} \rightarrow \tau \land \text{None} \} \).

Proof: By inversion on the typing assumption in \( \langle 2 \rangle 3 \) using \( \langle 2 \rangle 4 \), \( \Phi \vdash h' \subseteq R' \) and lemma B8.4 (Normalised resource context determines structure of heap).

\( \langle 2 \rangle 5 \). Step with \text{Memop\_Is\_PtrValidForDeref}.

\( \langle 1 \rangle 21. \) Case: \text{Expl\_Is\_Memop\_PtrWellAligned}.
   Proof: \( pval \) must be a \text{mem\_ptr}, so step with \text{Memop\_Is\_PtrWellAligned}.

\( \langle 1 \rangle 22. \) Case: \text{Expl\_Is\_Memop\_PtrArrayShift}.
   Proof: \( pval_1 \) must be a \text{mem\_ptr} and \( pval_2 \) must be a \text{mem\_int}, so step with \text{Memop\_Is\_PtrArrayShift}.

\( \langle 1 \rangle 23. \) Case: \text{Expl\_Seq\_CCall}.

\( \langle 2 \rangle 1. \) ident:fun \equiv \overline{x_i^i} \mapsto texpr \in \text{Globals}
   \vdots \vdots \Phi; R \vdash \text{spine\_elem_1}^i \cdot \text{fun} \gg \text{ret}.
   Proof: By inversion.

\( \langle 2 \rangle 2. \) \( \langle h; x_i = \text{spine\_elem_1}^i \rangle \cdot \text{fun} \gg \langle h'; \sigma_2; \text{ret} \rangle \).
   Proof: By \( \langle 2 \rangle 1 \) and lemma B9.3 (Well-typed spines produce substitutions and the same return type).

\( \langle 2 \rangle 3. \) Step with \text{Seq\_T\_CCall}.

\( \langle 1 \rangle 24. \) Case: \text{Expl\_Seq\_Proc}, \text{Expl\_Top\_Seq\_Run}.
   Proof: Similar to \text{Expl\_Seq\_CCall}, but step with \text{Seq\_T\_Proc} / \text{TSeq\_T\_Run}.

\( \langle 1 \rangle 25. \) Case: \text{Expl\_Is\_Memop}.
   Proof: By induction, if \text{memop} is unreachable, then the whole expression is so. \text{memops} are not values. Only stepping cases applies, so step with \text{Is\_Is\_Memop}.

\( \langle 1 \rangle 26. \) Case: \text{Expl\_Is\_Action}, \text{Expl\_Is\_Neg\_Action}.
   Proof: By induction, if \text{action} is unreachable, then the whole expression is so. \text{actions} are not values. Only stepping case applies, so step with \text{Is\_Is\_Action} (or \text{Is\_Is\_Neg\_Action} respectively).

\( \langle 1 \rangle 27. \) Case: \text{Expl\_Top\_Seq\_LetP}, \text{Expl\_Top\_Seq\_LetTP}.
   Proof: See \text{Expl\_Top\_Seq\_Let} / \text{Expl\_Top\_Seq\_LetTP} for more general cases and proofs.

\( \langle 1 \rangle 28. \) Case: \text{Expl\_Top\_Seq\_Let}.
   Proof: By induction, since \text{seq\_expr} is not value, if it is unreachable, the whole expression is so. If \text{seq\_expr} takes a step, the whole expression steps with \text{TSeq\_T\_Let\_LetT}.

\( \langle 1 \rangle 29. \) Case: \text{Expl\_Top\_Seq\_LetT}.  

36
Proof: By induction, if \( texpr \) is unreachable, so is the whole expression.

If it a \( tval \), use lemma B9.4 (Well-typed values pattern-match successfully), with lemma B9.5 (\( \Phi \vdash \text{to_fun} \ ret \sim ret \)) and the assumption that all patterns are exhaustive, so the whole expression steps with \( TSeq.T_LetT.Sub \).

If \( texpr \) takes a step, the whole expression steps with \( TSeq.T_LetT.LetT \).

\(1\)\(^{30}\). **Case: Expl_TopSeq_Case.**
Proof: By assumption that all patterns are exhaustive, and lemma B9.4 (Well-typed values pattern-match successfully), there is at least one pattern against which \( pval \) will match, so \( TSeq.T.Case \).

\(1\)\(^{31}\). **Case: Expl_TopSeq_If.**
Proof: \( pval \) must be a \( bool_value \) and so \( TSeq.T.If.True / TSeq.T.If.False \).

\(1\)\(^{32}\). **Case: Expl_TopSeq_Bound.**
Proof: Step with \( TSeq.T.Bound \).

\(1\)\(^{33}\). **Case: Expl_Top_Is_LetS.**
Proof: Similar to \( Expl_TopSeq_LetT \), but step with \( TIs.T.LetS.Sub / TIs.T.LetS.LetS \) instead.

