View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by ScholarlyCommons@Penn

University of Pennsylvania
ScholarlyCommons

Technical Reports (CIS)

Department of Computer & Information Science

1-1-2013

Core Ironclad

Peter-Michael Osera
University of Pennsylvania, posera@cis.upenn.edu

Richard A. Eisenberg
University of Pennsylvania, eir@cis.upenn.edu

Christian DeLozier
University of Pennsylvania, delozier@cis.upenn.edu

Santosh Nagarakatte
University of Pennsylvania, santoshn@seas.upenn.edu

Milo Martin
University of Pennsylvania, milom@cis.upenn.edu

See next page for additional authors

Follow this and additional works at: https://repository.upenn.edu/cis_reports

Recommended Citation

Peter-Michael Osera, Richard A. Eisenberg, Christian DeLozier, Santosh Nagarakatte, Milo Martin, and Stephan A. Zdancewic, "Core Ironclad", . January 2013.

This paper is posted at ScholarlyCommons. https://repository.upenn.edu/cis_reports/984
For more information, please contact repository@pobox.upenn.edu.

Core Ironclad

Abstract

Core Ironclad is a core calculus that models the salient features of Ironclad C++, a library-augmented type-safe subset of C++. We give an overview of the language including its definition and key design points. We then prove type safety for the language and use that result to show that the pointer lifetime invariant, a key property of Ironclad C++, holds within the system.

Author(s)

Peter-Michael Osera, Richard A. Eisenberg, Christian DeLozier, Santosh Nagarakatte, Milo Martin, and Stephan A. Zdancewic

This technical report is available at ScholarlyCommons: https://repository.upenn.edu/cis_reports/984

Core Ironclad

CIS Technical Report #MS-CIS-13-06

Peter-Michael Osera
posera@cis.upenn.edu

Richard Eisenberg
eir@cis.upenn.edu

Christian DeLozier
delozier@cis.upenn.edu

Santosh Nagarakatte
santoshn@cis.upenn.edu

Milo M. K. Martin
milom@cis.upenn.edu

Steve Zdancewic
stevez@cis.upenn.edu

July 30, 2013

Core Ironclad is a core calculus that models the salient features of Ironclad C++, a library-augmented type-safe subset of C++. We give an overview of the language including its definition and key design points. We then prove type safety for the language and use that result to show that the pointer lifetime invariant, a key property of Ironclad C++, holds within the system.

1. Introduction

Ironclad C++ is a library-augmented type-safe subset of C++ that provides efficient memory safety [2]. Ironclad C++ accomplishes this through a collection of smart pointer classes as well as a hybrid static-dynamic checking scheme for pointers to values on the stack. Because of the complexity of C++, we would like to be able to prove that Ironclad C++, indeed, provides the safety guarantees we claim.

However, trying to capture the complete Ironclad C++ system requires that we model much of the C++ language itself. This goal is impractical for our present purposes. Furthermore, type safety in the context of object-oriented programming [1, 4] and memory safety in the presence of pointers [5, 7, 8, 9] have both been thoroughly explored. Thus, our formalism, *Core Ironclad*, focuses instead on the C++ language features that are necessary to show the correctness of Ironclad's *pointer lifetime invariant*, which informally states that a pointer cannot outlive the value that it references.

In summary, we present the following:

- The definition of Core Ironclad, a core calculus designed to capture the essence of the Ironclad C++ system and

Types	τ	$::=$	$\mathsf{ptr}\langle\tau\rangle \mid \mathsf{lptr}\langle\tau\rangle \mid C$
Surface exprs	e	$::=$	$x \mid \mathsf{null} \mid e.x \mid e_1.f(e_2) \mid \mathsf{new}\ C() \mid \&e \mid *e$
Internal exprs	e	$::=$	$\ell \mid \{s; \mathsf{return}\ e\} \mid \mathsf{error}$
Statements	s	$::=$	$e_1 = e_2 \mid s_1; s_2 \mid \mathsf{skip} \mid \mathsf{error}$
Class decls	cls	$::=$	$\mathsf{struct}\ C\ \{\overline{\tau_i\ x_i;}^{\,i} meths\};$
Methods	$meth$	$::=$	$\tau_1\ f(\tau_2\ x)\{\overline{\tau_i\ x_i}^{\,i}; s; \mathsf{return}\ e\}$
Programs	$prog$	$::=$	$\Delta; \mathsf{void}\ \mathsf{main}()\ \{s\}$
Locations	ℓ	$::=$	$x^n@y_1\ ..\ y_m$
Pointer values	pv	$::=$	$\ell \mid \mathsf{bad\_ptr}$
Values	v	$::=$	$\mathsf{ptr}\,(pv) \mid \mathsf{lptr}\,(pv) \mid C$
Store	Σ	$::=$	$\cdot \mid \Sigma\,[\ell \mapsto v]$
Store typing	Ψ	$::=$	$\cdot \mid \Psi\,[\ell : \tau]$
Class context	Δ	$::=$	$\{cls_1\ ..\ cls_m\}$

Figure 1: Core Ironclad Syntax

- A proof that the pointer lifetime invariant holds within Core Ironclad by way of standard type safety results for the system.

Section 2 provides a brief overview of the Core Ironclad language. Section 3 gives insight into the design decisions behind Core Ironclad. Section 4 describes the more complex aspects of Core Ironclad in more detail. Finally, section 5 gives the full proofs of type safety for Core Ironclad and uses those results to show that the pointer lifetime invariant holds within the system. Appendix A contains the complete definition of Core Ironclad as a reference.

2. Language Overview

In the name of simplicity, Core Ironclad omits most language features of C++ that do not directly interact with our pointer lifetime system. For example, inheritance, templates, and overloading do not interact with pointer lifetimes, so they are left out. What is left is a small, C++-like core calculus with just enough features to cover the interesting parts of the pointer lifetime checking system.

2.1. Syntax

Figure 1 gives the syntax of the calculus. Core Ironclad is a statement-and-expression language where values are ptrs and lptrs, along with simple classes and methods. Methods are syntactically required to have a single argument and to return a result. While there is no inheritance, classes are important because the this pointer is a potential source of trouble — as a built-in special form, Ironclad C++ cannot automatically wrap `this` in a smart pointer. Each class has a default, no-argument constructor that initializes its members to be invalid pointers and is invoked when calling new $C()$ or when creating

an object on the stack. There are interesting technical issues that arise with the this pointer within constructors and destructors but they are addressed in prior work [10] and are ultimately orthogonal concerns.

Core Ironclad enforces the pointer lifetime invariant with respect to pointers in the heap and in the stack (collectively, the *store*). Locations ℓ in the store are the combination of a base location x^n coupled with a path $x_1 \,..\, x_m$ in the style of Rossie and Friedman [6], Wasserrab [11], and Ramananandro [10]. The use of paths allows locations to refer to the inner members of a class. For example, consider the following definitions:

$$\mathsf{struct}\ C\ \{\ \mathsf{ptr}\langle C\rangle\ a;\ \};$$
$$\mathsf{struct}\ B\ \{\ C\ c;\ \};$$

A declaration `B x;` in the `main` function creates a `B` object that lives at base location x^1. The location $x^1@c.a$ refers to the a field of the `C` sub-object within `B`.

Base locations in Core Ironclad are interesting because they not only represent a unique location in the store x but also its *position* n in the stack. The stack in Core Ironclad grows with increasing indices. For example, location x^3 sits one stack frame lower than x^4. Data in the heap exists at index 0. This is consistent with the intuition that earlier stack frames outlive later frames; the heap simply outlives all stack frames.

The store Σ in Core Ironclad is a mapping from locations to store values v. The store typing Ψ is an analogous mapping from locations to types. Store values are class tags C or pointer values $\mathsf{lptr}\,(pv)$ and $\mathsf{ptr}\,(pv)$ that may reference live locations or the null location bad_ptr. A class tag is assigned to locations that represent the base of some object. In the above example, the location $x^1@$ (with the empty path) maps to the class tag B and $x^1@c$ maps to the class tag C. In this scheme, an object is defined by all locations that share the prefix of a location with a class tag value. The tags themselves are necessary to facilitate class method lookup.

2.1.1. References

Of note, we elide reference types in Core Ironclad. At a basic level, references are simply sugar for implicitly-used pointers that cannot be re-seated once initialized, *i.e.*, a `const` pointer. However, in C++, reference types are important to the language semantics. Also, in Ironclad C++, we have special rules for validating and dealing with references as they are left unwrapped.

To the first point, the cases in which distinguishing reference types from pointers is necessary do not directly impact memory safety. These cases include special member functions such as copy constructors or overload resolution. To the second point, the rules that we have introduced for references in Ironclad C++ involve straightforward static checks for a number of common cases with a catch-all dynamic check (on returned references) to catch the rest. Such an approach is obviously safe and the value in formally verifying these checks does not outweigh the amount of extra machinery necessary to faithfully model references.

In light of these reasons, we choose to elide reference types from Core Ironclad and instead focus our efforts on the novel ptr/lptr dynamic, instead.

2.2. Semantics

Typing and evaluation of statements and expressions, written

$$\begin{aligned}
\Psi \vdash^{\Delta;n}_{\mathsf{stmt}} s\ \mathsf{ok} &\qquad \text{Statement well-formedness}\\
\Psi \vdash^{\Delta;n}_{\mathsf{exp}} e : \tau &\qquad \text{Expression typing}\\
(\Sigma, s) \xrightarrow[\mathsf{stmt}]{}{}^{n}_{\Delta} (\Sigma', s') &\qquad \text{Statement evaluation}\\
(\Sigma, e) \xrightarrow[\mathsf{exp}]{}{}^{n}_{\Delta} (\Sigma', e') &\qquad \text{Expression evaluation}
\end{aligned}$$

are straightforward in Core Ironclad. However, several aspects of these semantics deserve special mention.

2.2.1. Locations

Core Ironclad is a *location-based* language. That is, rather than expressions evaluating to typical values, expressions evaluate to locations that can either be assigned into or used in a store lookup. The primary motivation for this design decision is to be able to model C++ objects that have temporary, yet stable storage such as those returned from methods. Such objects, while being temporary, can still be the subject of an assignment or mutated via method calls.

This set up also simplifies the evaluation rules in several places. For example, field access simply appends onto the current location's path.

$$\frac{}{(\Sigma, (x^{n'}@\pi).x') \xrightarrow[\mathsf{exp}]{}{}^{n}_{\Delta} (\Sigma, x^{n'}@\pi +\!\!+ x')}\text{EVAL\_EXP\_FLD}$$

Also, Core Ironclad does not need to syntactically distinguish between left-values and right-values since the assignment rule can appropriately use the locations it receives.

2.2.2. The Stack

Because Core Ironclad deals with the stack explicitly, the typing and evaluation judgments all note the current stack frame n. This is important for type-checking and evaluating variables.

$$\frac{\Psi(x^{n}@\,) = \tau}{\Psi \vdash^{\Delta;n}_{\mathsf{exp}} x : \tau}\text{TYPE\_EXP\_VAR}$$

$$\frac{}{(\Sigma, x) \xrightarrow[\mathsf{exp}]{}{}^{n}_{\Delta} (\Sigma, x^{n}@\,)}\text{EVAL\_EXP\_VAR}$$

A variable evaluates to a location in stack frame n where the names of the variable and the location coincide.

Core Ironclad embeds the active call frame within the term language using frame expressions of the form $\{s;\ \mathsf{return}\ e\}$ rather than using continuations or a separate stack

syntax. This is similar in style to Core Cyclone [3]. For example, the (abbreviated) rule for method calls

$$\frac{\begin{array}{l}\vdots \\ \Sigma' = \ldots [this^{n+1}@ \mapsto \mathsf{lptr}\,(\ell_1)]\end{array}}{(\Sigma, \ell_1.f(x_2{}^{n_2}@\pi_2)) \xrightarrow[\mathsf{exp}]{}{}^{n}_{\Delta} (\Sigma', \{s;\ \mathsf{return}\ e\})}\text{EVAL\_EXP\_METH}$$

replaces a method invocation with an appropriate frame expression. While the this pointer cannot be wrapped in a smart pointer in the implementation, the Ironclad validator ensures that the `this` pointer behaves like an `lptr`. Consequently, Core Ironclad treats the this pointer as an lptr rather than a third pointer type distinct from ptr and lptr. The remaining premises (not shown) look up the appropriate method body to invoke and set up the arguments and local variable declarations in the store. Because the method body is evaluated at any index n, we typecheck the method at index 0 (*i.e.*, has no dependence on prior stack frames) and prove a lemma that shows we can lift that result to the required index n.

When the statement of a frame expression steps, the stack count must be one higher than that of the frame expression to reflect the new stack frame:

$$\frac{(\Sigma, s) \xrightarrow[\mathsf{stmt}]{}{}^{n+1}_{\Delta} (\Sigma', s')}{(\Sigma, \{s;\ \mathsf{return}\ e\}) \xrightarrow[\mathsf{exp}]{}{}^{n}_{\Delta} (\Sigma', \{s';\ \mathsf{return}\ e\})}\text{EVAL\_EXP\_BODY\_CONG1}$$

Finally, when a frame expression returns, the frame expression is replaced with the location of the return value.

$$\frac{\begin{array}{l}x^n\ \textbf{fresh for}\ \Sigma \\ \Sigma_2 = \mathsf{copy\_store}(\Sigma, \ell, x^n) \\ \Sigma' = \Sigma\,\Sigma_2 \backslash (n+1)\end{array}}{(\Sigma, \{\mathsf{skip};\ \mathsf{return}\ \ell\}) \xrightarrow[\mathsf{exp}]{}{}^{n}_{\Delta} (\Sigma', x^n@\,)}\text{EVAL\_EXP\_BODY\_RET}$$

The premises copy the return value ℓ into a fresh base location x^n in the caller's frame (taking care to copy additional locations if the returned value is an object) and pop the stack. The result of the method call then becomes that fresh location. Note that no dynamic check is needed here because the type system enforces that the return value cannot be a lptr.

$$\frac{\begin{array}{l}\forall \tau'.\tau \neq \mathsf{lptr}\langle\tau'\rangle \\ \Psi \vdash^{\Delta;n+1}_{\mathsf{stmt}} s\ \mathsf{ok} \qquad\qquad \Psi \vdash^{\Delta;n+1}_{\mathsf{exp}} e : \tau\end{array}}{\Psi \vdash^{\Delta;n}_{\mathsf{exp}} \{s;\ \mathsf{return}\ e\} : \tau}\text{TYPE\_EXP\_BODY}$$

2.2.3. Dynamic Checks

In addition to null checks on dereference, dynamic checks are necessary during pointer assignment to ensure that the pointer lifetime invariant holds. When assigning between

two ptrs, no dynamic check is necessary.

$$\frac{\begin{array}{l}\Sigma(\ell_1) = \mathsf{ptr}\,(pv_1) \qquad \Sigma(\ell_2) = \mathsf{ptr}\,(pv_2) \\ \Sigma' = \Sigma\,[\ell_1 \mapsto \mathsf{ptr}\,(pv_2)]\end{array}}{(\Sigma, \ell_1 = \ell_2) \underset{\mathsf{stmt}}{\longrightarrow}^{n}_{\Delta} (\Sigma', \mathsf{skip})}\text{EVAL\_STMT\_ASSIGN\_PTR\_PTR}$$

The dynamic check when assigning a (non-null) lptr to a ptr verifies that the lptr does indeed point to the heap by checking that the store index of the location referred to by the lptr is 0.

$$\frac{\begin{array}{l}\Sigma(\ell_1) = \mathsf{ptr}\,(pv_1) \qquad \Sigma(\ell_2) = \mathsf{lptr}\,(x^0@\pi) \\ \Sigma' = \Sigma\,[\ell_1 \mapsto \mathsf{ptr}\,(x^0@\pi)]\end{array}}{(\Sigma, \ell_1 = \ell_2) \underset{\mathsf{stmt}}{\longrightarrow}^{n}_{\Delta} (\Sigma', \mathsf{skip})}\text{EVAL\_STMT\_ASSIGN\_PTR\_LPTR}$$

When assigning (non-null) lptrs, the dynamic check ensures that the lptr being assigned to out-lives the location it receives by comparing the appropriate store indices.

$$\frac{\begin{array}{l}\Sigma({x_1}^{n_1}@\pi_1) = \mathsf{lptr}\,(pv_1) \\ \Sigma(\ell_2) = \mathsf{lptr}\,({x_2}^{n_2}@\pi_2) \\ \Sigma' = \Sigma\,[{x_1}^{n_1}@\pi_1 \mapsto \mathsf{lptr}\,({x_2}^{n_2}@\pi_2)] \qquad n_2 \leq n_1\end{array}}{(\Sigma, {x_1}^{n_1}@\pi_1 = \ell_2) \underset{\mathsf{stmt}}{\longrightarrow}^{n}_{\Delta} (\Sigma', \mathsf{skip})}\text{EVAL\_STMT\_ASSIGN\_LPTR\_LPTR}$$

If these dynamic checks fail, we raise an error by evaluating to the error term, e.g., between lptrs:

$$\frac{\begin{array}{l}\Sigma({x_1}^{n_1}@\pi_1) = \mathsf{lptr}\,(pv_1) \qquad n_2 \not\leq n_1 \\ \Sigma(\ell_2) = \mathsf{lptr}\,({x_2}^{n_2}@\pi_2)\end{array}}{(\Sigma, {x_1}^{n_1}@\pi_1 = \ell_2) \underset{\mathsf{stmt}}{\longrightarrow}^{n}_{\Delta} (\Sigma, \mathsf{error})}\text{EVAL\_STMT\_ASSIGN\_LPTR\_LPTR\_ERR}$$

2.2.4. Store Consistency

In addition to typing for statements and expressions, Core Ironclad contains a series of judgments that ensure the well-formedness and consistency of the class context, store typing, and store itself. The relation $\mathsf{wf}\,(\Delta, \Psi, \Sigma, n)$ summarizes these judgments and says that the class context, store typing, and store are all consistent with each other up to stack height n and that no bindings exist above that stack height.

The store consistency judgment is of particular interest because it expresses precisely the key invariants of the Ironclad system. Store consistency boils down to the consistency of the individual bindings of the store. In particular, two of these binding consistency rules for pointers capture these invariants.

The first rule concerns ptrs and requires that the location pointed to by the ptr is on the heap (at index 0).

$$\frac{\begin{array}{l}\Sigma(x^n@\pi) = \mathsf{ptr}\,({x'}^{n'}@\pi') \qquad n' = 0 \\ \Psi({x'}^{n'}@\pi') = \tau\end{array}}{\Psi; \Sigma \vdash_{\mathsf{st1}} x^n@\pi : \mathsf{ptr}\langle\tau\rangle\ \mathsf{ok}}\text{CONS\_BINDING\_PTR}$$

The second rule concerns lptrs and requires that lptrs only exist at base locations without paths (not embedded within a class) and that the location pointed to by a particular lptr is in the same stack frame or a lower one.

$$\frac{\Sigma(x^n@\,) = \mathsf{lptr}\,(x'^{n'}@\pi') \qquad n' \leq n \qquad \Psi(x'^{n'}@\pi') = \tau}{\Psi; \Sigma \vdash_{\mathsf{st1}} x^n@\, : \mathsf{lptr}\langle\tau\rangle\,\mathsf{ok}}\text{CONS\_BINDING\_LPTR}$$

This final property is precisely the pointer lifetime invariant which now has a precise definition in the context of Core Ironclad.

Invariant (Pointer lifetime)**.** *For all bindings of the form* $[x_1{}^{n_1}@\pi_1 \mapsto \mathsf{ptr}\,(\ell)]$ *and* $[x_1{}^{n_1}@\pi_1 \mapsto \mathsf{lptr}\,(\ell)]$ *in* Σ*, if* $\ell = x_2{}^{n_2}@\pi_2$ *(i.e., is non-null) then* $n_2 \leq n_1$*.*

3. Design Decisions

Core Ironclad features a number of design decisions to help it model C-like, imperative code. We discuss these points briefly here.

3.1. Mapping locations with paths

An alternative design of the language would map bare locations to structured values. In such a design, a class object would be stored as is, mapped to a single location. Sub-objects would be accessed by extraction, and an expression of the form $e.x$ would evaluate to some extraction internal form.

Here are some of the factors that influenced this design decision:

- In the current design, copying values is challenging, requiring the two judgments copy_store and copy_types, which appear as the following:

  $$\Sigma' = \mathsf{copy\_store}(\Sigma, \ell, x^n) \qquad \Psi' = \mathsf{copy\_types}(\Psi, \ell, x^n)$$

  These judgments are required whenever a parameter is passed into a method or whenever a value is returned. Each recursively searches through either Σ or Ψ looking for bindings with a location prefixed by ℓ, builds a new location for that binding prefixed by x^n, and then adds the binding to the result. Note that the location being copied to is always a location without a path, so the meta-syntax above contains just x^n not $x^n@\pi$.

  The full definitions of the judgments indicate that, as expected, copying simple ptrs and lptrs is easy and is done in one step; it is copying objects that is difficult. It would be possible to eliminate the possibility of passing or returning objects in this system, but the designers felt that would be too restrictive.

  Note that the copy_types judgment is not needed anywhere in the judgment rules. This is because, in those rules, Ψ is used only for static type-checking, and it is

unnecessary to reason about passing parameters or returning values. However, the ability to copy a part of Ψ in an analogous manner to copying part of Σ is necessary to complete the proof of preservation.

- In the alternate design, updating a field in an object would be challenging. Because the value in the store would be the entire object, a new object would have to be created with just one change. Because objects can contain objects, the judgment implementing this idea would have to be recursive. Furthermore, because the nesting can be arbitrarily deep, defining a small-step operational semantics for this field update operation would be delicate.

- As touched on above, extraction of a field from an object would be challenging in the alternate design. Because all values of evaluation are locations, we would need either an enhanced location that could refer to an internal part of an object, or we would need to create a fresh spot on the store for a sub-object.

In the end, one solution here may not be strictly better than the other.

3.2. Stack Direction

In Core Ironclad, the stack grows up with increasing indices. An alternate design of the language would have the top of the stack be at $n = 0$, with older frames having positive indices. This would have simplified the statement of the evaluation relation and typing judgment, as the parameter n would no longer be necessary. However, it would have made the method call, congruence, and return rules rather more cumbersome. In the current design, a change of stack frame is effected simply by incrementing the index of execution. In the alternate design, it would be necessary to increment every index in Ψ and Σ, making parts of the proofs more cumbersome. It is also interesting to note that the alternate design would feature active stack frames with negative indices — these are the frames inside of method body expressions within the current expression.

This alternative design bears some resemblance to the use of de Bruijn indices, but the exact correspondence and ramifications are not clear.

3.3. Location-based Language

An alternate design of the language would have expressions evaluate directly to more traditional values, instead of locations. In this design, a variable x storing $\mathsf{ptr}\,(y^0@\,)$ would actually evaluate to $\mathsf{ptr}\,(y^0@\,)$ instead of a location $x^n@$ that stores the value $\mathsf{ptr}\,(y^0@\,)$. (Here, we are assuming evaluation is taking place in stack frame n.) This alternate design would require separate evaluation relations for the left-hand-side and the right-hand-side of assignments; left-hand-size expressions would still have to evaluate to a location that can be assigned to.

The chief reason that locations are the values in Core Ironclad is that there needs to be a sensible value for the *this* pointer in a method called using a temporary object. If there were object values, what could *this* be? One possible solution is to have the object values

contain a self-reference, but this would complicate the notion of copying, because the self-reference would change during a copy. Temporaries have not been directly addressed in previous work on formalizing C and C++ which makes our approach particularly novel.

A side-effect of making all expressions evaluate to locations is that there is no longer a need to differentiate left-hand-side from right-hand-side in both evaluation and typing.

3.4. Heap and Stack Unification

This design decision was made mainly for convenience. It is easy to imagine having separate heap, stack, and temporary stores. However, doing so would require many more judgment rules to handle the different choices for where to look up a value. Because the salient difference between the stack and the heap in Core Ironclad is the lifetimes of pointers, choosing the heap simply to be the bottom of the stack works well, for the bottom frame outlives all other frames.

3.5. Local Variable Declarations

Core Ironclad requires all local variables declared in a method to be declared at the top of that method. This is a simplification used to avoid the possibility that a statement modifies the typing environment. It does not affect the expressivity of the language.

4. Language Details

While Core Ironclad is relatively simple on the surface, the system requires a number of technical devices and sub-systems in order to accurately model the capabilities of Ironclad C++. Here we give an overview of these systems before diving into the details of the proofs.

4.1. default and build_types

$$\Sigma = \mathsf{default}_\Delta\langle\tau\rangle(\ell)$$

$$\Psi = \mathsf{build\_types}_\Delta\langle\tau\rangle(\ell)$$

The default and build_types judgments represent deterministic algorithms for building the default store and typing context, respectively, upon declaration of variables. For a pointer, the algorithms create a null pointer value and the correct type. For a class, the algorithms recursively add bindings for all fields within that class. The one interesting detail about these algorithms is in rule AUX_DEFAULT_CLASS: a class *tag* is inserted in the store at the location of the whole object.

$$\frac{\begin{array}{l}\mathsf{struct}\ C\ \{\ fldsmeths\};\in\Delta \\ \overline{\tau_i\ x_i;}^{\,i} = flds \\ \overline{\Sigma_i = \mathsf{default}_\Delta\langle\tau_i\rangle(x^n@\pi +\!\!+ x_i)}^{\,i} \\ \Sigma = [x^n@\pi \mapsto C]\,\overline{\Sigma_i}^{\,i}\end{array}}{\Sigma = \mathsf{default}_\Delta\langle C\rangle(x^n@\pi)}\ \text{AUX\_DEFAULT\_CLASS}$$

For example, say we have the following:

$$\begin{array}{l}\mathsf{struct}\ C\ \{\ \mathsf{ptr}\langle C\rangle\ a; \mathsf{ptr}\langle C\rangle\ b;\ \};\\ \mathsf{ptr}\langle C\rangle\ f(C\ x)\{\ ;\mathsf{skip};\mathsf{return\ null}\}\end{array}$$

If we are evaluating function f in stack frame 1, then the store Σ will look like this:

$$[x^1@\ \mapsto\ C]\,[x^1@a \mapsto \mathsf{ptr}\,(\mathsf{bad\_ptr})]\,[x^1@b \mapsto \mathsf{ptr}\,(\mathsf{bad\_ptr})]$$

The reason that the first binding (*i.e.* the tag) is necessary is to perform correct method lookup if we were to call a method on location $x^1@$.

4.2. copy_store and copy_types

$$\Sigma' = \mathsf{copy\_store}(\Sigma, \ell, base)$$
$$\Psi' = \mathsf{copy\_types}(\Psi, \ell, base)$$

These are described in detail in section 3.1.

4.3. Statement and expression sizes

$$k = |s|$$
$$k = |e|$$

In the proof of preservation, it is necessary to explicitly keep track of the current height of the stack. The necessity arises for two reasons:

- In the small-step operational semantics, a statement or expression steps in a method call when the method body expression steps in the outer method. (This is a straightforward congruence rule for method body expressions.) Showing that the consistency of the expression, store, and typing context are preserved requires that no deletions happen in the store or typing context below the level of the height of the stack. It is not sufficient just to check up to the stack level of the method body expression, because the internal statements and expressions in the body need to retain their locations as well.
- When a method body returns, it is necessary to "pop" the top level off the stack. If the bindings were simply to remain, then it is possible that a future method call would happen at the same index and with the same local variable names as the leftover bindings. The conflict would violate the conditions of some of the lemmas that the proof relies on to show consistency preservation.

For these reasons, there are statement and expression size judgments that count how many stack levels the statement or expression contains. For all statements and expressions other than the method body expression, the size is defined to be the maximum size of the substatements or subexpressions, or 0 if there are no substatements and

subexpressions. In the case of a method body expression, the size is one more than the maximum of the sizes of the method body's statement and expression. In this way, the total number of nested method bodies is counted.

It is worth noting that the well-formedness condition, described fully in section 5.1, ensures that no statement or expression has more than one substatement or subexpression containing a method body expression. Because of this, all but one substatement or subexpression will always have size 0.

4.4. Quotienting

$$\Sigma' = \Sigma \backslash n$$

$$\Psi' = \Psi \backslash n$$

The quotient operation implements "popping" the stack. When a method is done executing, it is necessary to remove all bindings at the height of that method in both the store and typing context. The quotient judgments represent deterministic algorithms that do just that.

The type context quotienting judgment $\Psi' = \Psi \backslash n$ is used only in the proof of preservation; no other judgment refers directly to it.

4.5. Well-formed summary judgment

$$\mathsf{wf}\,(\Delta, \Psi, \Sigma, n)$$

This judgment is used to group together the following consistency judgments for convenience:

$$\frac{\vdash_{\overline{\mathsf{env}}} \Delta\ \mathsf{ok} \qquad \vdash_{\overline{\mathsf{ty}}}^{\Delta;n} \Psi\ \mathsf{ok} \qquad \Psi \vdash_{\overline{\mathsf{st}}} \Sigma\ \mathsf{ok}}{\mathsf{wf}\,(\Delta, \Psi, \Sigma, n)}\ \text{CONS\_WFSUMMARY\_WF}$$

4.6. Type context consistency

$$\vdash_{\overline{\mathsf{ty}}}^{\Delta;k} \Psi\ \mathsf{ok}$$

This judgment ensures the following conditions on Ψ:

- Ψ contains no bindings with an index greater than k.
- All bindings to a class tag C in Ψ are accompanied by bindings for all the fields of C.
- All bindings in Ψ with a non-empty path are accompanied by the binding for the class tag of the enclosing object.
- There are no lptrs with non-empty paths in Ψ.

Because checking an individual binding ($\Psi \vdash^{\Delta;k}_{\mathsf{ty1}} \ell : \tau\,\mathsf{ok}$) may have to look both forward and backward in Ψ to ensure these conditions, the judgment is written in a *two-pass* style, where Ψ is identified with a list of bindings and this entire list is used to check each element in the list. A side effect of this style of definition is that some of the technical lemmas have unusual proofs. In particular, lemma 12 requires deriving a non-trivial induction hypothesis.

4.7. Store consistency

$$\Psi \vdash_{\mathsf{st}} \Sigma\,\mathsf{ok}$$

This judgment ensures the following conditions on Ψ and Σ:

- $dom\,(\Sigma) \subseteq dom\,(\Psi)$
- All locations mapping to $\mathsf{lptr}\,(\ell)$ values have type $\mathsf{lptr}\langle\tau\rangle$, where $\Psi(\ell) = \tau$.
- All locations mapping to $\mathsf{ptr}\,(\ell)$ values have type $\mathsf{ptr}\langle\tau\rangle$, where $\Psi(\ell) = \tau$.
- All locations mapping to a class tag in one environment is tagged correctly in the other.
- There are no lptrs with non-empty paths.
- All lptrs point to locations that will outlive the lptr.
- All ptrs point to the heap.

Essentially, this judgment enforces the pointer lifetime invariant and that the values in the store are well-typed.

This judgment is also written in the same two-pass style as the typing consistency judgment. The judgment over individual bindings is written $\Psi; \Sigma \vdash_{\mathsf{st1}} \ell : \tau\,\mathsf{ok}$.

4.8. The source and wf judgments

$$\mathsf{source}\,(s) \quad \mathsf{source}\,(e) \quad \Delta\,\mathsf{wf} \quad cls\,\mathsf{wf} \quad meth\,\mathsf{wf} \quad s\,\mathsf{wf} \quad e\,\mathsf{wf}$$

These are described fully in section 5.1.

4.9. Type well-formedness

$$\vdash^{\Delta}_{\mathsf{twf}} \tau$$

This judgment simply says that any class name used in a type has been declared in Δ.

4.10. The Assignable relation

$$\tau_1 \Leftarrow \tau_2$$

The operational semantics do not define a behavior for assigning one object to another. This is because that effect could be achieved by element-wise assignment of fields. However, the two pointer types are interchangeable, provided they point to the same target type. The Assignable relation $\tau_1 \Leftarrow \tau_2$ enforces these rules.

4.11. Expression typing

$$\Psi \vdash^{\Delta;n}_{\mathsf{exp}} e : \tau$$

Most of these rules are straightforward. The two rules below are the most interesting:

$$\frac{\Psi(x^{n'}@\pi) = \tau \qquad n' \leq n}{\Psi \vdash^{\Delta;n}_{\mathsf{exp}} x^{n'}@\pi : \tau}\text{TYPE\_EXP\_LOC}$$

This rule contains the condition that $n' \leq n$; in other words, that the referent in the location has a lifetime longer than or equal to that of the current stack frame. The existence of this condition ensures that no pointers to popped stack frames exist. Because there is no typing rule for bad_ptr, this rule also implies that any dereference of null is mal-formed. (The rule TYPE_EXP_NULL types the use, but not dereference, of null.)

$$\frac{\begin{array}{l}\forall \tau'.\tau \neq \mathsf{lptr}\langle\tau'\rangle \\ \Psi \vdash^{\Delta;n+1}_{\mathsf{stmt}} s\ \mathsf{ok} \qquad \Psi \vdash^{\Delta;n+1}_{\mathsf{exp}} e : \tau\end{array}}{\Psi \vdash^{\Delta;n}_{\mathsf{exp}} \{s;\ \mathsf{return}\ e\} : \tau}\text{TYPE\_EXP\_BODY}$$

This is the typing rule for method body expressions. The first premise ensures that the return type is not an lptr. This premise is necessary in showing that popping the stack frame on a method return does not remove any binding that is referred to in the return value. The other two premises show that the inner statement and expression are type-checked at a stack frame one higher than the current stack frame. This is exactly how the tiered stack is implemented in the typing context.

4.12. Statement consistency

$$\Psi \vdash^{\Delta;n}_{\mathsf{stmt}} s\ \mathsf{ok}$$

The statement typing rules are very straightforward. The only rule of note is assignment where we use the Assignable relation $\tau_1 \Leftarrow \tau_2$ to determine if the assignment is valid:

$$\frac{\begin{array}{l}\Psi \vdash^{\Delta;n}_{\mathsf{exp}} e_1 : \tau_1 \\ \Psi \vdash^{\Delta;n}_{\mathsf{exp}} e_2 : \tau_2 \\ \tau_1 \Leftarrow \tau_2\end{array}}{\Psi \vdash^{\Delta;n}_{\mathsf{stmt}} e_1 = e_2\ \mathsf{ok}}\text{TYPE\_STMT\_ASSIGN}$$

4.13. Method consistency

$$C \vdash^{\Delta}_{\mathsf{meth}} meth\,\mathsf{ok}$$

The one rule in this judgment checks that a method declaration is well-typed. The premises build up an appropriate typing context to use in checking the internal statement and expression using build_types. There is also a condition that the return type is not an lptr. The internal statement and expression are typed at stack level 0, though this choice of 0 is rather arbitrary. Lemma 17 says that it is possible to change the stack level to any level n in a method. That lemma is necessary to show that types are preserved when a method call evaluates to a method body expression.

4.14. Class consistency

$$\vdash^{\Delta}_{\mathsf{class}} cls\,\mathsf{ok}$$

The one rule in this judgment checks all the methods in the class and enforces the constraint that there are no lptr fields in a class.

4.15. Class environment consistency

$$\vdash_{\mathsf{env}} \Delta\,\mathsf{ok}$$

The one rule in this judgment checks all of the classes in the environment in the two-pass style described above, in section 4.6. This is because classes may have mutual dependencies. The last premise prohibits the following declarations:

$$\mathsf{struct}\ A\ \{\ B\ b;\ \};$$
$$\mathsf{struct}\ B\ \{\ A\ a;\ \};$$

In real C++, this would lead to an infinitely-sized data structure. In Core Ironclad, the effect of these declarations is to make default and build_types diverge.

4.16. Program consistency

$$\vdash_{\mathsf{prog}} \Delta; \mathsf{void\ main}()\ \{s\}\,\mathsf{ok}$$

This judgment checks to make sure that Δ is consistent and that s is consistent in the context of Δ:

$$\frac{\begin{array}{c}\vdash_{\mathsf{env}} \Delta\,\mathsf{ok} \\ \cdot \vdash^{\Delta;0}_{\mathsf{stmt}} s\,\mathsf{ok}\end{array}}{\vdash_{\mathsf{prog}} \Delta; \mathsf{void\ main}()\ \{s\}\,\mathsf{ok}}\text{TYPE\_PROG\_MAIN}$$

4.17. Statement evaluation

$$(\Sigma, s) \underset{\mathsf{stmt}}{\longrightarrow}^{n}_{\Delta} (\Sigma', s')$$

The rules in this judgment are straightforward. The most interesting details are in the assignment rules, which enforce the usual constraints on lptrs and ptrs — that lptrs point to referents at or below the lptr on the stack and that ptrs point to the heap. In the cases where these requirements are not met, the assignment evaluates to error, the model of an thrown exception in Core Ironclad.