\(1\)\(^{34}\). **Case: Expl_TopSeq_Expl_Top_Is.**
Proof: Step with \( T.T.TSeq.T / T.T.TIs.T \) respectively.
B10  Type Preservation

B10.1  Owned \(\langle \tau \rangle\) resource output values have type \(\beta_\tau\)

If \(C; L; \Phi; R \vdash \text{Owned}(\langle \tau \rangle)(\text{ptr}) \leftarrow \text{ptr}' \mapsto^{\text{init}} \text{pval}\) then \(C \vdash \text{pval} \Rightarrow \beta_\tau\) and \(C; L \vdash \text{init} \Rightarrow \text{bool}_\tau\).

Proof sketch: Induction over the typing judgements. Only \text{EXPL_IS_ACTION_STORE} constrain \_value of \text{Owned}(\langle \tau \rangle) resources, and its premises ensure it has type \(\beta_\tau\); \text{EXPL_IS_ACTION_LOAD} and \text{EXPL_IS_MEMOP_PtrValidForDeref} simply propagate the value. \text{EXPL_IS_ACTION_Create}, \text{EXPL_IS_ACTION_LOAD} and \text{EXPL_IS_ACTION_STORE} and ensure \_init has type \text{bool}_\tau.

B10.2  Type Preservation Statement and Proof

If \(\cdot \cdot \cdot; \Phi; R \vdash \text{texpr} \leftarrow \text{ret}\) and \(\Phi \vdash h \leftarrow R\), and all top-level functions are well-typed\(^1\) then
\[
\forall f. (h + f; \text{texpr}) \rightarrow (\text{heap}; \text{texpr}') \Rightarrow \exists \Phi', h', R'. (\cdot \cdot \cdot; \Phi; \cdot \subseteq \cdot \cdot; \Phi'; \cdot) \land \text{heap} = h' + f \land (\Phi' \vdash h' \leftarrow R') \land (\cdot \cdot \cdot; \Phi'; R' \vdash \text{texpr}' \leftarrow \text{ret}).
\]

You can equally well prove \(\forall R'. \Phi \vdash h' \leftarrow R' \Rightarrow \cdot \cdot \cdot; \Phi; R' \vdash \text{texpr}' \leftarrow \text{ret}\) instead. Instead of supplying \(R'\) and proving heap typing, you instead invert heap typing to deduce that \(R'\) can only be what you would have supplied anyways.

It’s worth noting that the constraint context will always only contain trivially true constraints (since \(C; L\) are both empty, all the terms in \(\Phi; R\) will be closed). This does not, by itself, guarantee that all conditional resources will be determined (e.g. \text{if default bool then res}_1 \text{ else res}_2\), but there are other ways of excluding this (not allowing under-determined in heaps).

Proof sketch: Induction over the typing rules, which don’t refer to values or unreachable program points.

Assume: 1. \(\cdot \cdot \cdot; \Phi; R \vdash \text{texpr} \leftarrow \text{ret}\),
2. \(\Phi \vdash h \leftarrow R\)
3. all top-level functions are well-typed
4. \(\forall f. (h + f; \text{texpr}) \rightarrow (\text{heap}; \text{texpr}')\)

Prove: \(\exists \Phi', h', R'.
1. \(\cdot \cdot \cdot; \Phi; \cdot \subseteq \cdot \cdot; \Phi'; \cdot\)
2. \(\text{heap} = h' + f\)
3. \(\Phi' \vdash h' \leftarrow R'\)
4. \(\cdot \cdot \cdot; \Phi'; R' \vdash \text{texpr}' \leftarrow \text{ret}\).

(1) Case: Pure_Expr_Array_Shift.

For all pure expressions, \(\Phi \vdash h \leftarrow \cdot, h = \cdot, \text{heap} = f\).

Let: \(h' = \cdot\) and \(R' = \cdot\), so \(\text{heap} = h' + f\) trivially and \(\Phi \vdash \cdot \leftarrow \cdot\) (by Heap_Empty).

\text{ret} = \Sigma y : \text{pointer}. y = \text{mem_ptr} + \text{ptr} (\text{mem_int} \times \text{size_of}(\tau)) \land I

Proof: By Pure_Top_Val.Done, suffices to show \(\cdot \vdash \text{mem_ptr}' \Rightarrow \text{pointer}\) (true by

\(^1\)More precisely, if \text{ident:fun} \equiv \varphi_i \mapsto \text{texpr} \in \text{Globals} and \varphi_i : \text{fun} \rightarrow C''; L''; \Phi''; R'' | \text{ret}'' then \(C''; L''; \Phi''; R'' \vdash \text{texpr} \leftarrow \text{ret}'').
\text{PURE\_VAL\_OBJ\_Ptr} \) and \( \text{smt}(\Phi \Rightarrow \text{mem\_ptr}' = \text{mem\_ptr} + \text{ptr}(\text{mem\_int} \times \text{size}(\tau))) \)
(true by definition of \text{PE\_TP\_ARRAY\_SHIFT}).