4.18. Expression evaluation

$$(\Sigma, e) \underset{\mathsf{exp}}{\longrightarrow}^{n}_{\Delta} (\Sigma', e')$$

Here we describe the interesting expression evaluation rules:

$$\frac{}{(\Sigma, x) \underset{\mathsf{exp}}{\longrightarrow}^{n}_{\Delta} (\Sigma, x^{n}@\,)}\text{EVAL\_EXP\_VAR}$$

This rule contains the notion of lookup in the active stack frame when evaluating a variable. Note that the index in the resultant location is the index used in the evaluation step.

$$\frac{\begin{array}{l} x^{n} \textbf{ fresh for } \Sigma \\ \Sigma' = \Sigma\,[x^{n}@ \mapsto \mathsf{ptr}\,(\mathsf{bad\_ptr})] \end{array}}{(\Sigma, \mathsf{null}) \underset{\mathsf{exp}}{\longrightarrow}^{n}_{\Delta} (\Sigma', x^{n}@\,)}\text{EVAL\_EXP\_NULL}$$

This rule, along with EVAL_EXP_BODY_RET, EVAL_EXP_NEW, and EVAL_EXP_ADDR demonstrate the use of allocating temporary storage locations in the store for temporary values. They all must allocation a fresh location and then evaluate to that location.

$$\frac{}{(\Sigma, (x^{n'}@\pi).x') \underset{\mathsf{exp}}{\longrightarrow}^{n}_{\Delta} (\Sigma, x^{n'}@\pi +\!\!+ x')}\text{EVAL\_EXP\_FLD}$$

This rule demonstrates the relationship between field extraction and the location-based store. Field extraction is exceedingly simple: just append the field name to the path.

$$\frac{\begin{array}{l} \Sigma(\ell_1) = C \\ \mathsf{struct}\ C\,\{\,fldsmeths\}; \in \Delta \\ \tau_1\ f(\tau_2\ x)\{\overline{\tau_i\ x_i}^{\,i}; s; \mathsf{return}\ e\} \in meths \\ \Sigma_2 = \mathsf{copy\_store}(\Sigma, {x_2}^{n_2}@\pi_2, x^{n+1}) \\ \Sigma_3 = [{this}^{n+1}@ \mapsto \mathsf{lptr}\,(\ell_1)] \\ \overline{\Sigma_{4\,i} = \mathsf{default}_{\Delta}\langle\tau_i\rangle({x_i}^{n+1}@\,)}^{\,i} \\ \Sigma' = \Sigma\,\Sigma_2\,\Sigma_3\,\overline{\Sigma_{4\,i}}^{\,i} \end{array}}{(\Sigma, \ell_1.f({x_2}^{n_2}@\pi_2)) \underset{\mathsf{exp}}{\longrightarrow}^{n}_{\Delta} (\Sigma', \{s;\ \mathsf{return}\ e\})}\text{EVAL\_EXP\_METH}$$

This rule builds up the correct store with which to evaluate the body of a method. It allocates space for the one parameter of the method, the *this* pointer, and all local variables of the method being called. The value of the actual parameter is copied into the location of the formal parameter, *this* is initialized appropriately, and default values are created for the local variables. Note that all of these locations are at index $n+1$, because the locations should be local to the method being called.

$$\frac{\begin{array}{l} x^n \textbf{ fresh for } \Sigma \\ \Sigma_2 = \mathsf{copy\_store}(\Sigma, \ell, x^n) \\ \Sigma' = \Sigma\, \Sigma_2 \backslash (n+1) \end{array}}{(\Sigma, \{\mathsf{skip};\ \mathsf{return}\ \ell\}) \underset{\mathsf{exp}}{\longrightarrow}^{n}_{\Delta} (\Sigma', x^n@\,)}\text{EVAL\_EXP\_BODY\_RET}$$

This rule is triggered when the statement and expression in a method body expression are both done evaluating. It copies the return value into a freshly allocated location, local to the calling method. The premises also indicate that the store is effectively popped — all locations at index $n+1$ are removed from the store because the method at level $n+1$ is done evaluating. Proving that this rule does not violate preservation requires that the returned value does not reference any locations in the $n+1$ stack frame.

$$\frac{(\Sigma, s) \underset{\mathsf{stmt}}{\longrightarrow}^{n+1}_{\Delta} (\Sigma', s')}{(\Sigma, \{s;\ \mathsf{return}\ e\}) \underset{\mathsf{exp}}{\longrightarrow}^{n}_{\Delta} (\Sigma', \{s';\ \mathsf{return}\ e\})}\text{EVAL\_EXP\_BODY\_CONG1}$$

$$\frac{(\Sigma, e) \underset{\mathsf{exp}}{\longrightarrow}^{n+1}_{\Delta} (\Sigma', e')}{(\Sigma, \{\mathsf{skip};\ \mathsf{return}\ e\}) \underset{\mathsf{exp}}{\longrightarrow}^{n}_{\Delta} (\Sigma', \{\mathsf{skip};\ \mathsf{return}\ e'\})}\text{EVAL\_EXP\_BODY\_CONG2}$$

These rules are the congruence rules for evaluating within a method body expression. They are notable because the inner statement or expression is evaluated in stack frame $n+1$. The other congruence rules naturally evaluate their inner expression(s) in frame n.

5. Proofs

Type-safety for Core Ironclad follows from standard progress and preservation lemmas. We first build up the necessary infrastructure and then prove these lemmas in detail. We then use these results to show that the pointer lifetime invariant as discussed in section 2 holds.

5.1. Well-formedness

Because we encode the call stack into our term language with the $\{s;\ \mathsf{return}\ e\}$ block expression, we must restrict our metatheory to well-formed terms. Intuitively, well-formedness for terms implies that we never have multiple, active stack frames in distinct sub-terms. For example, consider the syntactically valid method call:

$$\{\mathsf{skip};\ \mathsf{return}\ x\}.f(\{\mathsf{skip};\ \mathsf{return}\ x\})$$

Here, both the receiver and argument expressions have an active call frame that mentions the variable x. If we are currently at stack depth n then both call frames are at stack depth $n+1$ and thus when we typecheck x in each context, we'll use the same location in the store typing $x^{n+1}@$. This is clearly not correct as the body expressions may have been generated by invocations to different methods and thus the xs may have different types.

This situation should never occur because body expressions are not part of the source language and our left-to-right call-by-value semantics guarantees we fully evaluate the receiver before evaluating the argument.

To enforce this property, we define a source-level and well-formedness judgment over classes, statements, and expressions. Appendix A.4 gives the complete rules of these judgments.

For example, with SWF_ASSIGN1 and SWF_ASSIGN2, the assignment $e_1 = e_2$ is well-formed if (1) e_1 is not a location (i.e., not yet fully evaluated) and e_2 contains no block expressions or (2) e_1 is fully evaluated in which case there is no restriction on e_2. All of syntactic forms that contain sub-expressions or statements are similarly defined.

The important property of well-formedness is that it is closed under evaluation. To prove this, we need a lemma stating the fact that source-level programs are well-formed.

Lemma 1 (Source-level Terms are Well-formed)**.**

1. *If* $\mathsf{source}\,(s)$ *then* $s\,\mathsf{wf}$.
2. *If* $\mathsf{source}\,(e)$ *then* $e\,\mathsf{wf}$.

Proof Sketch: Straightforward induction on the source predicate in each part. source makes a strictly stronger claim that wf: there can be no non-source level subterms within the given term. □

Theorem 1 (Well-formedness Preservation)**.**

1. *If* $\Delta\,\mathsf{wf}$, $s\,\mathsf{wf}$, *and* $(\Sigma, s) \underset{stmt}{\longrightarrow}^{n}_{\Delta} (\Sigma', s')$ *then* $s'\,\mathsf{wf}$.
2. *If* $\Delta\,\mathsf{wf}$, $e\,\mathsf{wf}$, *and* $(\Sigma, e) \underset{exp}{\longrightarrow}^{n}_{\Delta} (\Sigma', e')$ *then* $e'\,\mathsf{wf}$.

Proof Sketch: By a straightforward mutual induction on the two evaluation derivations. The two interesting sets of cases are unsurprisingly when we consider how the term steps via congruence and when we invoke a method. In the case of congruence, we must use Lemma 1 to be able to conclude that the sub-term is well-formed when the premises of well-formedness judgment say that it is source. The case of method invocation is similar, but relies on the fact that since $\Delta\,\mathsf{wf}$ that the method body is source. □

In the interest of brevity, we assume that we are only working with well-formed classes, statements, and expressions for the rest of these proofs.

5.2. Progress

Progress in Core Ironbound is standard. The one caveat is that values in our language exist only in the store, so our lemmas reflect what we can expect when we perform store lookup.

Lemma 2 (Existence of Store Values)**.**

If $\Psi; \Sigma \vdash_{\mathsf{st1}} \ell : \tau\ \mathsf{ok}$ *then* $\Sigma(\ell) = v$ *for some* v*.*

Proof. Immediate by inversion on the judgment $\Psi; \Sigma \vdash_{\mathsf{st1}} \ell : \tau\ \mathsf{ok}$. In all cases, we demand that $\Sigma(\ell) = v$ for some v. □

Lemma 3 (Canonical Forms of Store Values)**.**

1. *If* $\Psi \vdash^{\Delta;n}_{\mathit{exp}} \ell : C$ *and* $\Psi \vdash_{\mathit{st}} \Sigma\ \mathsf{ok}$ *then* $\Sigma(\ell) = C$*.*
2. *If* $\Psi \vdash^{\Delta;n}_{\mathit{exp}} \ell : \mathsf{ptr}\langle\tau\rangle$ *and* $\Psi \vdash_{\mathit{st}} \Sigma\ \mathsf{ok}$ *then* $\Sigma(\ell) = \mathsf{ptr}\,(\ell')$ *for some* ℓ'*.*
3. *if* $\Psi \vdash^{\Delta;n}_{\mathit{exp}} \ell : \mathsf{lptr}\langle\tau\rangle$ *and* $\Psi \vdash_{\mathit{st}} \Sigma\ \mathsf{ok}$ *then* $\Sigma(\ell) = \mathsf{lptr}\,(\ell')$ *for some* ℓ'*.*

Proof. By inversion on the typing judgment to obtain that $\Psi(\ell) = \tau$. Because $\Psi \vdash_{\mathsf{st}} \Sigma\ \mathsf{ok}$ we know that $\Psi; \Sigma \vdash_{\mathsf{st1}} \ell : \tau\ \mathsf{ok}$. By inversion on the binding consistency judgment, we can conclude in each of the three cases that the store produces the appropriate value. □

Theorem 2 (Progress)**.**

1. *If* $\Psi \vdash^{\Delta;n}_{\mathit{stmt}} s\ \mathsf{ok}$ *and* $\Psi \vdash_{\mathit{st}} \Sigma\ \mathsf{ok}$ *then* s *is* skip*,* error*, or* $(\Sigma, s) \xrightarrow[\mathit{stmt}]{}{}^{n}_{\Delta} (\Sigma', s')$*.*
2. *If* $\Psi \vdash^{\Delta;n}_{\mathit{exp}} e : \tau$ *and* $\Psi \vdash_{\mathit{st}} \Sigma\ \mathsf{ok}$ *then* e *is* ℓ*,* error*, or* $(\Sigma, e) \xrightarrow[\mathit{exp}]{}{}^{n}_{\Delta} (\Sigma', e')$*.*

Proof. By mutual induction on the typing derivations of s and e.
First consider the cases for typing s.

Case STMT_ASSIGN The derivation ends in

$$\frac{\begin{array}{l}\Psi \vdash^{\Delta;n}_{\mathsf{exp}} e_1 : \tau_1 \\ \Psi \vdash^{\Delta;n}_{\mathsf{exp}} e_2 : \tau_2 \\ \tau_1 \Leftarrow \tau_2\end{array}}{\Psi \vdash^{\Delta;n}_{\mathsf{stmt}} e_1 = e_2\ \mathsf{ok}}\text{TYPE\_STMT\_ASSIGN}$$

By inversion on the typing judgment we know that e_1 and e_2 are well-typed. By the induction hypothesis, e_1 is error, steps, or is some ℓ_1. If e_1 is error then s steps by EVAL_STMT_ASSIGN_ERR1. If e_1, then s steps by EVAL_STMT_ASSIGN_CONG1.

If e_1 is some ℓ_1 then by the induction hypothesis, e_2 is error, steps, or is some ℓ_2. If e_2 is error then s steps by EVAL_STMT_ASSIGN_ERR2. If e_2 steps, then s steps by EVAL_STMT_ASSIGN_CONG2.

If $s = \ell_1 = \ell_2$ then by the typing for ℓ_1 and ℓ_2 and Lemma 2 we know that $\Sigma(\ell_1) = v_1$ and $\Sigma(\ell_2) = v_2$ for some v_1 and v_2. By inversion of the assignable judgment $\tau_1 \Leftarrow \tau_2$ we can refine v_1 and v_2 to the following cases:

- $v_1 = \mathsf{ptr}\,(pv_1)$ and $v_2 = \mathsf{ptr}\,(pv_2)$. s steps by EVAL_STMT_ASSIGN_PTR_PTR.
- $v_1 = \mathsf{lptr}\,(pv_1)$ and $v_2 = \mathsf{ptr}\,(pv_2)$. s by EVAL_STMT_ASSIGN_LPTR_PTR.
- $v_1 = \mathsf{ptr}\,(pv_1)$ and $v_2 = \mathsf{lptr}\,(pv_2)$. If $pv_2 = \mathsf{bad\_ptr}$ then s steps by EVAL_STMT_ASSIGN_PTR_LPTR_NULL. If pv_2 is some heap value then it steps by EVAL_STMT_ASSIGN_PTR_LPTR. Else s steps by EVAL_STMT_ASSIGN_PTR_LPTR_ERR.
- $v_1 = \mathsf{lptr}\,(pv_1)$ and $v_2 = \mathsf{lptr}\,(pv_2)$. If $pv_2 = \mathsf{bad\_ptr}$ then s steps by EVAL_STMT_ASSIGN_LPTR_LPTR_NULL. If pv_2 is some heap value then it steps by EVAL_STMT_ASSIGN_LPTR_LPTR. Else s steps by EVAL_STMT_ASSIGN_LPTR_LPTR_ERR.

Case STMT_SEQ The derivation ends in

$$\frac{\Psi \vdash^{\Delta;n}_{\mathsf{stmt}} s_1\ \mathsf{ok} \quad \Psi \vdash^{\Delta;n}_{\mathsf{stmt}} s_2\ \mathsf{ok}}{\Psi \vdash^{\Delta;n}_{\mathsf{stmt}} s_1; s_2\ \mathsf{ok}}\ \text{TYPE\_STMT\_SEQ}$$

By inversion on the typing judgment we know that s_1 and s_2 are well-formed. By the induction hypothesis, s_1 is error, steps, or is skip. If s_1 is error then s steps by EVAL_STMT_SEQ_ERR. If s_1 steps then s steps by EVAL_STMT_SEQ_CONG. Finally if s_1 is skip then s steps by EVAL_STMT_SEQ_SKIP.

Case STMT_SKIP Trivial as s is skip.

Case STMT_ERROR Trivial as s is error.

Next, consider the cases for typing e.

Case EXP_VAR x immediately steps by EVAL_EXP_VAR.

Case EXP_FLD The derivation ends in

$$\frac{\begin{array}{l}\Psi \vdash^{\Delta;n}_{\mathsf{exp}} e_1 : C \\ \mathsf{struct}\ C\ \{\ fldsmeths\}; \in \Delta \\ \tau\ x; \in flds\end{array}}{\Psi \vdash^{\Delta;n}_{\mathsf{exp}} e_1.x : \tau}\ \text{TYPE\_EXP\_FLD}$$

By inversion on the typing judgment, we know that e_1 is well-typed. By the induction hypothesis, e_1 is error, steps, or is some ℓ_1. If e_1 is error, e steps by EVAL_EXP_FLD_ERR. If e_1 steps, then e steps by EVAL_EXP_FLD_CONG. Finally if e_1 is some ℓ_1 then e steps by EVAL_EXP_FLD.

Case EXP_DEREF_PTR The derivation ends in

$$\frac{\Psi \vdash^{\Delta;n}_{\mathsf{exp}} e_1 : \mathsf{ptr}\langle\tau\rangle}{\Psi \vdash^{\Delta;n}_{\mathsf{exp}} *e_1 : \tau}\ \text{TYPE\_EXP\_DEREF\_PTR}$$

By inversion on the typing judgment, we know that e_1 is well-typed. By the induction hypothesis, e_1 is error, steps, or is some ℓ_1. In the first case, e steps by EVAL_EXP_DEREF_CONG. In the second case, e steps by EVAL_EXP_DEREF_ERROR. In the final case, e steps either by EVAL_EXP_DEREF_PTR or EVAL_EXP_DEREF_-PTR_NULL.

Case EXP_DEREF_LPTR Analogous to the EXP_DEREF_PTR case utilizing the lptr rules rather than the ptr rules.

Case EXP_LOC Trivial as e is ℓ.

Case EXP_NULL null immediately steps by EVAL_EXP_NULL.

Case EXP_METH The derivation ends in

$$\frac{\begin{array}{l}\Psi \vdash^{\Delta;n}_{\mathsf{exp}} e_1 : C \\ \mathsf{struct}\ C\ \{\ fldsmeths\}; \in \Delta \\ \tau_1\ f(\tau_2\ x)\{vardecls; s; \mathsf{return}\ e\} \in meths \\ \Psi \vdash^{\Delta;n}_{\mathsf{exp}} e_2 : \tau_2\end{array}}{\Psi \vdash^{\Delta;n}_{\mathsf{exp}} e_1.f(e_2) : \tau_1}\ \text{TYPE\_EXP\_METH}$$

By inversion on the typing judgment, we know that e_1 and e_2 are well-typed. By the induction hypothesis, e_1 is error, steps, or is some ℓ_1. If e_1 is error then e steps by EVAL_EXP_METH_ERR1. If e_1 steps, then e steps by EVAL_EXP_METH_CONG1.

Otherwise, if e_1 is some ℓ_1 then by the induction hypothesis, e_2 is error, steps, or is some ℓ_2. In the first case, e steps by EVAL_EXP_METH_ERR2. In the second case, e steps by EVAL_EXP_METH_CONG2. In the final case, e steps by EVAL_EXP_METH provided that $\Sigma(\ell_1) = C$ which we know holds by Lemmas 2 and 3.

Case EXP_NEW new immediately steps by EVAL_EXP_NEW.

Case EXP_ADDR Analogous to the EXP_DEREF_PTR case utilizing the addr rules rather than the ptr rules.

Case EXP_BODY The derivation ends in

$$\frac{\begin{array}{l}\tau \neq \mathsf{lptr}\langle\tau'\rangle \\ \Psi \vdash^{\Delta;n+1}_{\mathsf{stmt}} s\ \mathsf{ok} \qquad\qquad \Psi \vdash^{\Delta;n+1}_{\mathsf{exp}} e_1 : \tau\end{array}}{\Psi \vdash^{\Delta;n}_{\mathsf{exp}} \{s;\ \mathsf{return}\ e_1\} : \tau}\ \text{TYPE\_EXP\_BODY}$$

By inversion on the typing judgment we know s and e are well-typed. By the induction hypothesis, s is error, steps, or is skip. If s is error then e steps by EVAL_EXP_BODY_ERR1. If s steps, then e steps by EVAL_EXP_BODY_CONG1.