\begin{enumerate}
\item \text{CASE: } \text{PURE\_EXPR\_MEMBER\_SHIFT}, \text{PURE\_EXPR\_NOT}, \text{PURE\_EXPR\_ARITH\_BINOP}, \text{PURE\_EXPR\_BOOL\_BINOP}, \text{PURE\_EXPR\_REL\_BINOP}, \text{PURE\_EXPR\_ASSERT\_UNDEF}, \text{PURE\_EXPR\_BOOL\_TO\_INTEGER}, \text{PURE\_EXPR\_WRAP\_I}.
\text{PROOF: } \text{Similar to } \text{PURE\_EXPR\_ARRAY\_SHIFT}.
\item \text{CASE: } \text{PURE\_EXPR\_CALL}.
\text{PROOF: } \text{See } \text{EXPL\_SEQ\_CCALL} \text{ for a more general case and proof.}
\item \text{CASE: } \text{PURE\_TOP\_IF}.
\text{PROOF: } \text{See } \text{EXPL\_TOP\_SEQ\_IF} \text{ for a more general case and proof.}
\item \text{CASE: } \text{PURE\_TOP\_LET}.
\text{PROOF: } \text{See } \text{EXPL\_TOP\_SEQ\_LET} \text{ for a more general case and proof.}
\item \text{CASE: } \text{PURE\_TOP\_LET\_T}.
\text{PROOF: } \text{See } \text{EXPL\_TOP\_SEQ\_LET\_T} \text{ for a more general case and proof.}
\item \text{CASE: } \text{PURE\_TOP\_CASE}.
\text{PROOF: } \text{See } \text{EXPL\_TOP\_SEQ\_CASE} \text{ for a more general case and proof.}
\item \text{CASE: } \text{EXPL\_IS\_ACTION\_CREATE}.
\text{LET: } ret = \Sigma y_p: \beta_{\tau}. \text{term} \land (y_p \stackrel{\text{const}\_false}{\rightarrow_{\tau}} \text{default}(\beta_{\tau}) \ast I)
where \text{term} = \text{representable}(\tau\ast, y_p) \land \text{aligned}(\text{mem\_int}, y_p).
\text{pt} = \text{Owned}(\tau)(\text{mem\_ptr})(oarg) \text{ where}
oarg = \{\text{init} = \text{const}\_false, \text{value} = \text{default}\beta\}_{\tau}.
\text{ASSUME: } \vdots \vdash \Phi \vdash \text{create}(\text{mem\_int}, \tau) \Rightarrow ret \text{ and so } h = \cdot \text{ (by inversion, } \text{HEAP\_EMPTY})
\text{and } \text{heap} = f + \{\text{pt & None}\}.
\text{LET: } h' = \{\text{pt & None}\}, R' = \cdot \text{pt}.
This means \text{heap} = h' + f \text{ (trivially) and } \Phi \vdash h' \Leftarrow R' \text{ (by } \text{HEAP\_PRED\_OWNED}).
\text{PROVE: } \vdots \vdash \Phi; R' \vdash \text{done}(\text{mem\_ptr}, \text{Owned}(\tau)(\text{mem\_ptr}): ret \Rightarrow ret).
\item \text{1. } \vdash \text{mem\_ptr} \Rightarrow \text{pointer} \text{ by } \text{PURE\_VAL\_OBJ\_Ptr} \text{ and } \text{PURE\_VAL\_OBJ}.
\item \text{2. } \text{smt}(\cdot \Rightarrow \text{term}) \text{ by construction of } \text{mem\_ptr}.\)
\item \text{3. } \vdots \vdots \vdash R' \vdash \text{Owned}(\tau)(\text{mem\_ptr}) \Leftarrow \text{pt} \text{ by } \text{RES\_SYN\_PRED}.
\item \text{4. } \text{Prove typing with } \text{EXPL\_SPINE\_RET}; \langle 2 \rangle 3 \sim \langle 2 \rangle 1 \text{ with } \text{EXPL\_SPINE\_RES}, \text{EXPL\_SPINE\_PHI}, \text{EXPL\_SPINE\_COMP} \text{ respectively; } \text{EXPL\_IS\_TVAL}.
\item \text{1. } \text{CASE: } \text{EXPL\_IS\_ACTION\_LOAD}.
\text{LET: } ret = \Sigma y: \beta_{\tau}. y = pval \land (\text{mem\_ptr} \stackrel{\text{const}\_true}{\rightarrow_{\tau}} pval) \ast I
\end{enumerate}
\[ pt = \text{Owned}(\tau)(\text{mem_ptr})(\text{oarg}) \text{ where } \text{oarg} = \{\text{init} = \text{const, true, value} = \text{pval}\}. \]

**Assume:** \[ \vdash; \Phi; \mathcal{R} \vdash \text{load}(\tau, \text{mem_ptr}, \_, \_, \text{res_term}) \Rightarrow \text{ret} \]
and \[ \text{heap} = \text{heap}' + \{\text{pt} \& \text{None}\} \text{ so} \]
\[ \langle h + f; \text{res_term} \rangle \Downarrow \langle \text{heap}' + \{\text{pt} \& \text{None}\}; \text{Owned}(\tau)(\text{mem_ptr}) \rangle. \]

**Let:** \( h' \) and \( \mathcal{R}' \) be as per lemma B8.8 (Resource term reduction is isolated).
\( \mathcal{R}' = \_\text{pt} \) by lemma B8.3 and \( h' = \{\text{pt} \& \text{None}\} \) by lemma B8.4, hence \( \text{heap}' = f \).
This means \( \text{heap} = h' + f, \Phi \vdash h' \leftarrow \mathcal{R}' \) and \( \vdash; \Phi; \mathcal{R}' \vdash \text{Owned}(\tau)(\text{mem_ptr}) \Rightarrow \text{pt} \).

**Prove:** \[ \vdash; \Phi; \mathcal{R}' \vdash \text{done}(\text{pval}, \text{Owned}(\tau)(\text{mem_ptr})); \text{ret} \Rightarrow \text{ret}. \]

\( 2.1. \cdot \vdash \text{pval} \Rightarrow \beta_r \) by lemma B10.1 \( \text{(Owned}(\tau) \text{ resource output values have type } \beta_r) \).

\( 2.2. \cdot \vdash \text{smt}(\cdot \Rightarrow \text{pval} = \text{pval}) \text{ trivially}. \)

\( 2.3. \vdash; \Phi; \mathcal{R}' \vdash \text{Owned}(\tau)(\text{mem_ptr}) \Rightarrow \text{pt}, \text{already established}. \)

\( 2.4. \text{Prove typing with } \text{EXPL\_SPINE\_RET}; \langle 2.3 - 2.1 \rangle \text{ with } \text{EXPL\_SPINE\_RES}, \text{EXPL\_SPINE\_LOG}, \text{EXPL\_SPINE\_COMP} \text{ respectively}; \text{EXPL\_IS\_TVal}. \)

(1) 10. **Case:** \text{EXPL\_IS\_ACTION\_STORE}.

**Let:** \( \text{ret} = \Sigma \_\_\_\text{unit.}(\text{mem_ptr} \stackrel{\text{const, true}}{\Rightarrow} \text{pval})* \_\_\_ \cdot I. \)
\[ pt = \text{Owned}(\tau)(\text{mem_ptr})(\_), \text{pt}' = \text{Owned}(\tau)(\text{mem_ptr})(\text{oarg}), \text{where} \]
\[ \text{oarg} = \{\text{init} = \text{const, true, value} = \text{pval}\}. \]

**Assume:** \[ \vdash; \Phi; \mathcal{R} \vdash \text{store}(\_, \tau, \text{mem_ptr}, \text{pval}, \_, \_\_, \text{res_term}) \Rightarrow \text{ret} \]
and \[ \text{heap} = \text{heap}' + \{\text{pt}' \& \text{None}\} \text{ so} \]
\[ \langle h + f; \text{res_term} \rangle \Downarrow \langle \text{heap}' + \{\text{pt} \& \text{None}\}; \text{Owned}(\tau)(\text{mem_ptr}) \rangle. \]

\[ \exists h'', \mathcal{R}'' \text{ such that } \text{heap}' + \{\text{pt} \& \text{None}\} = h'' + f, \Phi \vdash h'' \leftarrow \mathcal{R}'' \]
and \[ \vdash; \Phi; \mathcal{R}'' \vdash \text{Owned}(\tau)(\text{mem_ptr}) \Rightarrow \text{pt}, \text{by lemma B8.8 (Resource term reduction is isolated)}. \]

\( \mathcal{R}'' = \_\text{pt} \) by lemma B8.3 and \( h'' = \{\text{pt} \& \text{None}\} \) by lemma B8.4, hence \( \text{heap}' = f \).
**Let:** \( h' = \{\text{pt}' \& \text{None}\} \) and \( \mathcal{R}' = \_\text{pt}' \).
This means \( \text{heap} = h' + f \) and \( \Phi \vdash h' \leftarrow \mathcal{R}' \) (by \text{HEAP\_PRED\_OWNED}).