If s is skip then by the induction hypothesis, e_1 is error, steps, or is some ℓ_1. In the first case, e steps by EVAL_EXP_BODY_ERR2. In the second case, e steps by EVAL_EXP_BODY_CONG2. Finally, if $e = \{\mathsf{skip}; \mathsf{return}\,\ell_1\}$ then it steps by EVAL_EXP_BODY_RET.

□

5.3. Preservation

The proof of preservation is relatively straightforward, but it requires a large number of technical lemmas to deal with all of the structure of our judgments. The largest source of novelty in this proof comes from a non-standard subsetting relation that examines only those elements at or below the level of the top of the stack.

Definition (Subsetting)**.** *We define* $\Psi \subseteq_n \Psi'$ *to mean that for every binding of the form* $\Psi(x^{n'}@\pi) = \tau$ *such that* $n' \leq n$*, then* $\Psi'(x^{n'}@\pi) = \tau$*.*

Lemma 4 (ptrs Point to the Heap)**.** *If* $\Psi \vdash_{\overline{st}} \Sigma\,\mathsf{ok}$ *and* $\Sigma(\ell) = \mathsf{ptr}\,(x^n@\pi)$ *then* $n = 0$*.*

Proof. This comes directly from the fact that we require that $n = 0$ to conclude that $\Psi; \Sigma \vdash_{\overline{\mathsf{st1}}} \ell : \mathsf{ptr}\langle\tau\rangle\,\mathsf{ok}$. □

Lemma 5 (No lptrs in Objects)**.** *If* $\Psi \vdash_{\overline{st}} \Sigma\,\mathsf{ok}$ *and* $\Psi(\ell) = \mathsf{lptr}\langle\tau\rangle$*, then* l *has the form* $x^n@$*. In other words, the path associated with* ℓ *is empty.*

Proof. This comes directly from the fact that both cases for $\Psi; \Sigma \vdash_{\overline{\mathsf{st1}}} \ell : \mathsf{lptr}\langle\tau\rangle\,\mathsf{ok}$ require the path to be empty. □

Lemma 6 (lptrs Point Down the Stack)**.** *If* $\Psi \vdash_{\overline{st}} \Sigma\,\mathsf{ok}$ *and* $\Sigma({x_1}^{n_1}@\pi_1) = \mathsf{lptr}\,({x_2}^{n_2}@\pi_2)$*, then* $n_2 \leq n_1$*.*

Proof. The only evidence for $\Psi; \Sigma \vdash_{\overline{\mathsf{st1}}} \ell : \mathsf{lptr}\langle\tau\rangle\,\mathsf{ok}$ where $\Sigma(\ell) \neq \mathsf{lptr}\,(\mathsf{bad\_ptr})$ is CONS_BINDING_LPTR. According to this rule $n_2 \leq n_1$. □

Lemma 7 (Binding Consistency Weakening)**.** *If* $\Psi; \Sigma \vdash_{\overline{st1}} \ell : \tau\,\mathsf{ok}$*, (for all locations* $\ell' \in dom(\Psi)$*,* $\Psi(\ell') = \Psi'(\ell')$*), and (for all locations* $\ell' \in dom(\Sigma)$*,* $\Sigma(\ell') = \Sigma'(\ell')$*), then* $\Psi'; \Sigma' \vdash_{\overline{st1}} \ell : \tau\,\mathsf{ok}$*.*

Proof. Straightforward case analysis on $\Psi; \Sigma \vdash_{\overline{\mathsf{st1}}} \ell : \tau\,\mathsf{ok}$. The rules refer to Ψ and Σ only via lookup operations, and the results of lookup operations do not change by assumption. □

Lemma 8 (Lookup Concatenation)**.** *If for some environments* ϵ *and* ϵ' *(an* ϵ *is either a* Ψ *or a* Σ*),* $dom(\epsilon) \cap dom(\epsilon') = \emptyset$ *and* $\ell \in dom(\epsilon)$*, then* $\epsilon(\ell) = (\epsilon\,\epsilon')(\ell) = (\epsilon'\,\epsilon)(\ell)$*. (*$\epsilon\,\epsilon'$ *denotes the concatenation of* ϵ *and* ϵ'*).*

Proof. By the definition of a lookup operation, adding new elements to an environment at new locations does not change the value of any lookup operations of the original locations. □

Lemma 9 (Concatenation Consistency)**.** *If* $\Psi_1 \vdash_{\overline{st}} \Sigma_1\,\mathsf{ok}$*,* $\Psi_2 \vdash_{\overline{st}} \Sigma_2\,\mathsf{ok}$*,* $dom(\Psi_1) \cap dom(\Psi_2) = \emptyset$*, and* $dom(\Sigma_1) \cap dom(\Sigma_2) = \emptyset$*, then* $\Psi_1\,\Psi_2 \vdash_{\overline{st}} \Sigma_1\,\Sigma_2\,\mathsf{ok}$*.*

Proof. Let $\Psi_1 = \overline{[\ell_{1\,i} : \tau_{1\,i}]}^{\,i}$ and $\Psi_2 = \overline{[\ell_{2\,j} : \tau_{2\,j}]}^{\,j}$. Let $\Psi = \Psi_1\,\Psi_2$ and $\Sigma = \Sigma_1\,\Sigma_2$. Then, we must show that $\Psi; \Sigma \vdash_{\overline{\mathsf{st1}}} \ell_{1\,i} : \tau_{1\,i}\,\mathsf{ok}$ for every i and $\Psi; \Sigma \vdash_{\overline{\mathsf{st1}}} \ell_{2\,j} : \tau_{2\,j}\,\mathsf{ok}$ for every j. We must also show that $dom\,(\Sigma) \subseteq dom\,(\Psi)$.

For some i, consider $\Psi; \Sigma \vdash_{\overline{\mathsf{st1}}} \ell_{1\,i} : \tau_{1\,i}\,\mathsf{ok}$. We know that $\Psi_1; \Sigma_1 \vdash_{\overline{\mathsf{st1}}} \ell_{1\,i} : \tau_{1\,i}\,\mathsf{ok}$. Because the domains of Ψ_1 and Ψ_2 are distinct and the domains of Σ_1 and Σ_2 are distinct, lemma 8 tells us that $\Sigma(\ell) = \Sigma_1(\ell)$ for some $\ell \in dom(\Sigma_1)$ and $\Psi(\ell) = \Psi_1(\ell)$ for some $\ell \in dom(\Psi_1)$. We then use lemma 7 to conclude $\Psi; \Sigma \vdash_{\overline{\mathsf{st1}}} \ell_{1\,i} : \tau_{1\,i}\,\mathsf{ok}$ for all i, as desired.

A similar argument gives us $\Psi; \Sigma \vdash_{\overline{\mathsf{st1}}} \ell_{2\,j} : \tau_{2\,j}\,\mathsf{ok}$ for all j.

The subset relationship holds from the fact that $(A \subseteq C, B \subseteq D)$ implies $(A \cup B) \subseteq (C \cup D)$. □

Lemma 10 (Type Binding Consistency Weakening)**.** *If* $\Psi \vdash^{\Delta;k}_{\mathsf{ty1}} \ell : \tau\,\mathsf{ok}$ *and for all locations* $\ell' \in dom(\Psi)$, $\Psi(\ell') = \Psi'(\ell')$, *then* $\Psi' \vdash^{\Delta;k}_{\mathsf{ty1}} \ell : \tau\,\mathsf{ok}$.

Proof. Straightforward case analysis on $\Psi \vdash^{\Delta;k}_{\mathsf{ty1}} \ell : \tau\,\mathsf{ok}$. The rules refer to Ψ only via lookup operations, and the results of lookup operations do not change by assumption. □

Lemma 11 (Type Concatenation Consistency)**.** *If* $\vdash^{\Delta;k}_{\mathsf{ty}} \Psi_1\,\mathsf{ok}$, $\vdash^{\Delta;k}_{\mathsf{ty}} \Psi_2\,\mathsf{ok}$, *and* $dom(\Psi_1) \cap dom(\Psi_2) = \emptyset$, *then* $\vdash^{\Delta;k}_{\mathsf{ty}} \Psi_1\,\Psi_2\,\mathsf{ok}$.

Proof. The proof follows much like that of lemma 9, but without the subset requirement. We use lemmas 8 and 10. □

Lemma 12 (Max Stack Height)**.** *If* $\vdash^{\Delta;k}_{\mathsf{ty}} \Psi\,\mathsf{ok}$, *then every element in* $dom(\Psi)$ *has an index* n *such that* $n \leq k$.

Proof. We invert $\vdash^{\Delta;k}_{\mathsf{ty}} \Psi\,\mathsf{ok}$ to get $\Psi \vdash^{\Delta;k}_{\mathsf{ty1}} {x_i}^{n_i}@\pi_i : \tau_i\,\mathsf{ok}$ for every binding $[{x_i}^{n_i}@\pi_i : \tau_i]$ in Ψ. We consider a specific binding i and proceed by induction on π_i.

In the base case, where π_i is empty, we know we are in case CONS_TYPE_BINDING_LPTR, CONS_TYPE_BINDING_LOCAL_PTR, or CONS_TYPE_BINDING_LOCAL_CLASS. All of these cases require $n_i \leq k$, so we are done.

In the inductive case, we let $\pi_i = \pi +\!\!+ x$. The inductive hypothesis states that if ${x_i}^{n_i}@\pi \in dom(\Psi)$ and $\Psi \vdash^{\Delta;k}_{\mathsf{exp}} {x_i}^{n_i}@\pi : \tau$ (for some τ), then $n \leq k$. We must be in case CONS_TYPE_BINDING_FIELD_PTR or CONS_TYPE_BINDING_FIELD_CLASS. By inversion of $\Psi \vdash^{\Delta;k}_{\mathsf{ty1}} {x_i}^{n_i}@\pi_i : \tau_i\,\mathsf{ok}$, we can see that ${x_i}^{n_i}@\pi$ must be in $dom(\Psi)$. Similarly, by inversion of $\vdash^{\Delta;k}_{\mathsf{ty}} \Psi\,\mathsf{ok}$ and the fact that $[{x_i}^{n_i}@\pi : \tau]$ is a binding in Ψ, we can derive $\Psi \vdash^{\Delta;k}_{\mathsf{ty1}} {x_i}^{n_i}@\pi : \tau\,\mathsf{ok}$. We can now use the induction hypothesis to get $n \leq k$ as desired. □

Lemma 13 (default Consistency)**.** *If* $\vdash_{\overline{env}} \Delta\,\mathsf{ok}$, $\Psi = \mathsf{build\_types}_\Delta\langle\tau\rangle(\ell)$ *and* $\Sigma = \mathsf{default}_\Delta\langle\tau\rangle(\ell)$, *then* $\Psi \vdash_{\overline{st}} \Sigma\,\mathsf{ok}$ *for all* τ *and* ℓ.

Proof. By induction on the derivation of default.

Case CONS_DEFAULT_PTR We know that $\tau = \mathsf{ptr}\langle\tau'\rangle$ and thus $[\ell \mapsto \mathsf{ptr}\,(\mathsf{bad\_ptr})] = \mathsf{default}_\Delta\langle\mathsf{ptr}\langle\tau'\rangle\rangle(\ell)$ and $[\ell : \mathsf{ptr}\langle\tau'\rangle] = \mathsf{build\_types}_\Delta\langle\mathsf{lptr}\langle\tau\rangle\rangle(\ell)$. We must show that $[\ell : \mathsf{ptr}\langle\tau'\rangle] \vdash_{\overline{\mathsf{st}}} [\ell \mapsto \mathsf{ptr}\,(\mathsf{bad\_ptr})]\,\mathsf{ok}$ which is immediate by CONS_BINDING_PTR_NULL.

Case CONS_DEFAULT_LPTR We know that $\tau = \mathsf{lptr}\langle\tau'\rangle$ and thus $[\ell \mapsto \mathsf{lptr}\,(\mathsf{bad\_ptr})] = \mathsf{default}_\Delta\langle\mathsf{lptr}\langle\tau'\rangle\rangle(\ell)$ and $[\ell : \mathsf{lptr}\langle\tau'\rangle] = \mathsf{build\_types}_\Delta\langle\mathsf{lptr}\langle\tau'\rangle\rangle(\ell)$. We must show that $[\ell : \mathsf{lptr}\langle\tau'\rangle] \vdash_{\overline{\mathsf{st}}} [\ell \mapsto \mathsf{lptr}\,(\mathsf{bad\_ptr})]\,\mathsf{ok}$ which is immediate by CONS_BINDING_LPTR_NULL.

Case CONS_DEFAULT_CLASS We know that $\tau = C$. Let the fields of C be $\overline{\tau_i\ x_i;}^{\,i}$. Since $\overline{\Psi_i = \mathsf{build\_types}_\Delta\langle\tau_i\rangle(x^n@\pi +\!\!+ x_i)}^{\,i}$ and $\overline{\Sigma_i = \mathsf{default}_\Delta\langle\tau_i\rangle(x^n@\pi +\!\!+ x_i)}^{\,i}$ then by induction $\overline{\Psi_i \vdash_{\overline{\mathsf{st}}} \Sigma_i\,\mathsf{ok}}^{\,i}$. By CONS_BINDING_CLS $[x^n@\pi : C] \vdash_{\overline{\mathsf{st}}} [x^n@\pi \mapsto C]\,\mathsf{ok}$ so putting this together with Lemma 9, $[x^n@\pi : C]\,\overline{\Psi_i}^{\,i} \vdash_{\overline{\mathsf{st}}} [x^n@\pi \mapsto C]\,\overline{\Sigma_i}^{\,i}\,\mathsf{ok}$. Note that we know the domains of the individual pieces are disjoint because we only append onto paths and the field names must be unique, by $\vdash_{\overline{\mathsf{env}}} \Delta\,\mathsf{ok}$.

□

Lemma 14 ($\mathsf{copy\_types}$ Copies)**.** *If* $\Psi(x^n@\pi_1 +\!\!+ \pi_2) = \tau$ *and* $\Psi' = \mathsf{copy\_types}(\Psi, x^n@\pi_1, {x'}^{n'})$, *then* $\Psi'({x'}^{n'}@\pi_2) = \tau$.

Proof. Straightforward induction on $\mathsf{copy\_types}$. □

Lemma 15 ($\mathsf{copy\_store}$ Copies)**.** *If* $\Sigma(x^n@\pi_1 +\!\!+ \pi_2) = v$ *and* $\Sigma' = \mathsf{copy\_store}(\Sigma, x^n@\pi_1, {x'}^{n'})$, *then* $\Sigma'({x'}^{n'}@\pi_2) = v$.

Proof. Straightforward induction on $\mathsf{copy\_store}$. □

Lemma 16 (Copy Consistency)**.** *If* $\Psi \vdash_{\overline{\mathsf{st}}} \Sigma\,\mathsf{ok}$, $\Psi' = \mathsf{copy\_types}(\Psi, \ell_1, {x_2}^{n_2})$, $\Sigma' = \mathsf{copy\_store}(\Sigma, \ell_1, {x_2}^{n_2})$, ${x_2}^{n_2}@$ *is not the prefix of any location in dom*(Ψ), *and (if* $\Sigma(\ell_1) = \mathsf{lptr}\,({x_3}^{n_3}@\pi_3)$ *then* $n_3 \leq n_2$*), then* $\Psi\,\Psi' \vdash_{\overline{\mathsf{st}}} \Sigma\,\Sigma'\,\mathsf{ok}$.

Proof. We must show the following:

- $\Psi\,\Psi'; \Sigma\,\Sigma' \vdash_{\overline{\mathsf{st1}}} \ell_i : \tau_i\,\mathsf{ok}$ for every binding $[\ell_i : \tau_i]$ in $\Psi\,\Psi'$: A binding $[\ell_i : \tau_i]$ is either in Ψ or Ψ'. A binding may not be in both because of the condition that ${x_2}^{n_2}@$ is not a prefix of any location in the domain of Ψ. This gives us two cases:
  - $[\ell_i : \tau_i] \in \Psi$: We use lemma 7 on $\Psi; \Sigma \vdash_{\overline{\mathsf{st1}}} \ell_i : \tau_i\,\mathsf{ok}$, which we got from inversion on $\Psi \vdash_{\overline{\mathsf{st}}} \Sigma\,\mathsf{ok}$. The condition on lookup operations is satisfied because of the domain condition described above.
  - $[\ell_i : \tau_i] \in \Psi'$: By easy induction on the definitions of $\mathsf{copy\_types}$ and $\mathsf{copy\_store}$, we can see that every domain element ℓ_i has the form ${x_2}^{n_2}@\pi_i$, for some path (possibly empty) π_i. We use lemma 15 to conclude that $\Sigma'({x_2}^{n_2}@\pi_i) = \Sigma({x_1}^{n_1}@\pi_1 +\!\!+ \pi_i)$.

    The inversion of $\Psi \vdash_{\overline{\mathsf{st}}} \Sigma\,\mathsf{ok}$ gives us $\Psi; \Sigma \vdash_{\overline{\mathsf{st1}}} {x_1}^{n_1}@\pi_1 +\!\!+ \pi_i : \tau_i\,\mathsf{ok}$. We first use lemma 7 to get $\Psi\,\Psi'; \Sigma\,\Sigma' \vdash_{\overline{\mathsf{st1}}} {x_1}^{n_1}@\pi_1 +\!\!+ \pi_i : \tau_1\,\mathsf{ok}$. We then invert this judgment and can change a lookup operation $(\Sigma\,\Sigma')({x_1}^{n_1}@\pi_1 +\!\!+ \pi_i)$ to be $(\Sigma\,\Sigma')({x_2}^{n_2}@\pi_i)$. Noting that Ψ does not change and the lookup on Ψ in the CONS_BINDING_LPTR and CONS_BINDING_PTR cases remains the same, we now

have all the premises for the different cases fulfilled except for the $n' \leq n_2$ premise in CONS_BINDING_LPTR. In this specific case, we use lemma 5 to find that π_i is the empty path. We then know $\Sigma(\ell_1) = \mathsf{lptr}\,(x_3{}^{n_3}@\pi_3)$ and therefore, by assumption, (identifying n_3 in the statement of this lemma with n' in the statement of the rule) $n' \leq n_2$ as desired.