**Prove:** \[ \vdash; \Phi; \mathcal{R}' \vdash \text{done}(\text{Unit}, \text{Owned}(\tau)(\text{mem_ptr})); \text{ret} \Rightarrow \text{ret}. \]

\( 2.1. \cdot \vdash \text{Unit} \Rightarrow \text{unit} \) by \text{PURE\_VAL\_UNIT}. \]

\( 2.2. \vdash; \Phi; \cdot \vdash \text{pt}' \leftarrow \text{Owned}(\tau)(\text{mem_ptr}) \Leftarrow \text{pt}' \) by \text{RES\_SYN\_PRED}. \]

\( 2.3. \text{Prove typing with } \text{EXPL\_SPINE\_RET}; \langle 2.2 - 2.1 \rangle \text{ with } \text{EXPL\_SPINE\_RES}, \text{EXPL\_SPINE\_COMP} \text{ respectively}; \text{EXPL\_IS\_TVal}. \)

(1) 11. **Case:** \text{EXPL\_IS\_ACTION\_KILL\_STATIC}.

**Assume:** \[ \vdash; \Phi; \mathcal{R} \vdash \text{kill}(\text{static } \tau, \text{mem_ptr}, \text{res_term}) \Rightarrow \Sigma \_\_\_\_\text{unit.} \_\_\_ \cdot I \text{ and} \]

\( 40 \)
\langle h + f; \text{res\_term} \rangle \Downarrow \langle \text{heap} + \{pt \& \text{None}\}; \text{Owned} \langle \tau \rangle (\text{mem\_ptr}) \rangle.

\exists h'', R'' \text{ such that } \text{heap} + \{pt \& \text{None}\} = h'' + f, \Phi \vdash h'' \Leftarrow R'' \text{ and } ::; \Phi; R'' \vdash \text{Owned} \langle \tau \rangle (\text{mem\_ptr}) \Rightarrow pt, \text{ by lemma B8.8 (Resource term reduction is isolated).}

R'' = \omega pt \text{ by lemma B8.3 and } h'' = \{pt \& \text{None}\} \text{ by lemma B8.4, hence heap} = f.
Let: h' = \cdot \text{ and } R' = \cdot.
This means heap = h' + f \text{ and } \Phi \vdash h' \Leftarrow R' \text{ (by HEAP\_EMPTY).}

\text{PROVE: } ::; \Phi; \vdash \text{done} (\langle \text{Unit} \rangle; \Sigma, \text{unit}; I \Rightarrow \Sigma, \text{unit}; I
\text{PROOF: } \text{By EXPL\_SPINE\_RET, PURE\_VAL\_UNIT, EXPL\_SPINE\_COMP, EXPL\_IS\_TVAL.}

(1)12. \text{CASE: EXPL\_IS\_MEMOP\_REL\_BINOP.}
\text{PROOF: Similar to PURE\_EXPR\_REL\_BINOP.}

(1)13. \text{CASE: EXPL\_IS\_MEMOP\_INTFROMPTR.}
Let: ret = \Sigma y: \text{integer}. y = \text{cast\_ptr\_to\_int} \text{mem\_ptr} \wedge I. Since \Phi \vdash h \Leftarrow \cdot \text{, } h = \cdot,
\text{heap} = f.
\text{ASSUME: } ::; \Phi; \vdash \text{intFromPtr}(t_1, t_2, \text{mem\_ptr}) \Rightarrow ret.
\text{Let: } h' = \cdot \text{ and } R' = \cdot, \text{ so heap} = h' + f \text{ trivially and } \Phi \vdash \cdot \Leftarrow \cdot \text{ (by HEAP\_EMPTY).}
\text{PROVE: } ::; ::; ::; \vdash \text{done} (\langle \text{mem\_int}\rangle; \text{ret} \Rightarrow ret
\text{PROOF: } \text{Prove typing with EXPL\_SPINE\_RET, EXPL\_SPINE\_PHI, EXPL\_SPINE\_COMP and EXPL\_TOP\_VAL\_DONE instead.}

(1)14. \text{CASE: EXPL\_IS\_MEMOP\_PTRFROMINT.}
\text{PROOF: Similar to EXPL\_IS\_MEMOP\_INTFROMPTR, swapping base types \text{integer} and \text{pointer}.}

(1)15. \text{CASE: EXPL\_IS\_MEMOP\_PTRVALIDFORDEREF.}
Let: pt = \text{Owned} \langle \tau \rangle (\text{mem\_ptr})(oarg) \text{ where } oarg = \{\text{init} = \text{const, true, value} = \text{value}\}
ret = \Sigma y: \text{bool}. y = \text{aligned}(\tau, \text{peal}) \wedge pt \ast I
\text{ASSUME: } ::; \Phi; R \vdash \text{ptrValidForDeref}(\tau, \text{mem\_ptr}, \text{res\_term}) \Rightarrow ret
\text{and } \text{heap} = \text{heap}' + \{pt \& \text{None}\} \text{ so }
\langle h + f; \text{res\_term} \rangle \Downarrow \langle \text{heap}' + \{pt \& \text{None}\}; \text{Owned} \langle \tau \rangle (\text{mem\_ptr}) \rangle.

\text{LET: } h' \text{ and } R' \text{ be as per lemma B8.8 (Resource term reduction is isolated).}
\text{R'} = \omega pt \text{ by lemma B8.3 and } h' = \{pt \& \text{None}\} \text{ by lemma B8.4, hence heap}' = f.
This means heap} = h' + f, \Phi \vdash h' \Leftarrow R' \text{ and } ::; \Phi; R' \vdash \text{Owned} \langle \tau \rangle (\text{mem\_ptr}) \Rightarrow pt.
\text{PROVE: } ::; ::; \Phi; R' \vdash \text{done} (\langle \text{bool\_value}, \text{Owned} \langle \tau \rangle (\text{mem\_ptr}) \rangle; \text{ret} \Rightarrow ret.

(2)1. \ast \vdash \text{bool\_value} \Rightarrow \text{bool} \text{ by PURE\_VAL\_TRUE/ PURE\_VAL\_FALSE.}

(2)2. \text{smt}(\ast \Rightarrow \text{bool\_value} = \text{aligned}(\tau, \text{mem\_ptr})).
\text{PROOF: } \text{By construction of } \text{bool\_value} \text{ (inversion on the transition).}
1. \begin{align*}
\langle \langle \langle & 1 \rangle \rangle \rangle 18. \quad \text{Case: } \text{EXPL-IS_MEMOP_PtrWellAligned.} \\
\text{LET: } & \text{ret} = \Sigma y: \text{bool. } y = \text{aligned}(\tau, \text{mem_ptr}) \land I. \\
\text{ASSUME: } & \vdash_\cdot \Phi; \cdot \vdash \text{ptrWellAligned}(\tau, \text{mem_ptr}) \Rightarrow \text{ret}. \\
\text{Since } & \Phi \vdash h \Leftarrow \cdot, h = \cdot, \text{heap} = f. \\
\text{LET: } & h' = \cdot \text{ and } R' = \cdot, \text{so heap} = h' + f \text{ trivially and } \Phi \vdash \cdot \Leftarrow \cdot \text{ (by } \text{HEAP_EMPTY).} \\
\text{PROVE: } & \vdash_\cdot \Phi; \cdot \vdash \text{done}(\text{bool_value}): \text{ret} \Rightarrow \text{ret}. \\
2. \vdash_\cdot \text{bool_value} \Rightarrow \text{bool} \text{ by } \text{PURE_VAL_TRUE/ PURE_VAL_FALSE.} \\
2. \text{smt} (\cdot \Rightarrow \text{bool_value} = \text{aligned}(\tau, \text{mem_ptr})) \text{ by construction of } \text{bool_value}. \\
2. \text{Prove typing with EXPL_SPINE_RET, EXPL_SPINE_PHI, EXPL_SPINE_COMP, EXPL_IS_TVAL.} \\
\end{align*}

2. \begin{align*}
\text{(1) 16. Case: } \text{EXPL-IS_MEMOP_PtrWellAligned.} \\
\text{LET: } & \text{ret} = \Sigma y: \text{bool. } y = \text{aligned}(\tau, \text{mem_ptr}) \land I. \\
\text{ASSUME: } & \vdash_\cdot \Phi; \cdot \vdash \text{ptrWellAligned}(\tau, \text{mem_ptr}) \Rightarrow \text{ret}. \\
\text{PROVE: } & \vdash_\cdot \Phi; \cdot \vdash \text{done}(\text{bool_value}): \text{ret} \Rightarrow \text{ret}. \\
\end{align*}

(2) \begin{align*}
\text{(1) 17. Case: } \text{EXPL-IS_MEMOP_PtrArrayShift.} \\
\text{PROOF: Similar to } \text{PURE_EXPR_ARRAY_SHIFT, but with EXPL-IS_TVAL.} \\
\end{align*}

2. \begin{align*}
\text{1.8. Case: } \text{EXPL_SEQ_CCall.} \\
\text{ASSUME: } & \text{ident}: \text{fun} \equiv \overline{x_i} \mapsto \text{teexpr} \in \text{Globals} \\
& \vdash_\cdot \Phi; R \vdash \text{spine_elem}_i \mapsto \text{fun} \Rightarrow \text{ret}. \\
& \Phi \vdash h \Leftarrow R \\
& \langle h + f; \text{ccall}(\tau, \text{ident}, \text{spine_elem}_i) \rangle \Rightarrow \langle \text{heap}; \sigma_2(\text{teexpr}); \text{ret}' \rangle \\
& C; L; \Phi''; R'' \vdash \text{teexpr} \Leftarrow \text{ret}'' \text{ where } \overline{x_i} \mapsto \text{fun} \Rightarrow C; L; \Phi''; R'' \Rightarrow \text{ret}''.
\end{align*}

\begin{align*}
\text{PROVE: } & \exists h', \Phi', R' \text{ such that} \\
& \vdash_\cdot \Phi; \cdot \vdash \cdot \vdash_\cdot \Phi'; \cdot \\
& \text{heap} = h' + f \\
& \Phi' \vdash h' \Leftarrow R' \\
& \text{and } \vdash_\cdot \Phi'; R' \vdash \sigma_2(\text{teexpr}) \Leftarrow \text{ret}. \\
\end{align*}