- $dom\,(\Sigma\,\Sigma') \subseteq dom\,(\Psi\,\Psi')$: We know that $dom\,(\Sigma) \subseteq dom\,(\Psi)$ from inversion on $\Psi \models_{\mathsf{st}} \Sigma\,\mathsf{ok}$. We must show that $dom\,(\Sigma') \subseteq dom\,(\Psi')$. This fact can be seen from induction on copy_types and copy_store — whenever a domain element is added to Σ', that same domain element is added to Ψ', knowing that $dom\,(\Sigma) \subseteq dom\,(\Psi)$.

□

Lemma 17 (Stack Change)**.**

1. *If* $\Psi \vdash^{\Delta;n}_{stmt} s\,\mathsf{ok}$*, every element in* $dom(\Psi)$ *has index* n*, and* $\mathsf{source}\,(s)$*, then* $\Psi' \vdash^{\Delta;n'}_{stmt} s\,\mathsf{ok}$ *where* Ψ' *is identical to* Ψ *except all the indices in the domain are changed from* n *to* n'*.*

2. *If* $\Psi \vdash^{\Delta;n}_{exp} e : \tau$*, every element in* $dom(\Psi)$ *has index* n*, and* $\mathsf{source}\,(e)$*, then* $\Psi' \vdash^{\Delta;n'}_{exp} e : \tau$ *where* Ψ' *is identical to* Ψ *except all the indices in the domain are changed from* n *to* n'*.*

Proof Sketch: By mutual induction on typing derivations. The interesting cases are TYPE_EXP_VAR and TYPE_EXP_LOC. TYPE_EXP_VAR is straightforward. TYPE_EXP_LOC can't happen by hypothesis. □

Lemma 18 (Subset Weakening)**.**

1. *If* $\Psi \subseteq \Psi'$*, then* $\Psi \subseteq_n \Psi'$ *for any* $n \geq 0$*.*

2. *If* $\Psi \subseteq'_n \Psi'$ *and* $n \leq n'$*, then* $\Psi \subseteq_n \Psi'$*.*

Proof. Immediate from the definition of $\subseteq_n$. □

Lemma 19 (Subset Transitivity)**.** *If* $\Psi_1 \subseteq_n \Psi_2$ *and* $\Psi_2 \subseteq_n \Psi_3$*, then* $\Psi_1 \subseteq_n \Psi_3$*.*

Proof. Immediate from the definition of $\subseteq_n$ and the transitivity of $\subseteq$. □

Lemma 20 (Weakening)**.**

1. *If* $\Psi \vdash^{\Delta;n}_{stmt} s\,\mathsf{ok}$*,* $\mathsf{source}\,(s)$*, and* $dom(\Psi) \cap dom(\Psi') = \emptyset$*, then* $\Psi\,\Psi' \vdash^{\Delta;n}_{stmt} s\,\mathsf{ok}$*.*

2. *If* $\Psi \vdash^{\Delta;n}_{exp} e : \tau$*,* $\mathsf{source}\,(e)$*, and* $dom(\Psi) \cap dom(\Psi') = \emptyset$*, then* $\Psi\,\Psi' \vdash^{\Delta;n}_{exp} e : \tau$*.*

Proof. By straightforward mutual induction on typing derivations. □

Lemma 21 (Stack Height Weakening)**.** *If* $\vdash^{\Delta;k}_{ty} \Psi\,\mathsf{ok}$ *and* $k' \geq k$*, then* $\vdash^{\Delta;k'}_{ty} \Psi\,\mathsf{ok}$*.*

Proof. By straightforward case analyses on the premises of $\vdash^{\Delta;k}_{\mathsf{ty}} \Psi\,\mathsf{ok}$, noting that k appears only as an upper bound in the premises of $\Psi \vdash^{\Delta;k}_{\mathsf{ty1}} \ell : \tau\,\mathsf{ok}$. □

Lemma 22 (Stack Height Strengthening)**.** *If* $\vdash^{\Delta;k}_{\mathsf{ty}} \Psi\,\mathsf{ok}$*, and every domain element of* Ψ *has an index less than or equal to some* k'*, then* $\vdash^{\Delta;k'}_{\mathsf{ty}} \Psi\,\mathsf{ok}$*.*

Proof. By straightforward case analyses on the premises of $\vdash^{\Delta;k}_{\mathsf{ty}} \Psi\,\mathsf{ok}$. □

Lemma 23 ($\mathsf{build\_types}$ Unique)**.** *If* $\Psi = \mathsf{build\_types}_\Delta\langle\tau\rangle(\ell)$ *and* $\Psi' = \mathsf{build\_types}_\Delta\langle\tau\rangle(\ell)$*, then* $\Psi = \Psi'$*.*

Proof. Straightforward induction on $\Psi = \mathsf{build\_types}_\Delta\langle\tau\rangle(\ell)$. □

Lemma 24 ($\mathsf{build\_types}$ Consistent (Inductive Case))**.** *If* $\Psi = \mathsf{build\_types}_\Delta\langle\tau\rangle(x^n@\pi +\!\!+ x')$*,* $\vdash^{\Delta}_{\mathsf{twf}} \tau$*,* $n \leq k$*,* $\mathsf{struct}\ C\,\{\,fldsmeths\};\in \Delta$*, and* $\tau\,x';\in flds$ *then* $[x^n@\pi : C]\,\Psi \vdash^{\Delta;k}_{\mathsf{ty1}} x^n@\pi +\!\!+ x' : \tau\,\mathsf{ok}$*.*

Proof Sketch: By induction on the definition of $\mathsf{build\_types}$. □

Lemma 25 ($\mathsf{build\_types}$ Consistent)**.** *If* $\Psi = \mathsf{build\_types}_\Delta\langle\tau\rangle(x^n@\,)$*,* $\vdash^{\Delta}_{\mathsf{twf}} \tau$*, and* $n \leq k$*, then* $\vdash^{\Delta;k}_{\mathsf{ty}} \Psi\,\mathsf{ok}$*.*

Proof Sketch: Straightforward case analysis on the definition of $\mathsf{build\_types}$, invoking lemma 24 in case AUX_BUILD_TYPES_CLASS. □

Lemma 26 (Size of source)**.**

1. *If* $\mathsf{source}\,(s)$ *then* $0 = |s|$*.*
2. *If* $\mathsf{source}\,(e)$ *then* $0 = |e|$*.*

Proof. Straightforward mutual induction on the definition of size of statements and expressions. □

Lemma 27 (Copy/Build Consistency)**.** *If* $\vdash^{\Delta;k}_{\mathsf{ty}} \Psi''\,\mathsf{ok}$*,* $\Psi = \mathsf{build\_types}_\Delta\langle\tau\rangle(x^n@\,)$*,* $\Psi' = \mathsf{copy\_types}(\Psi'', x'^{n'}@\pi', x^n)$*,* $\Psi''(\ell) = \tau$*,* $\vdash^{\Delta}_{\mathsf{twf}} \tau$*, and* $n' \leq k$*, then* $\Psi = \Psi'$*.*

Proof Sketch: We show $\Psi \subseteq \Psi'$ and $\Psi' \subseteq \Psi$.

- $\Psi \subseteq \Psi'$: By induction over the definition of $\mathsf{build\_types}$.
- $\Psi' \subseteq \Psi$: By induction over the definition of $\mathsf{copy\_types}$.

□

Lemma 28 (Quotienting)**.**

1. *If* $\Sigma' = \Sigma\backslash n$*, then there are no domain elements in* Σ' *with an index equal to* n*.*

2. *If* $\Psi' = \Psi\backslash n$*, then there are no domain elements in* Ψ' *with an index equal to* n*.*

Proof. Straightforward induction on the length of Σ and Ψ. □

Lemma 29 (Quotient Consistency)**.** *If* $\Psi \vdash_{\overline{st}} \Sigma\,\mathsf{ok}$*,* $\Psi' = \Psi\backslash n$*,* $\Sigma' = \Sigma\backslash n$*, and there exist no domain elements of* Ψ *with an index greater than* n*, then* $\Psi' \vdash_{\overline{st}} \Sigma'\,\mathsf{ok}$*.*

Proof Sketch: Straightforward case analysis of the elements in Ψ'. We will need lemmas 4 and 6 for ptrs and lptrs, respectively. □

Lemma 30 (Quotient Distribution)**.**

1. *If* $\Sigma' = \Sigma_1\,\Sigma_2\backslash n$*, then* $\Sigma' = \Sigma_1\,\backslash n\,\Sigma_2\backslash n$*.*
2. *If* $\Psi' = \Psi_1\,\Psi_2\backslash n$*, then* $\Psi' = \Psi_1\,\backslash n\,\Psi_2\backslash n$*.*

Proof. Straightforward induction on the length of Σ_2 and Ψ_2. □

Lemma 31 (Quotient Preserves Others)**.**

1. *If* $\Sigma' = \Sigma\backslash n$*,* $\Sigma(x^{n'}@\pi) = v$*, and* $n' \neq n$*, then* $\Sigma'(x^{n'}@\pi) = v$*.*
2. *If* $\Psi' = \Psi\backslash n$*,* $\Psi(x^{n'}@\pi) = \tau$*, and* $n' \neq n$*, then* $\Psi'(x^{n'}@\pi) = \tau$*.*

Proof. Straightforward induction on the length of Σ or Ψ. □

Lemma 32 (Quotient Preserves Subset)**.** *If* $\Psi \subseteq_n \Psi'$ *and* $n' > n$*, then* $\Psi \subseteq_n \Psi'\backslash n'$*.*

Proof. By use of lemma 31 at all levels i such that $i \leq n$. □

Lemma 33 (One Change in Consistency)**.** *If* $\Psi \vdash_{\overline{st}} \Sigma\,\mathsf{ok}$*,* $\ell \in dom(\Psi)$*, and* $\Psi; \Sigma\,[\ell \mapsto v] \vdash_{\overline{st1}} \ell : \tau\,\mathsf{ok}$*, then* $\Psi \vdash_{\overline{st}} \Sigma\,[\ell \mapsto v]\,\mathsf{ok}$*.*

Proof. The only change from $\Psi \vdash_{\overline{\mathsf{st}}} \Sigma\,\mathsf{ok}$ is the entry for ℓ in Σ. Because $\Sigma(\ell)$ for some Σ and ℓ is mentioned only in the case for $\Psi; \Sigma \vdash_{\overline{\mathsf{st1}}} \ell : \tau\,\mathsf{ok}$, we must know only that $\Psi; \Sigma\,[\ell \mapsto v] \vdash_{\overline{\mathsf{st1}}} \ell : \tau\,\mathsf{ok}$, for $\Psi(\ell) = \tau$, to conclude that $\Psi \vdash_{\overline{\mathsf{st}}} \Sigma\,[\ell \mapsto v]\,\mathsf{ok}$. □

Lemma 34 (Type Binding Consistency Weakening)**.** *If* $\Psi \vdash^{\Delta;k}_{\overline{ty1}} \ell : \tau\,\mathsf{ok}$ *and* $\ell' \notin dom(\Psi)$*, then* $\Psi\,[\ell' : \tau'] \vdash^{\Delta;k}_{\overline{ty1}} \ell : \tau\,\mathsf{ok}$*.*

Proof. Straightforward case analysis on $\Psi \vdash^{\Delta;k}_{\overline{\mathsf{ty1}}} \ell : \tau\,\mathsf{ok}$. The rules refer to Ψ only via lookup operations, and the results of lookup operations do not change with a fresh binding added. □

Theorem 3 (Preservation)**.**

1. *If* $\mathsf{wf}\,(\Delta, \Psi, \Sigma, |s| + n)$*,* $\Psi \vdash^{\Delta;n}_{\overline{stmt}} s\,\mathsf{ok}$*, and* $(\Sigma, s) \xrightarrow[stmt]{n}_{\Delta} (\Sigma', s')$*, then there exists* Ψ' *such that* $\mathsf{wf}\,(\Delta, \Psi', \Sigma', |s| + n)$*,* $\Psi \vdash^{\Delta;n}_{\overline{stmt}} s'\,\mathsf{ok}$*, and* $\Psi \subseteq_n \Psi'$*.*

2. *If* $\mathsf{wf}\,(\Delta, \Psi, \Sigma, |e| + n)$, $\Psi \vdash^{\Delta;n}_{exp} e : \tau$, *and* $(\Sigma, e) \xrightarrow[exp]{}{}^{n}_{\Delta} (\Sigma', e')$, *then there exists* Ψ' *such that* $\mathsf{wf}\,(\Delta, \Psi', \Sigma', |e| + n)$, $\Psi' \vdash^{\Delta;n}_{exp} e' : \tau$, *and* $\Psi \subseteq_n \Psi'$.

Proof. We proceed by mutual induction on the evaluation derivations from each portion of the theorem.

Case EVAL_STMT_ASSIGN_PTR_PTR

$$\frac{\begin{array}{ll}\Sigma(\ell_1) = \mathsf{ptr}\,(pv_1) & \Sigma(\ell_2) = \mathsf{ptr}\,(pv_2) \\ \Sigma' = \Sigma\,[\ell_1 \mapsto \mathsf{ptr}\,(pv_2)] & \end{array}}{(\Sigma, \ell_1 = \ell_2) \xrightarrow[\mathsf{stmt}]{}{}^{n}_{\Delta} (\Sigma', \mathsf{skip})}\text{EVAL\_STMT\_ASSIGN\_PTR\_PTR}$$

Choose $\Psi' = \Psi$.

We must show the following:

- $\vdash^{\Delta;n}_{\mathsf{ty}} \Psi\,\mathsf{ok}$: By assumption.
- $\Psi \vdash_{\mathsf{st}} \Sigma'\,\mathsf{ok}$: Using the subsetting relation we get by inversion on $\Psi \vdash_{\mathsf{st}} \Sigma\,\mathsf{ok}$, we see that $\ell \in dom(\Psi)$. We use lemma 33 to show that we need to show only that $\Psi; \Sigma' \vdash_{\mathsf{st1}} \ell_1 : \tau\,\mathsf{ok}$. From inversion on $\Psi; \Sigma \vdash_{\mathsf{st1}} \ell_1 : \tau\,\mathsf{ok}$, knowing that $\Sigma(\ell_1) = \mathsf{ptr}\,(pv_1)$, we see that $\tau = \mathsf{ptr}\langle\tau'\rangle$ for some τ'. We have two cases here: either $pv_2 = \mathsf{bad\_ptr}$ or $pv_2 = x^{n'}@\pi$. In the first case, we can derive $\Psi; \Sigma' \vdash_{\mathsf{st1}} \ell_1 : \mathsf{ptr}\langle\tau'\rangle\,\mathsf{ok}$ directly from CONS_BINDING_PTR_NULL. In the second case, we use inversion on $\Psi; \Sigma \vdash_{\mathsf{st1}} \ell_2 : \tau_2\,\mathsf{ok}$, knowing that $\Sigma(\ell_2) = \mathsf{ptr}\,(x^{n'}@\pi)$ to see that $\tau_2 = \mathsf{ptr}\langle\tau_2'\rangle$, $\Psi(x^{n'}@\pi) = \tau_2'$, and $n' = 0$. Now, we use inversion on $\Psi \vdash^{\Delta;n}_{\mathsf{stmt}} \ell_1 = \ell_2\,\mathsf{ok}$ to see that $\Psi \vdash^{\Delta;n}_{\mathsf{exp}} \ell_1 : \tau$, $\Psi \vdash^{\Delta;n}_{\mathsf{exp}} \ell_2 : \tau_2$, and $\tau \Leftarrow \tau_2$. We have already determined that $\tau = \mathsf{ptr}\langle\tau'\rangle$ and $\tau_2 = \mathsf{ptr}\langle\tau_2'\rangle$. Using inversion on $\mathsf{ptr}\langle\tau'\rangle \Leftarrow \mathsf{ptr}\langle\tau_2'\rangle$ gives us $\tau' = \tau_2'$, by TYPE_ASSIGNABLE_PTR_PTR. Thus, we can conclude $\Psi; \Sigma' \vdash_{\mathsf{st1}} \ell_1 : \mathsf{ptr}\langle\tau'\rangle\,\mathsf{ok}$ as desired.
- $\Psi \vdash^{\Delta;n}_{\mathsf{stmt}} \mathsf{skip}\,\mathsf{ok}$: Immediate from TYPE_STMT_SKIP.
- $\Psi \subseteq_n \Psi$: By reflexivity of $\subseteq_n$.

Case EVAL_STMT_ASSIGN_LPTR_PTR

$$\frac{\begin{array}{ll}\Sigma(\ell_1) = \mathsf{lptr}\,(pv_1) & \Sigma(\ell_2) = \mathsf{ptr}\,(pv_2) \\ \Sigma' = \Sigma\,[\ell_1 \mapsto \mathsf{lptr}\,(pv_2)] & \end{array}}{(\Sigma, \ell_1 = \ell_2) \xrightarrow[\mathsf{stmt}]{}{}^{n}_{\Delta} (\Sigma', \mathsf{skip})}\text{EVAL\_STMT\_ASSIGN\_LPTR\_PTR}$$

$$\frac{\begin{array}{l}\Psi \vdash^{\Delta;n}_{\mathsf{exp}} e_1 : \tau_1 \\ \Psi \vdash^{\Delta;n}_{\mathsf{exp}} e_2 : \tau_2 \\ \tau_1 \Leftarrow \tau_2\end{array}}{\Psi \vdash^{\Delta;n}_{\mathsf{stmt}} e_1 = e_2\,\mathsf{ok}}\text{TYPE\_STMT\_ASSIGN}$$

Choose $\Psi' = \Psi$.

We must show the following:

- $\vdash^{\Delta;n}_{\mathsf{ty}} \Psi\,\mathsf{ok}$: By assumption.
- $\Psi \vdash_{\mathsf{st}} \Sigma'\,\mathsf{ok}$: We proceed as in the case for EVAL_STMT_ASSIGN_PTR_PTR. This case differs in that $\Sigma(\ell_1) = \mathsf{lptr}\,(pv_1)$ and therefore $\tau = \mathsf{lptr}\langle\tau'\rangle$. Later in the proof, we invoke CONS_BINDING_LPTR_NULL instead of CONS_BINDING_PTR_NULL and TYPE_ASSIGNABLE_LPTR_PTR instead of TYPE_ASSIGNABLE_PTR_PTR. Note that the proof above shows that $n' = 0$, where $\Sigma(\ell_2) = \mathsf{ptr}\,(x^{n'}@\pi)$. Because $n \geq 0$ by assumption, we know that $n' \leq n$. The last part to show is that ℓ_1 has the form $x^n@$ — in other words, ℓ_1 has no path. This can be seen by repeated inversion on $\Psi \vdash_{\mathsf{st}} \Sigma\,\mathsf{ok}$, noting that both LPTR cases require that ℓ_1 have no path. We can now conclude that $\Psi; \Sigma' \vdash_{\mathsf{st1}} \ell_1 : \mathsf{lptr}\langle\tau'\rangle\,\mathsf{ok}$, as desired.
- $\Psi \vdash^{\Delta;n}_{\mathsf{stmt}} \mathsf{skip}\,\mathsf{ok}$: Immediate from TYPE_STMT_SKIP.
- $\Psi \subseteq_n \Psi$: By reflexivity of $\subseteq_n$.

Case EVAL_STMT_ASSIGN_PTR_LPTR_NULL

$$\frac{\begin{array}{l}\Sigma(\ell_1) = \mathsf{ptr}\,(pv_1) \qquad \Sigma(\ell_2) = \mathsf{lptr}\,(\mathsf{bad\_ptr}) \\ \Sigma' = \Sigma\,[\ell_1 \mapsto \mathsf{ptr}\,(\mathsf{bad\_ptr})]\end{array}}{(\Sigma, \ell_1 = \ell_2) \underset{\mathsf{stmt}}{\longrightarrow}^n_\Delta (\Sigma', \mathsf{skip})}\text{EVAL\_STMT\_ASSIGN\_PTR\_LPTR\_NULL}$$

Choose $\Psi' = \Psi$.

We must show the following:

- $\vdash^{\Delta;n}_{\mathsf{ty}} \Psi\,\mathsf{ok}$: By assumption.
- $\Psi \vdash_{\mathsf{st}} \Sigma'\,\mathsf{ok}$: As argued above, we need only show that $\Psi; \Sigma' \vdash_{\mathsf{st1}} \ell_1 : \mathsf{ptr}\langle\tau'\rangle\,\mathsf{ok}$. This fact is immediate from $\Sigma'(\ell_1) = \mathsf{ptr}\,(\mathsf{bad\_ptr})$ and CONS_BINDING_PTR_NULL.
- $\Psi \vdash^{\Delta;n}_{\mathsf{stmt}} \mathsf{skip}\,\mathsf{ok}$: Immediate from TYPE_STMT_SKIP.
- $\Psi \subseteq_n \Psi$: By reflexivity of $\subseteq_n$.

Case EVAL_STMT_ASSIGN_PTR_LPTR

$$\frac{\begin{array}{l}\Sigma(\ell_1) = \mathsf{ptr}\,(pv_1) \qquad \Sigma(\ell_2) = \mathsf{lptr}\,(x^0@\pi) \\ \Sigma' = \Sigma\,[\ell_1 \mapsto \mathsf{ptr}\,(x^0@\pi)]\end{array}}{(\Sigma, \ell_1 = \ell_2) \underset{\mathsf{stmt}}{\longrightarrow}^n_\Delta (\Sigma', \mathsf{skip})}\text{EVAL\_STMT\_ASSIGN\_PTR\_LPTR}$$

Choose $\Psi' = \Psi$.

We must show the following:

- $\vdash^{\Delta;n}_{\mathsf{ty}} \Psi\,\mathsf{ok}$: By assumption.
- $\Psi \vdash_{\mathsf{st}} \Sigma'\,\mathsf{ok}$: As argued above, we need only show that $\Psi; \Sigma' \vdash_{\mathsf{st1}} \ell_1 : \mathsf{ptr}\langle\tau'\rangle\,\mathsf{ok}$. This fact is immediate from $\Sigma'(\ell_1) = \mathsf{ptr}\,(x^0@\pi)$ and CONS_BINDING_PTR, using the argument presented above to show that $\tau' = \tau'_2$, where $\Psi \vdash^{\Delta;n}_{\mathsf{exp}} \ell_2 : \mathsf{lptr}\langle\tau'_2\rangle$.

- $\Psi \vdash^{\Delta;n}_{\mathsf{stmt}} \mathsf{skip}\,\mathsf{ok}$: Immediate from TYPE_STMT_SKIP.
- $\Psi \subseteq_n \Psi$: By reflexivity of $\subseteq_n$.

Cases EVAL_STMT_ASSIGN_PTR_LPTR_ERR, EVAL_STMT_ASSIGN_LPTR_LPTR_ERR

$$\frac{\Sigma(\ell_1) = \mathsf{ptr}\,(pv_1) \qquad n' \neq 0 \\ \Sigma(\ell_2) = \mathsf{lptr}\,({x^{n'}}@\pi_2)}{(\Sigma, \ell_1 = \ell_2) \longrightarrow^{n}_{\mathsf{stmt}\,\Delta} (\Sigma, \mathsf{error})}\text{EVAL\_STMT\_ASSIGN\_PTR\_LPTR\_ERR}$$

$$\frac{\Sigma({x_1}^{n_1}@\pi_1) = \mathsf{lptr}\,(pv_1) \qquad n_2 \not\leq n_1 \\ \Sigma(\ell_2) = \mathsf{lptr}\,({x_2}^{n_2}@\pi_2)}{(\Sigma, {x_1}^{n_1}@\pi_1 = \ell_2) \longrightarrow^{n}_{\mathsf{stmt}\,\Delta} (\Sigma, \mathsf{error})}\text{EVAL\_STMT\_ASSIGN\_LPTR\_LPTR\_ERR}$$

Choose $\Psi' = \Psi$.

We must show the following:

- $\vdash^{\Delta;n}_{\mathsf{ty}} \Psi\,\mathsf{ok}$: By assumption.
- $\Psi \vdash_{\mathsf{st}} \Sigma\,\mathsf{ok}$: By assumption.
- $\Psi \vdash^{\Delta;n}_{\mathsf{stmt}} \mathsf{error}\,\mathsf{ok}$: Immediate from TYPE_STMT_ERROR.
- $\Psi \subseteq_n \Psi$: By reflexivity of $\subseteq_n$.

Case EVAL_STMT_ASSIGN_LPTR_LPTR_NULL

$$\frac{\Sigma(\ell_1) = \mathsf{lptr}\,(pv_1) \qquad \Sigma(\ell_2) = \mathsf{lptr}\,(\mathsf{bad\_ptr}) \\ \Sigma' = \Sigma\,[\ell_1 \mapsto \mathsf{lptr}\,(\mathsf{bad\_ptr})]}{(\Sigma, \ell_1 = \ell_2) \longrightarrow^{n}_{\mathsf{stmt}\,\Delta} (\Sigma', \mathsf{skip})}\text{EVAL\_STMT\_ASSIGN\_LPTR\_LPTR\_NULL}$$

Choose $\Psi' = \Psi$.

We must show the following:

- $\vdash^{\Delta;n}_{\mathsf{ty}} \Psi\,\mathsf{ok}$: By assumption.
- $\Psi \vdash_{\mathsf{st}} \Sigma'\,\mathsf{ok}$: As argued above, we need only show that $\Psi; \Sigma' \vdash_{\mathsf{st1}} \ell_1 : \mathsf{lptr}\langle\tau'\rangle\,\mathsf{ok}$. We know by repeated inversion on $\Psi \vdash_{\mathsf{st}} \Sigma\,\mathsf{ok}$ that ℓ_1 has no path. Then, the fact that $\Sigma'(\ell_1) = \mathsf{lptr}\,(\mathsf{bad\_ptr})$ gives us $\Psi; \Sigma' \vdash_{\mathsf{st1}} \ell_1 : \mathsf{lptr}\langle\tau'\rangle\,\mathsf{ok}$ immediately, using CONS_BINDING_LPTR_NULL.
- $\Psi \vdash^{\Delta;n}_{\mathsf{stmt}} \mathsf{skip}\,\mathsf{ok}$: Immediate from TYPE_STMT_SKIP.
- $\Psi \subseteq_n \Psi$: By reflexivity of $\subseteq_n$.

Case EVAL_STMT_ASSIGN_LPTR_LPTR

$$\frac{\Sigma({x_1}^{n_1}@\pi_1) = \mathsf{lptr}\,(pv_1) \\ \Sigma(\ell_2) = \mathsf{lptr}\,({x_2}^{n_2}@\pi_2) \\ \Sigma' = \Sigma\,[{x_1}^{n_1}@\pi_1 \mapsto \mathsf{lptr}\,({x_2}^{n_2}@\pi_2)] \qquad n_2 \leq n_1}{(\Sigma, {x_1}^{n_1}@\pi_1 = \ell_2) \longrightarrow^{n}_{\mathsf{stmt}\,\Delta} (\Sigma', \mathsf{skip})}\text{EVAL\_STMT\_ASSIGN\_LPTR\_LPTR}$$

Choose $\Psi' = \Psi$.

We must show the following:

- $\vdash^{\Delta;n}_{\mathsf{ty}} \Psi\,\mathsf{ok}$: By assumption.
- $\Psi \vdash_{\mathsf{st}} \Sigma'\,\mathsf{ok}$: As argued above, we need only show that $\Psi; \Sigma' \vdash_{\mathsf{st1}} \ell_1 : \mathsf{lptr}\langle\tau'\rangle\,\mathsf{ok}$. We know by repeated inversion on $\Psi \vdash_{\mathsf{st}} \Sigma\,\mathsf{ok}$ that ℓ_1 has no path – in other words π_1 is the empty path. By construction of Σ', $\Sigma'({x_1}^{n_1}@\,) = \mathsf{lptr}\,({x_2}^{n_2}@\pi_2)$. By the argument above, we know that $\Psi({x_2}^{n_2}@\pi_2) = \tau'$. Thus, we can conclude $\Psi; \Sigma' \vdash_{\mathsf{st1}} \ell_1 : \mathsf{lptr}\langle\tau'\rangle\,\mathsf{ok}$ as desired, by CONS_BINDING_LPTR.
- $\Psi \vdash^{\Delta;n}_{\mathsf{stmt}} \mathsf{skip}\,\mathsf{ok}$: Immediate from TYPE_STMT_SKIP.
- $\Psi \subseteq_n \Psi$: By reflexivity of $\subseteq_n$.

Case EVAL_STMT_SEQ_SKIP

$$\frac{}{(\Sigma, \mathsf{skip}; s) \underset{\mathsf{stmt}}{\longrightarrow}^{n}_{\Delta} (\Sigma, s)}\text{EVAL\_STMT\_SEQ\_SKIP}$$

Choose $\Psi' = \Psi$.

We must show the following:

- $\vdash^{\Delta;|s|+n}_{\mathsf{ty}} \Psi\,\mathsf{ok}$: By the definition of the size operation, we can see that $|\mathsf{skip}; s| = |s|$, and so we have this by assumption.
- $\Psi \vdash_{\mathsf{st}} \Sigma\,\mathsf{ok}$: By assumption.
- $\Psi \vdash^{\Delta;n}_{\mathsf{stmt}} s\,\mathsf{ok}$: By inversion on $\Psi \vdash^{\Delta;n}_{\mathsf{stmt}} \mathsf{skip}; s\,\mathsf{ok}$.
- $\Psi \subseteq_n \Psi$: By reflexivity of $\subseteq_n$.

Case EVAL_STMT_ASSIGN_CONG1

$$\frac{(\Sigma, e_1) \underset{\mathsf{exp}}{\longrightarrow}^{n}_{\Delta} (\Sigma', e_1')}{(\Sigma, e_1 = e_2) \underset{\mathsf{stmt}}{\longrightarrow}^{n}_{\Delta} (\Sigma', e_1' = e_2)}\text{EVAL\_STMT\_ASSIGN\_CONG1}$$

To use the induction hypothesis, we must show only that $\vdash^{\Delta;|e_1|+n}_{\mathsf{ty}} \Psi\,\mathsf{ok}$; the other conditions are immediate from the assumptions. We know that $\mathsf{source}\,(e_2)$ by the well-formedness condition SWF_ASSIGN1, along with the fact that e_1 takes a step. By lemma 26, we know that $0 = |e_2|$. Therefore $|e_1 = e_2| = |e_1|$. Thus, we have the desired condition, $\vdash^{\Delta;|e_1|+n}_{\mathsf{ty}} \Psi\,\mathsf{ok}$, by assumption.

The application of the induction hypothesis gives us a Ψ_1 such that $\vdash^{\Delta;|e_1'|+n}_{\mathsf{ty}} \Psi_1\,\mathsf{ok}$, $\Psi_1 \vdash_{\mathsf{st}} \Sigma'\,\mathsf{ok}$, $\Psi_1 \vdash^{\Delta;n}_{\mathsf{exp}} e_1' : \tau_1$, and $\Psi \subseteq_n \Psi_1$.

Choose $\Psi' = \Psi_1$.

We must show the following:

- $\vdash_{\mathsf{ty}}^{\Delta;|e_1'=e_2|+n} \Psi_1\,\mathsf{ok}$: Because e_2 has not changed, we know $|e_1' = e_2| = |e_1'|$. Therefore, this condition is immediate from IH.
- $\Psi_1 \vdash_{\mathsf{st}} \Sigma'\,\mathsf{ok}$: Immediate from IH.
- $\Psi_1 \vdash_{\mathsf{stmt}}^{\Delta;n} e_1' = e_2\,\mathsf{ok}$: The IH gives us $\Psi_1 \vdash_{\mathsf{exp}}^{\Delta;n} e_1' : \tau_1$. We must show that $\Psi_1 \vdash_{\mathsf{exp}}^{\Delta;n} e_2 : \tau_2$ for the same τ_2 as before the step; then inversion on $\Psi \vdash_{\mathsf{stmt}}^{\Delta;n} e_1 = e_2\,\mathsf{ok}$ gives us $\tau_1 \Leftarrow \tau_2$. We use lemma 20, noting that $\mathsf{source}(e_2)$ and $\Psi \subseteq_n \Psi_1$, as required. We can then conclude that $\Psi_1 \vdash_{\mathsf{stmt}}^{\Delta;n} e_1' = e_2\,\mathsf{ok}$.
- $\Psi \subseteq_n \Psi_1$: Immediate from IH.

Case EVAL_STMT_ASSIGN_CONG2

$$\frac{(\Sigma, e_2) \xrightarrow[\mathsf{exp}]{}{}^{n}_{\Delta} (\Sigma', e_2')}{(\Sigma, \ell_1 = e_2) \xrightarrow[\mathsf{stmt}]{}{}^{n}_{\Delta} (\Sigma', \ell_1 = e_2')}\text{EVAL\_STMT\_ASSIGN\_CONG2}$$

To use the induction hypothesis, we must show only that $\vdash_{\mathsf{ty}}^{\Delta;|e_2|+n} \Psi\,\mathsf{ok}$; the other conditions are immediate from the assumptions. We know that $0 = |\ell_1|$. Therefore $|\ell_1 = e_2| = |e_2|$. Thus, we have $\vdash_{\mathsf{ty}}^{\Delta;|e_2|+n} \Psi\,\mathsf{ok}$ by assumption.

The application of the induction hypothesis gives us a Ψ_1 such that $\vdash_{\mathsf{ty}}^{\Delta;|e_2'|+n} \Psi_1\,\mathsf{ok}$, $\Psi_1 \vdash_{\mathsf{st}} \Sigma'\,\mathsf{ok}$, $\Psi_1 \vdash_{\mathsf{exp}}^{\Delta;n} e_2' : \tau_2$, and $\Psi \subseteq_n \Psi_1$.

Choose $\Psi' = \Psi_1$.

We must show the following:

- $\vdash_{\mathsf{ty}}^{\Delta;|\ell_1=e_2'|+n} \Psi_1\,\mathsf{ok}$: We still know $|e_2'| = |\ell_1 = e_2'|$. Therefore, this condition is immediate from IH.
- $\Psi_1 \vdash_{\mathsf{st}} \Sigma'\,\mathsf{ok}$: Immediate from IH.
- $\Psi_1 \vdash_{\mathsf{stmt}}^{\Delta;n} \ell_1 = e_2'\,\mathsf{ok}$: The IH gives us $\Psi_1 \vdash_{\mathsf{exp}}^{\Delta;n} e_2' : \tau_2$. We must show that $\Psi_1 \vdash_{\mathsf{exp}}^{\Delta;n} \ell_1 : \tau_1$ for the same τ_1 as before the step; then inversion on $\Psi \vdash_{\mathsf{stmt}}^{\Delta;n} e_1 = e_2\,\mathsf{ok}$ gives us $\tau_1 \Leftarrow \tau_2$. From the definition of $\subseteq_n$, we can see that $\Psi_1(\ell_1) = \Psi(\ell_1)$. Therefore, $\Psi_1 \vdash_{\mathsf{exp}}^{\Delta;n} \ell_1 : \tau_1$ as desired.
- $\Psi \subseteq_n \Psi_1$: Immediate from IH.

Case EVAL_STMT_ASSIGN_ERR1

$$\frac{}{(\Sigma, \mathsf{error} = e_2) \xrightarrow[\mathsf{stmt}]{}{}^{n}_{\Delta} (\Sigma, \mathsf{error})}\text{EVAL\_STMT\_ASSIGN\_ERR1}$$

Choose $\Psi' = \Psi$.

We must show the following:

- $\vdash^{\Delta;n}_{\mathsf{ty}} \Psi\,\mathsf{ok}$: By the well-formedness condition, $\mathsf{source}\,(e_2)$. Therefore, by lemma 26, $|\mathsf{error} = e_2| = 0$, meaning we have this condition by assumption.
- $\Psi \vdash_{\mathsf{st}} \Sigma\,\mathsf{ok}$: By assumption.
- $\Psi \vdash^{\Delta;n}_{\mathsf{stmt}} \mathsf{error}\,\mathsf{ok}$: Immediate from TYPE_STMT_ERROR.
- $\Psi \subseteq_n \Psi$: By reflexivity of $\subseteq_n$.

Case EVAL_STMT_ASSIGN_ERR2

$$\frac{}{(\Sigma, \ell_1 = \mathsf{error}) \xrightarrow[\mathsf{stmt}]{}{}^{n}_{\Delta} (\Sigma, \mathsf{error})}\text{EVAL\_STMT\_ASSIGN\_ERR2}$$

Choose $\Psi' = \Psi$.

We must show the following:

- $\vdash^{\Delta;n}_{\mathsf{ty}} \Psi\,\mathsf{ok}$: We can see that $|\ell_1 = \mathsf{error}| = 0$, meaning that we have this condition by assumption.
- $\Psi \vdash_{\mathsf{st}} \Sigma\,\mathsf{ok}$: By assumption.
- $\Psi \vdash^{\Delta;n}_{\mathsf{stmt}} \mathsf{error}\,\mathsf{ok}$: Immediate from TYPE_STMT_ERROR.
- $\Psi \subseteq_n \Psi$: By reflexivity of $\subseteq_n$.