2. \begin{align*}
\text{(1) 1. } C; L; \Phi; \Phi''; R'' \vdash \text{teexpr} \Leftarrow \text{ret}'' \\
& \Phi, \sigma_2(\Phi'') \vdash h \Leftarrow R'' \\
& \vdash_\cdot \Phi; \sigma_2(\Phi''); R \vdash \text{spine_elem}_i \mapsto \text{fun} \Rightarrow \text{ret}. \\
\text{PROOF: By lemma B6 (Weakening).} \\
\end{align*}

\begin{align*}
\text{(2) 2. } \text{ret} = \text{ret}' = \sigma_2(\text{ret}'' \wedge \exists h'_1, R'_1.} \\
& \text{heap} = h'_1 + f, \text{ and } (\Phi, \sigma_2(\Phi'') \vdash h'_1 \Leftarrow R'_1) \\
& \vdash_\cdot \Phi; \sigma_2(\Phi''); R'_1 \vdash \sigma(\text{teexpr}) \Leftarrow \sigma(\text{ret}''). \\
\text{PROOF: By lemma B9.3 (Well-typed spines produce substitutions and the same return type).} \\
\end{align*}

\begin{align*}
\text{(2) 3. } & \vdash_\cdot \Phi; \sigma_2(\Phi''); R'_1 \vdash \sigma(\text{teexpr}) \Leftarrow \sigma(\text{ret}''). \\
\text{PROOF: By lemma B7.3 (Substitution), because } \sigma_2(\Phi) = \Phi \text{ since it contains only closed terms / is well-formed w.r.t.; .} \\
\end{align*}
(2.4) Let: \( h' = h'_1 \); \( \Phi' = \Phi, \sigma_2(\Phi''); \mathcal{R}' = \mathcal{R}'_1 \).

(2.5) \( \vdash \vdash \Phi; :: \vdash \vdash \Phi, \sigma_2(\Phi''); :: \) trivially.

(1.19) **Case:** EXPL\_SEQ\_PROC.
**Proof:** Similar to EXPL\_SEQ\_PROC.

(1.20) **Case:** EXPL\_IS\_MEMOP, EXPL\_IS\_ACTION, EXPL\_IS\_NEG\_ACTION.
**Proof:** By induction.

(1.21) **Case:** EXPL\_TOP\_SEQ\_LET, EXPL\_TOP\_SEQ\_LET\_P, EXPL\_TOP\_SEQ\_LET\_TP, EXPL\_TOP\_SEQ\_LET.
**Proof:** See EXPL\_TOP\_SEQ\_LET\_T for a more general case and proof.

(1.22) **Case:** EXPL\_TOP\_SEQ\_LET\_T.
**Assume:** \( \vdash \vdash \mathcal{R}_1, \mathcal{R}_2 \vdash \text{let } \text{ret}\_pat_1 \vdash \text{ret}_1 = \text{done } \langle \text{ret}\_\text{term}_{i}' \rangle \text{ in } texpr_2 \Leftarrow \text{ret}_2 \)

so \( \vdash \vdash \Phi; \mathcal{R}_1 \vdash \text{done } \langle \text{ret}\_\text{term}_{i}' \rangle \Leftarrow \text{ret}_1 \)
and \( \Phi \vdash \text{ret}\_\text{pat}_1 : \text{ret}_1 \sim C_3; L_3; \Phi_3; \mathcal{R}_3 \)
and \( C_3; L_3; \Phi, \Phi_3; \mathcal{R}_2, \mathcal{R}_3 \vdash texpr \Leftarrow \text{ret}_2 \) (by inversion).

\( \Phi \vdash h \Leftarrow \mathcal{R}_1, \mathcal{R}_2 \) so \( h = h_1 + h_2 \) where \( \Phi \vdash h_1 \Leftarrow \mathcal{R}_1 \) and \( \Phi \vdash h_2 \Leftarrow \mathcal{R}_2 \) by lemma B8.4 (Normalised resource context determines structure of heap).

\( \langle h + f; \text{let } \text{ret}\_\text{pat}_1 \vdash \text{ret}_1 = \text{done } \langle \text{ret}\_\text{term}_{i}' \rangle \text{ in } texpr \rangle \rightarrow \langle \text{heap}; \sigma(texpr) \rangle \)

where \( \langle h; \text{ret}\_\text{pat}_1 = \text{ret}\_\text{term}_{i}' \rangle \sim \langle \text{heap}; \sigma \rangle \).