Case EVAL_STMT_SEQ_CONG

$$\frac{(\Sigma, s_1) \xrightarrow[\mathsf{stmt}]{}{}^{n}_{\Delta} (\Sigma', s_1')}{(\Sigma, s_1; s_2) \xrightarrow[\mathsf{stmt}]{}{}^{n}_{\Delta} (\Sigma', s_1'; s_2)}\text{EVAL\_STMT\_SEQ\_CONG}$$

To use the induction hypothesis, we must show only that $\vdash^{\Delta;|s_1|+n}_{\mathsf{ty}} \Psi\,\mathsf{ok}$; the other conditions are immediate from the assumptions. We know that $\mathsf{source}\,(s_2)$ by the well-formedness condition SWF_SEQ. By lemma 26, we know that $0 = |s_2|$. Therefore $|s_1; s_2| = |s_1|$. Thus, we have $\vdash^{\Delta;|s_1|+n}_{\mathsf{ty}} \Psi\,\mathsf{ok}$ by assumption.

The application of the induction hypothesis gives us a Ψ_1 such that $\vdash^{\Delta;|s_1'|+n}_{\mathsf{ty}} \Psi_1\,\mathsf{ok}$, $\Psi_1 \vdash_{\mathsf{st}} \Sigma'\,\mathsf{ok}$, $\Psi_1 \vdash^{\Delta;n}_{\mathsf{stmt}} s_1'\,\mathsf{ok}$, and $\Psi \subseteq_n \Psi_1$.

Choose $\Psi' = \Psi_1$.

We must show the following:

- $\vdash^{\Delta;|s_1';s_2|+n}_{\mathsf{ty}} \Psi_1\,\mathsf{ok}$: By the well-formedness preservation theorem, we know $\mathsf{source}\,(s_2)$ and therefore, by the same reasoning as above, $|s_1'| = |s_1'; s_2|$. Thus, we have this condition directly from the IH.
- $\Psi_1 \vdash_{\mathsf{st}} \Sigma'\,\mathsf{ok}$: Immediate from IH.

- $\Psi_1 \vdash^{\Delta;n}_{\mathsf{stmt}} s_1'; s_2\,\mathsf{ok}$: The IH gives us $\Psi_1 \vdash^{\Delta;n}_{\mathsf{stmt}} s_1'\,\mathsf{ok}$. We must show that $\Psi_1 \vdash^{\Delta;n}_{\mathsf{stmt}} s_2\,\mathsf{ok}$. We use lemma 20, noting that $\mathsf{source}\,(s_2)$ and $\Psi \subseteq_n \Psi_1$, as required. We can then conclude that $\Psi_1 \vdash^{\Delta;n}_{\mathsf{stmt}} s_1'; s_2\,\mathsf{ok}$.
- $\Psi \subseteq_n \Psi_1$: Immediate from IH.

Case EVAL_STMT_SEQ_ERROR

$$\frac{}{(\Sigma, \mathsf{error}; s_2) \xrightarrow[\mathsf{stmt}]{}{}^{n}_{\Delta} (\Sigma, \mathsf{error})}\text{EVAL\_STMT\_SEQ\_ERROR}$$

Choose $\Psi' = \Psi$.

We must show the following:

- $\vdash^{\Delta;n}_{\mathsf{ty}} \Psi\,\mathsf{ok}$: By the well-formedness condition, $\mathsf{source}\,(s_2)$. Therefore, as reasoned above, $|\mathsf{error}; s_2| = 0$, meaning that we have this condition by assumption.
- $\Psi \vdash_{\mathsf{st}} \Sigma\,\mathsf{ok}$: By assumption.
- $\Psi \vdash^{\Delta;n}_{\mathsf{stmt}} \mathsf{error}\,\mathsf{ok}$: Immediate from TYPE_STMT_ERROR.
- $\Psi \subseteq_n \Psi$: By reflexivity of $\subseteq_n$.

Case EVAL_EXP_VAR

$$\frac{}{(\Sigma, x) \xrightarrow[\mathsf{exp}]{}{}^{n}_{\Delta} (\Sigma, x^n@\,)}\text{EVAL\_EXP\_VAR}$$

Choose $\Psi' = \Psi$.

We must show the following:

- $\vdash^{\Delta;n}_{\mathsf{ty}} \Psi\,\mathsf{ok}$: We see that $|x| = 0$, so we have this condition by assumption.
- $\Psi \vdash_{\mathsf{st}} \Sigma\,\mathsf{ok}$: By assumption.
- $\Psi \vdash^{\Delta;n}_{\mathsf{exp}} x^n@ : \tau$: From inversion on $\Psi \vdash^{\Delta;n}_{\mathsf{exp}} x : \tau$, we get that $\Psi(x^n@\,) = \tau$. In order to use TYPE_EXP_LOC to satisfy the desired condition, we must only show that $n \leq n$, which is trivially true.
- $\Psi \subseteq_n \Psi$: By reflexivity of $\subseteq_n$.

Case EVAL_EXP_NULL

$$\frac{\begin{array}{l} x^n \textbf{ fresh for } \Sigma \\ \Sigma' = \Sigma\,[x^n@ \mapsto \mathsf{ptr}\,(\mathsf{bad\_ptr})] \end{array}}{(\Sigma, \mathsf{null}) \xrightarrow[\mathsf{exp}]{}{}^{n}_{\Delta} (\Sigma', x^n@\,)}\text{EVAL\_EXP\_NULL}$$

We use inversion on $\Psi \vdash^{\Delta;n}_{\mathsf{exp}} \mathsf{null} : \tau$ to get $\tau = \mathsf{ptr}\langle\tau'\rangle$. Choose $\Psi' = \Psi\,[x^n@ : \mathsf{ptr}\langle\tau'\rangle]$.

We must show the following:

- $\vdash^{\Delta;n}_{\mathsf{ty}} \Psi' \,\mathsf{ok}$: We see that $|\mathsf{null}| = 0$, so we have this condition by assumption..
- $\Psi' \vdash_{\overline{\mathsf{st}}} \Sigma' \,\mathsf{ok}$: We wish to use lemma 9. We already know that $\Psi \vdash_{\overline{\mathsf{st}}} \Sigma \,\mathsf{ok}$, and the freshness condition on x^n gives us that the domains of the old environments and new binding are distinct. We need only show that $[x^n@ : \mathsf{ptr}\langle\tau'\rangle] \vdash_{\overline{\mathsf{st}}} [x^n@ \mapsto \mathsf{ptr}\,(\mathsf{bad\_ptr})]\,\mathsf{ok}$ to show that $\Psi' \vdash_{\overline{\mathsf{st}}} \Sigma' \,\mathsf{ok}$. This boils down to $[x^n@ : \mathsf{ptr}\langle\tau'\rangle]; [x^n@ \mapsto \mathsf{ptr}\,(\mathsf{bad\_ptr})] \vdash_{\overline{\mathsf{st}}1} x^n@ : \mathsf{ptr}\langle\tau'\rangle\,\mathsf{ok}$, which is derivable directly from CONS_BINDING_PTR_NULL. We have satisfied this condition.
- $\Psi' \vdash^{\Delta;n}_{\mathsf{exp}} x^n@ : \tau$: We use TYPE_EXP_LOC, as its premises are immediate.
- $\Psi \subseteq_n \Psi'$: By construction.

Case EVAL_EXP_FLD

$$\frac{}{(\Sigma, (x^{n'}@\pi).x') \xrightarrow[\mathsf{exp}]{}{}^{n}_{\Delta} (\Sigma, x^{n'}@\pi +\!\!+ x')}\text{EVAL\_EXP\_FLD}$$

Choose $\Psi' = \Psi$.

We must show the following:

- $\vdash^{\Delta;n}_{\mathsf{ty}} \Psi\,\mathsf{ok}$: We see that $|(x^{n'}@\pi).x'| = 0$, so we have this condition by assumption.
- $\Psi \vdash_{\overline{\mathsf{st}}} \Sigma\,\mathsf{ok}$: By assumption.
- $\Psi \vdash^{\Delta;n}_{\mathsf{exp}} x^{n'}@\pi +\!\!+ x' : \tau$: In order to use TYPE_EXP_LOC, we must show that $\Psi(x^{n'}@\pi +\!\!+ x') = \tau$ and than $n' \leq n$. Repeated inversion on $\Psi \vdash^{\Delta;n}_{\mathsf{exp}} x^{n'}@\pi.x' : \tau$ gives us the following facts:
  1. $\Psi \vdash^{\Delta;n}_{\mathsf{exp}} x^{n'}@\pi : C$
     a) $\Psi(x^{n'}@\pi) = C$
     b) $n' \leq n$
  2. $\mathsf{struct}\ C\ \{\ fldsmeths\}; \in \Delta$
  3. $\tau\ x'; \in flds.$

  Now, we invert $\vdash^{\Delta;n}_{\mathsf{ty}} \Psi\,\mathsf{ok}$ to get $\Psi \vdash^{\Delta;n}_{\mathsf{ty1}} x^{n'}@\pi : C\,\mathsf{ok}$, which was proved either by case
  CONS_TYPE_BINDING_FIELD_CLASS or CONS_TYPE_BINDING_LOCAL_CLASS. In both cases, we can see that $\Psi(x^{n'}@\pi +\!\!+ x') = \tau$. Thus, we can use TYPE_EXP_LOC to show that $\Psi \vdash^{\Delta;n}_{\mathsf{exp}} x^{n'}@\pi +\!\!+ x' : \tau$.
- $\Psi \subseteq_n \Psi$: By reflexivity of $\subseteq_n$.

Case EVAL_EXP_METH

$$\frac{\begin{array}{l}\Sigma(\ell_1) = C \\ \mathsf{struct}\ C\ \{\ fldsmeths\}; \in \Delta \\ \tau_1\ f(\tau_2\ x)\{\overline{\tau_i\ x_i}^{\,i}; s; \mathsf{return}\ e\} \in meths \\ \Sigma_2 = \mathsf{copy\_store}(\Sigma, {x_2}^{n_2}@\pi_2, x^{n+1}) \\ \Sigma_3 = [{this}^{n+1}@ \mapsto \mathsf{lptr}\,(\ell_1)] \\ \overline{\Sigma_{4\,i} = \mathsf{default}_\Delta\langle\tau_i\rangle({x_i}^{n+1}@)}^{\,i} \\ \Sigma' = \Sigma\,\Sigma_2\,\Sigma_3\,\overline{\Sigma_{4\,i}}^{\,i}\end{array}}{(\Sigma, \ell_1.f(x_2{}^{n_2}@\pi_2)) \xrightarrow[\mathsf{exp}]{}{}^{n}_{\Delta} (\Sigma', \{s;\ \mathsf{return}\ e\})}\text{EVAL\_EXP\_METH}$$

Let the following definitions hold:
$\Psi_2 = \mathsf{copy\_types}(\Psi, \ell_2, {x^{n'}})$
$\Psi_3 = [{this}^{n'}@\ : \mathsf{lptr}\langle C\rangle]$
$\overline{\Psi_{4\,i} = \mathsf{build\_types}_\Delta\langle\tau_i\rangle({x_i}^{n'}@)}^{\,i}$
Note that Ψ_2, Ψ_3, and the $\Psi_{4\,i}$ are uniquely determined, invoking lemma 23. Then, choose $\Psi' = \Psi\,\Psi_2\,\Psi_3\,\overline{\Psi_{4\,i}}^{\,i}$.

First, we show a subsidiary fact: the domains of Ψ, Ψ_2, Ψ_3, and the $\Psi_{4\,i}$ are all distinct. That the domains of the last 3 are mutually distinct follows directly from repeated inversion on $\vdash_{\overline{\mathsf{env}}}\ \Delta\,\mathsf{ok}$, noting the uniqueness condition in rule TYPE_METH_DECL. That the domain of Ψ is distinct from that of the others follows from $\vdash^{\Delta;n}_{\overline{\mathsf{ty}}}\ \Psi\,\mathsf{ok}$ (noting that $|\ell_1.f(\ell_2)| = 0$), lemma 12, and that $n' = n + 1$.

We must show the following:

- $\vdash^{\Delta;|\{s;\,\mathsf{return}\,e\}|+n}_{\overline{\mathsf{ty}}}\ \Psi'\,\mathsf{ok}$: We let $k' = |\{s;\ \mathsf{return}\ e\}| + n$. We will show that $\vdash^{\Delta;k'}_{\overline{\mathsf{ty}}}\ \Psi\,\mathsf{ok}$, $\vdash^{\Delta;k'}_{\overline{\mathsf{ty}}}\ \Psi_2\,\mathsf{ok}$, $\vdash^{\Delta;k'}_{\overline{\mathsf{ty}}}\ \Psi_3\,\mathsf{ok}$, and $\vdash^{\Delta;k'}_{\overline{\mathsf{ty}}}\ \Psi_{4\,i}\,\mathsf{ok}$ for any i in range. By lemma 11, we will have shown $\vdash^{\Delta;k'}_{\overline{\mathsf{ty}}}\ \Psi'\,\mathsf{ok}$.
  - $\vdash^{\Delta;k'}_{\overline{\mathsf{ty}}}\ \Psi\,\mathsf{ok}$: We use $\vdash^{\Delta;n}_{\overline{\mathsf{ty}}}\ \Psi\,\mathsf{ok}$ and lemma 21.
  - $\vdash^{\Delta;k'}_{\overline{\mathsf{ty}}}\ \Psi_2\,\mathsf{ok}$: We use lemmas 27 and 25, noting that $k' \geq n + 1$ from the definition of the size operation. We get $\vdash^{\Delta}_{\mathsf{twf}}\ \tau_2$ from the premises of TYPE_METH_DECL.
  - $\vdash^{\Delta;k'}_{\overline{\mathsf{ty}}}\ \Psi_3\,\mathsf{ok}$: Immediate from CONS_TYPE_BINDING_LPTR.
  - $\vdash^{\Delta;k'}_{\overline{\mathsf{ty}}}\ \Psi_{4\,i}\,\mathsf{ok}$ for any i in range: Immediate from lemma 25.
- $\Psi' \vdash_{\overline{\mathsf{st}}}\ \Sigma'\,\mathsf{ok}$: To show $\Psi' \vdash_{\overline{\mathsf{st}}}\ \Sigma'\,\mathsf{ok}$, we break this down into showing that $\Psi\,\Psi_2 \vdash_{\overline{\mathsf{st}}}\ \Sigma\,\Sigma_2\,\mathsf{ok}$, $\Psi_3 \vdash_{\overline{\mathsf{st}}}\ \Sigma_3\,\mathsf{ok}$, and $\overline{\Psi_{4\,i} \vdash_{\overline{\mathsf{st}}}\ \Sigma_{4\,i}\,\mathsf{ok}}^{\,i}$. Then, by repeated application of lemma 9 (using the distinctness of names discussed above), we will be able to conclude $\Psi' \vdash_{\overline{\mathsf{st}}}\ \Sigma'\,\mathsf{ok}$ as desired.
  - $\Psi\,\Psi_2 \vdash_{\overline{\mathsf{st}}}\ \Sigma\,\Sigma_2\,\mathsf{ok}$: We invoke lemma 16. The only condition of that lemma that is not immediate is the last: we must show that if $\Sigma(\ell_2) =$

$\mathsf{lptr}\,({x_3}^{n_3}@\pi_3)$, then $n_3 \leq n'$. We know (from EVAL_EXP_METH) that $\ell_2 = {x_2}^{n_2}@\pi_2$. By lemma 6, $n_3 \leq n_2$. By inversion on $\Psi \vdash^{\Delta;n}_{\mathsf{exp}} \ell_1.f(\ell_2) : \tau$, we get $\Psi \vdash^{\Delta;n}_{\mathsf{exp}} \ell_2 : \tau_2$; by further inversion, we get that $n_2 \leq n$. By the fact that $n' = n + 1$ (in the premises of EVAL_EXP_METH), we can conclude that $n_3 \leq n'$ as desired.

  - $\Psi_3 \vdash_{\mathsf{st}} \Sigma_3\,\mathsf{ok}$: First, we know that $dom(\Sigma_3) = {this^{n'}}@ = dom(\Psi_3)$. Now, we need to show that $[{this^{n'}}@ : \mathsf{lptr}\langle C\rangle]; [{this^{n'}}@ \mapsto \mathsf{lptr}\,(\ell_1)] \vdash_{\mathsf{st1}} {this^{n'}}@ : \mathsf{lptr}\langle C\rangle\,\mathsf{ok}$. From EVAL_EXP_METH, we know that $\Sigma(\ell_1) = C$. We know $\ell_1 = {x_1}^{n_1}@\pi_1$ for some x_1, n_1, and π_1. To use CONS_BINDING_LPTR, we must show $n_1 \leq n'$. By inversion on $\Psi \vdash^{\Delta;n}_{\mathsf{exp}} \ell_1.f(\ell_2) : \tau$, we get $\Psi \vdash^{\Delta;n}_{\mathsf{exp}} \ell_1 : \tau'$; by further inversion, we get $n_1 \leq n$. Because $n' = n + 1$, we have $n_1 \leq n'$ as desired.
  - $\Psi_{4\,i} \vdash_{\mathsf{st}} \Sigma_{4\,i}\,\mathsf{ok}$ for any i: We simply invoke lemma 13.

- $\Psi' \vdash^{\Delta;n}_{\mathsf{exp}} \{s; \mathsf{return}\ e\} : \tau$: Using TYPE_EXP_BODY, we must show $\Psi' \vdash^{\Delta;n'}_{\mathsf{stmt}} s\,\mathsf{ok}$, $\Psi' \vdash^{\Delta;n'}_{\mathsf{exp}} e : \tau$, and that $\tau \neq \mathsf{lptr}\langle\tau_3\rangle$ for some τ_3. By repeated inversion on $\vdash_{\mathsf{env}} \Delta\,\mathsf{ok}$, we can get that there exists a Ψ'' such that the domain of Ψ'' mentions only index 0, that $\mathsf{source}\,(s)$, $\mathsf{source}\,(e)$, $\Psi'' \vdash^{\Delta;0}_{\mathsf{stmt}} s\,\mathsf{ok}$, $\Psi'' \vdash^{\Delta;0}_{\mathsf{exp}} e : \tau$, and that $\tau \neq \mathsf{lptr}\langle\tau_3\rangle$. Then, we can use lemma 17 to say that $\Psi''' \vdash^{\Delta;n'}_{\mathsf{stmt}} s\,\mathsf{ok}$ and $\Psi''' \vdash^{\Delta;n'}_{\mathsf{exp}} e : \tau$, where Ψ''' is just like Ψ'' but with all indices changed from 0 to n'. We wish to show that this $\Psi''' = \Psi_2\,\Psi_3\,\overline{\Psi_{4\,i}}^{\,i}$. It is easy to see that Ψ''' has parts equal to Ψ_3 and $\overline{\Psi_{4\,i}}^{\,i}$. To show that the last part of Ψ''' equals Ψ_2, we use lemma 27. Thus, $\Psi''' \subseteq \Psi'$. By lemma 20 (and using lemma 18), we now have $\Psi' \vdash^{\Delta;n'}_{\mathsf{stmt}} s\,\mathsf{ok}$, $\Psi' \vdash^{\Delta;n'}_{\mathsf{exp}} e : \tau$, and $\tau \neq \mathsf{lptr}\langle\tau_3\rangle$ as required.
- $\Psi \subseteq_n \Psi'$: Immediate from the definition of Ψ' in terms of Ψ.