**Prove:** \( \exists \Phi'; h', \mathcal{R}' . \)

\( \vdash \vdash \Phi; :: \vdash \vdash \Phi'; \)

\( h_{heap} = h' + f \) and \( \Phi' \vdash h' \Leftarrow \mathcal{R}' \)

\( \vdash \vdash \Phi'; \mathcal{R}' \vdash \sigma(texpr_2) \Leftarrow \sigma(\text{ret}_2) \).

\( \exists \mathcal{R}_1'. h_{heap} = h'_1 + h_2 + f \)
\( \Phi \vdash h'_1 \Leftarrow \mathcal{R}'_1 \) and \( \vdash \vdash \Phi; \mathcal{R}_1' \vdash \sigma \left( \langle \text{C}'; \text{L}'_i; \mathcal{R}' \rangle \right) \)

by lemma B9.4 (Well-typed values pattern-match successfully).

This means \( \vdash \vdash [\text{id}, \sigma] (\Phi, \Phi_3); \mathcal{R}_1', \mathcal{R}_2 \vdash [\text{id}, \sigma] \Leftarrow (\text{C}'_i; \text{L}'_i; \mathcal{R}_2, \mathcal{R}') \) by lemma B6 (Weakening).

**Let:** \( \Phi' = \Phi, \sigma(\Phi_3) \), \( h' = h'_1 + h_2 \) and \( \mathcal{R}' = \mathcal{R}'_1, \mathcal{R}_2 \).
By lemma B7.3 (Substitution), because \( \sigma(\Phi) = \Phi \) since it contains only closed terms / is well-formed w.r.t :: .

(1.23) **Case:** EXPL\_TOP\_SEQ\_LET\_T.
**Assume:** \( \vdash \vdash \Phi; \mathcal{R}_1, \mathcal{R}_2 \vdash \text{let } \text{ret}\_\text{pat}_1 \vdash \text{ret}_1 = texpr_1 \text{ in } texpr_2 \Leftarrow \text{ret}_2 \)

so \( \vdash \vdash \Phi; \mathcal{R}_1 \vdash \text{ret}_1 \Leftarrow \text{ret}_1 \)
and \( h = h_1 + h_2 \) where \( \Phi \vdash h_1 \Leftarrow \mathcal{R}_1 \) and \( \Phi \vdash h_2 \Leftarrow \mathcal{R}_2 \) by lemma B8.4 (Normalised resource context determines structure of heap).

\( \langle h; texpr_1 \rangle \rightarrow \langle \text{heap}; texpr_1' \rangle \).

Proceed by induction, instantiating the frame from the inductive hypothesis with \( h_2 + f \).
1) 24. **Case: EXPL_Top_SEQ_CASE.**

   **Assume:**
   
   \[
   \vdash \cdot ; \Phi ; R \vdash \text{case } pval \text{ of } \left\{ \begin{array}{l}
pat_i \Rightarrow \text{texpr}_i \\
pat_j \Rightarrow \text{texpr}_j
\end{array} \right. \text{ end } \Leftarrow \text{ret}^i
   \]

   \[
   \vdash \cdot ; \Phi ; C_i \vdash \text{term}_i
   \]

   \[
   \vdash \cdot ; \Phi ; \text{term}_i = pval ; R \vdash \text{texpr}_i \Leftarrow \text{ret}^i.
   \]

   \[
pat_j = pval \sim \sigma_j \text{ and } \forall i < j. \text{not}(pat_i = pval \sim \sigma_i).
   \]

   **Let:** \( \Phi' = \Phi, \sigma_j(\text{term}_j = pval), h' = h \text{ and } R' = R. \)

   \[
   \vdash \cdot ; \Phi' ; [\text{id}, \sigma_j] \Leftarrow (C_j ; \cdot ; R) \text{ by lemma B9.4 (Well-typed values pattern-match successfully)} \text{ and lemma B6 (Weakening)}. \]

   Hence \( \vdash \cdot ; \Phi ; \cdot \sqsubseteq ; ; ; \Phi' ; \cdot \text{ and } \vdash ; ; \Phi' ; R \vdash \sigma_j(\text{texpr}_j) = \sigma_j(\text{ret}) \text{ by lemma B7.3 (Substitution)}. \)

1) 25. **Case: EXPL_Top_SEQ_IF.**

   See EXPL_Top_SEQ_CASE for more general case and proof.

1) 26. **Case: EXPL_Top_SEQ_Run.**

   **Proof:** Similar to EXPL_SEQ_CCALL.

1) 27. **Case: EXPL_Top_SEQ_Bound.**

   **Proof:** By induction.

1) 28. **Case: EXPL_Top_IS_LetS.**

   **Proof:** Similar to EXPL_Top_SEQ_LetT.