Case EVAL_EXP_BODY_RET

$$\frac{\begin{array}{l} x^n\ \textbf{fresh for}\ \Sigma \\ \Sigma_2 = \mathsf{copy\_store}(\Sigma, \ell, x^n) \\ \Sigma' = \Sigma\,\Sigma_2 \backslash (n+1) \end{array}}{(\Sigma, \{\mathsf{skip}; \mathsf{return}\,\ell\}) \xrightarrow[\mathsf{exp}]{n}_{\Delta} (\Sigma', x^n@\,)}\text{EVAL\_EXP\_BODY\_RET}$$

Let $\Psi_2 = \mathsf{copy\_types}(\Psi, \ell, x^n)$. Then, we choose $\Psi' = (\Psi\,\Psi_2)\backslash n + 1$.

We must show the following:

- $\vdash^{\Delta;n}_{\mathsf{ty}} \Psi'\,\mathsf{ok}$: It is straightforward to calculate $|\{\mathsf{skip}; \mathsf{return}\,\ell\}| = |x^n@\,| + 1$. From $\vdash^{\Delta;n+1}_{\mathsf{ty}} \Psi\,\mathsf{ok}$ and lemma 12, we know that every domain element of Ψ either has an index less than or equal to n or an index equal to $n+1$. By easy induction on $\mathsf{copy\_types}$, we can see that every domain element of Ψ_2 has an index of $n + 1$. Thus, every domain element of $\Psi\,\Psi_2$ has an index less than or equal to n or an index equal to $n + 1$. Then, by lemma 28, every domain

element of Ψ' has an index less than or equal to n. Then, by lemma 22, $\vdash^{\Delta;n}_{\mathsf{ty}} \Psi'\,\mathsf{ok}$ as desired.

- $(\Psi\,\Psi_2)\backslash n+1 \vdash_{\mathsf{st}} (\Sigma\,\Sigma_2)\backslash n+1\,\mathsf{ok}$: This fact can be shown by lemma 29. To do so, we must show the following:
  - $\Psi\,\Psi_2 \vdash_{\mathsf{st}} \Sigma\,\Sigma_2\,\mathsf{ok}$: We use lemma 16. The only premise of this lemma that is not immediate is the one concerning lptrs. We must show that if $\Sigma(\ell) = \mathsf{lptr}\,({x_3}^{n_3}@\pi_3)$, then $n_3 \leq n$. However, we can use the fact $\tau \neq \mathsf{lptr}\langle\tau'\rangle$ (from TYPE_EXP_BODY). The premises of TYPE_EXP_BODY also give us that $\Psi \vdash^{\Delta;n+1}_{\mathsf{exp}} \ell : \tau$. The only rule that could give rise to this judgment is TYPE_EXP_LOC, which, in turn, says that $\Psi(\ell) = \tau$. Since $\tau \neq \mathsf{lptr}\langle\tau'\rangle$, then τ must be either $\mathsf{ptr}\langle\tau'\rangle$ for some τ' or C for some class C. Because $\Psi \vdash_{\mathsf{st}} \Sigma\,\mathsf{ok}$, we can see that $\Sigma(\ell) = \mathsf{ptr}\,(\ell')$ (for some ℓ') or $\Sigma(\ell) = C$, depending on the nature of τ. Thus, we can see that $\Sigma(\ell)$ can never have the form $\mathsf{lptr}\,({x_3}^{n_3}@\pi_3)$, and we can use lemma 16 as desired, giving us $(\Psi\,\Psi_2) \vdash_{\mathsf{st}} (\Sigma\,\Sigma_2)\,\mathsf{ok}$ as desired.
  - No domain element in $\Psi\,\Psi_2$ has an index greater than $n+1$: By lemma 12, no domain element in Ψ has an index equal to or greater than $n+1$. By easy induction on $\mathsf{copy\_types}$, we can see that every index in a location in Ψ_2 has an index exactly equal to $n+1$.
- $\Psi' \vdash^{\Delta;n}_{\mathsf{exp}} x^n@ : \tau$: To show this, we will have to show $\Psi'(x^n@) = \tau$. (The other condition in TYPE_EXP_LOC is $n \leq n$, which is trivially satisfied.) We know $\Psi(\ell) = \tau$ and $\Psi_2 = \mathsf{copy\_types}(\Psi, \ell, x^n)$. We invoke lemma 14 to say that $\Psi_2(x^n@) = \tau$. By lemma 31, we can say $(\Psi_2\backslash n+1)(x^n@) = \tau$, noting that $n \neq n+1$. Finally, because we know that, by construction and lemma 30, $\Psi_2\backslash n+1 \subseteq \Psi'$, $\Psi'(x^n@) = \tau$ as desired.
- $\Psi \subseteq_n \Psi'$: By construction and lemma 30, we know that $\Psi\backslash n+1 \subseteq \Psi'$. By lemma 18, we then know that $\Psi\backslash n+1 \subseteq_n \Psi'$. Therefore, invoking lemma 19 tells us that it is sufficient to show that $\Psi \subseteq_n \Psi\backslash n+1$. We invoke lemma 32 and we are done.

Case EVAL_EXP_NEW

$$\frac{\begin{array}{l} x^n, y^0 \textbf{ fresh for } \Sigma \\ \Sigma_1 = \mathsf{default}_\Delta\langle C\rangle(y^0@) \qquad\qquad \Sigma' = \Sigma\,\Sigma_1\,[x^n@ \mapsto \mathsf{ptr}\,(y^0@)] \end{array}}{(\Sigma, \mathsf{new}\ C()) \xrightarrow[\mathsf{exp}]{}{}^n_\Delta (\Sigma', x^n@)}\text{EVAL\_EXP\_NEW}$$

Let $\Psi_1 = \mathsf{build\_types}_\Delta\langle C\rangle(y^0@)$. Choose $\Psi' = \Psi\,\Psi_1\,[x^n@ : \mathsf{ptr}\langle C\rangle]$.

We must show the following:

- $\vdash^{\Delta;n}_{\mathsf{ty}} \Psi'\,\mathsf{ok}$: We break this down into three pieces, proved below. We then combine the pieces using lemma 11, noting that the domain distinctness condition holds from the freshness condition on x^n and y^0.

- $\vdash^{\Delta;n}_{\mathsf{ty}} \Psi\,\mathsf{ok}$: We see that $|\mathsf{new}\,C()| = 0$, so we have this condition by assumption.
- $\vdash^{\Delta;n}_{\mathsf{ty}} \Psi_1\,\mathsf{ok}$: We wish to use lemma 25. We must show the following:
  - $\vdash^{\Delta}_{\mathsf{twf}} C$: Immediate from inversion on $\Psi \vdash^{\Delta;n}_{\mathsf{exp}} \mathsf{new}\,C() : \mathsf{ptr}\langle C\rangle$.
  - $0 \leq n$: By the fact that n is a natural number.
- $\vdash^{\Delta;n}_{\mathsf{ty}} [x^n@ : \mathsf{ptr}\langle C\rangle]\,\mathsf{ok}$: From the rule CONS_TYPE_BINDINGS, we must show that $[x^n@ : \mathsf{ptr}\langle C\rangle] \vdash^{\Delta;n}_{\mathsf{ty1}} x^n@ : \mathsf{ptr}\langle C\rangle\,\mathsf{ok}$. We use rule CONS_TYPE_BINDING_LOCAL_PTR.

- $\Psi' \vdash_{\mathsf{st}} \Sigma'\,\mathsf{ok}$: We similarly break this into pieces:
  - $\Psi \vdash_{\mathsf{st}} \Sigma\,\mathsf{ok}$: By assumption.
  - $\Psi_1\,[x^n@ : \mathsf{ptr}\langle C\rangle] \vdash_{\mathsf{st}} \Sigma_1\,[x^n@ \mapsto \mathsf{ptr}\,(y^0@\,)]\,\mathsf{ok}$: We invoke lemma 13 to get $\Psi_1 \vdash_{\mathsf{st}} \Sigma_1\,\mathsf{ok}$. By inversion on this fact, we get $\Psi_1; \Sigma_1 \vdash_{\mathsf{st1}} \ell_i : \tau_i\,\mathsf{ok}$ for every binding $[\ell_i : \tau_i]$ in Ψ_1. Noting that the freshness condition gives us that $x^n@ \neq y^0@$ and that all domain elements in Ψ_1 and Σ_1 start with $y^0@$ (ensuring distinct domains), we use lemma 7 to say that $\Psi_1\,[x^n@ : \mathsf{ptr}\langle C\rangle]; \Sigma_1\,[x^n@ \mapsto \mathsf{ptr}\,(y^0@\,)] \vdash_{\mathsf{st1}} \ell_i : \tau_i\,\mathsf{ok}$ holds for every binding $[\ell_i : \tau_i]$ in Ψ_1. Now, to reach our goal, it is necessary to show the following:
    - $\Psi_1\,[x^n@ : \mathsf{ptr}\langle C\rangle]; \Sigma_1\,[x^n@ \mapsto \mathsf{ptr}\,(y^0@\,)] \vdash_{\mathsf{st1}} x^n@ : \mathsf{ptr}\langle C\rangle\,\mathsf{ok}$: We use CONS_BINDING_PTR. We need only show that $\Psi_1(y^0@\,) = C$. By inversion on $\Psi_1 = \mathsf{build\_types}_\Delta\langle C\rangle(y^0@\,)$, we see that $\Psi_1 = [y^0@ : C]\,\Psi_1'$ for some Ψ_1'. Because all elements in Ψ_1' have non-empty paths (as can be easily seen in the premises of AUX_BUILD_TYPES_CLASS), none of the bindings in Ψ_1' override that first binding in Ψ_1. Therefore $\Psi_1(y^0@\,) = C$ as desired.
    - $dom\,(\Sigma_1\,[x^n@ \mapsto \mathsf{ptr}\,(y^0@\,)]) \subseteq dom\,(\Psi_1\,[x^n@ : \mathsf{ptr}\langle C\rangle])$: From inversion on $\Psi_1 \vdash_{\mathsf{st}} \Sigma_1\,\mathsf{ok}$ (established earlier), we get $dom\,(\Sigma_1) \subseteq dom\,(\Psi_1)$. The new bindings in each have the same domain element, so we are done.

  We combine the pieces shown above by lemma 9 to show $\Psi' \vdash_{\mathsf{st}} \Sigma'\,\mathsf{ok}$.
- $\Psi' \vdash^{\Delta;n}_{\mathsf{exp}} x^n@ : \mathsf{ptr}\langle C\rangle$: We use rule TYPE_EXP_LOC. By construction, $\Psi'(x^n@\,) = \mathsf{ptr}\langle C\rangle$, and it is trivial to see that $n \leq n$.
- $\Psi \subseteq_n \Psi'$: By construction.

Case EVAL_EXP_ADDR

$$\frac{\begin{array}{l} y^n \textbf{ fresh for } \Sigma \\ \Sigma' = \Sigma\,[y^n@ \mapsto \mathsf{lptr}\,(x^{n'}@\pi)] \end{array}}{(\Sigma, \&x^{n'}@\pi) \underset{\mathsf{exp}}{\longrightarrow}^n_\Delta (\Sigma', y^n@\,)}\text{EVAL\_EXP\_ADDR}$$

Choose $\Psi' = \Psi\,[y^n@\ :\ \mathsf{lptr}\langle\tau\rangle]$, where τ is the τ on inversion of $\Psi \vdash^{\Delta;n}_{\mathsf{exp}} \&x^{n'}@\pi : \mathsf{lptr}\langle\tau\rangle$.

We must show the following:

- $\vdash^{\Delta;n}_{\mathsf{ty}} \Psi'\,\mathsf{ok}$: We note that $|\&x^{n'}@\pi| = |y^n@\,| = 0$. Then, by inversion on $\vdash^{\Delta;n}_{\mathsf{ty}} \Psi\,\mathsf{ok}$, we get $\Psi \vdash^{\Delta;n}_{\mathsf{ty1}} \ell_i : \tau_i\,\mathsf{ok}$ for every binding $[\ell_i : \tau_i]$ in Ψ. We repeatedly apply lemma 34 (noting that y^n is fresh) to get $\Psi' \vdash^{\Delta;n}_{\mathsf{ty1}} \ell_i : \tau_i\,\mathsf{ok}$ for every binding $[\ell_i : \tau_i]$ in Ψ. Now, we need only show $\Psi' \vdash^{\Delta;n}_{\mathsf{ty1}} y^n@\ :\ \mathsf{lptr}\langle\tau\rangle\,\mathsf{ok}$. Then, we use CONS_TYPE_BINDING_LPTR and we are done.
- $\Psi' \vdash_{\mathsf{st}} \Sigma'\,\mathsf{ok}$: By inversion on $\Psi \vdash_{\mathsf{st}} \Sigma\,\mathsf{ok}$, we get $\Psi;\Sigma \vdash_{\mathsf{st1}} \ell_i : \tau_i\,\mathsf{ok}$ for every binding $[\ell_i : \tau_i]$ in Ψ. We repeatedly use lemma 7 to get $\Psi';\Sigma' \vdash_{\mathsf{st1}} \ell_i : \tau_i\,\mathsf{ok}$ for every binding $[\ell_i : \tau_i]$ in Ψ. Now, we must show the following:
  - $\Psi';\Sigma' \vdash_{\mathsf{st1}} y^n@\ :\ \mathsf{lptr}\langle\tau\rangle\,\mathsf{ok}$: We use CONS_BINDING_LPTR. We must show the following:
    - $\Psi'(x^{n'}@\pi) = \tau$: We get this from two inversions on $\Psi \vdash^{\Delta;n}_{\mathsf{exp}} \&x^{n'}@\pi : \mathsf{lptr}\langle\tau\rangle$ (TYPE_EXP_ADDR and then TYPE_EXP_LOC).
    - $n' \leq n$: We also get this from those same two inversions.
  - $dom\,(\Sigma') \subseteq dom\,(\Psi')$: We know from inverting $\Psi \vdash_{\mathsf{st}} \Sigma\,\mathsf{ok}$ that $dom\,(\Sigma) \subseteq dom\,(\Psi)$. The new bindings have the same domain elements, so we are done.
- $\Psi' \vdash^{\Delta;n}_{\mathsf{exp}} y^n@\ :\ \mathsf{lptr}\langle\tau\rangle$: Noting that $n \leq n$ trivially, we use TYPE_EXP_LOC with $\Psi'(y^n@\,) = \mathsf{lptr}\langle\tau\rangle$ by construction.
- $\Psi \subseteq_n \Psi'$: By construction.

Case EVAL_EXP_DEREF_PTR

$$\frac{\Sigma(\ell) = \mathsf{ptr}\,(x^{n'}@\pi)}{(\Sigma, *\ell) \xrightarrow[\mathsf{exp}]{}{}^{n}_{\Delta} (\Sigma, x^{n'}@\pi)}\text{EVAL\_EXP\_DEREF\_PTR}$$

$$\frac{\Psi \vdash^{\Delta;n}_{\mathsf{exp}} e : \mathsf{ptr}\langle\tau\rangle}{\Psi \vdash^{\Delta;n}_{\mathsf{exp}} *e : \tau}\text{TYPE\_EXP\_DEREF\_PTR}$$

Choose $\Psi' = \Psi$.

We must show the following:

- $\vdash^{\Delta;n}_{\mathsf{ty}} \Psi\,\mathsf{ok}$: The size of the starting expression and the resulting expression are the same, so we have this condition by assumption.
- $\Psi \vdash_{\mathsf{st}} \Sigma\,\mathsf{ok}$: By assumption.
- $\Psi \vdash^{\Delta;n}_{\mathsf{exp}} x^{n'}@\pi : \tau$: Two inversions on $\Psi \vdash^{\Delta;n}_{\mathsf{exp}} *\ell : \tau$ (TYPE_EXP_DEREF_PTR and TYPE_EXP_LOC) give us $\Psi(\ell) = \mathsf{ptr}\langle\tau\rangle$. Inversion on $\Psi \vdash_{\mathsf{st}} \Sigma\,\mathsf{ok}$ must therefore give us $\Psi;\Sigma \vdash_{\mathsf{st1}} \ell\ :\ \mathsf{ptr}\langle\tau\rangle\,\mathsf{ok}$. Because we know that $\Sigma(\ell) = \mathsf{ptr}\,(x^{n'}@\pi)$,

$\Psi; \Sigma \vdash_{\mathsf{st1}} \ell : \mathsf{ptr}\langle\tau\rangle\,\mathsf{ok}$ must be derived by using CONS_BINDING_PTR. The premises of this rule state $\Psi(x^{n'}@\pi) = \tau$ and $n' = 0$. Therefore, we have the two premises of TYPE_EXP_LOC, and we can derive $\Psi \vdash^{\Delta;n}_{\mathsf{exp}} x^{n'}@\pi : \tau$ as desired.

- $\Psi \subseteq_n \Psi$: By reflexivity of $\subseteq_n$.

Case EVAL_EXP_DEREF_LPTR

$$\frac{\Sigma(\ell) = \mathsf{lptr}\,(x^{n'}@\pi)}{(\Sigma, *\ell) \xrightarrow[\mathsf{exp}]{}{}^{n}_{\Delta} (\Sigma, x^{n'}@\pi)}\text{EVAL\_EXP\_DEREF\_LPTR}$$

$$\frac{\Psi \vdash^{\Delta;n}_{\mathsf{exp}} e : \mathsf{lptr}\langle\tau\rangle}{\Psi \vdash^{\Delta;n}_{\mathsf{exp}} *e : \tau}\text{TYPE\_EXP\_DEREF\_LPTR}$$

Choose $\Psi' = \Psi$.

We must show the following:

- $\vdash^{\Delta;n}_{\mathsf{ty}} \Psi\,\mathsf{ok}$: The size of the starting expression and the resulting expression are the same, so we have this condition by assumption.
- $\Psi \vdash_{\mathsf{st}} \Sigma\,\mathsf{ok}$: By assumption.
- $\Psi \vdash^{\Delta;n}_{\mathsf{exp}} x^{n'}@\pi : \tau$: Two inversions on $\Psi \vdash^{\Delta;n}_{\mathsf{exp}} *\ell : \tau$ (TYPE_EXP_DEREF_LPTR and TYPE_EXP_LOC) give us $\Psi(\ell) = \mathsf{lptr}\langle\tau\rangle$. Inversion on $\Psi \vdash_{\mathsf{st}} \Sigma\,\mathsf{ok}$ must therefore give us $\Psi; \Sigma \vdash_{\mathsf{st1}} \ell : \mathsf{lptr}\langle\tau\rangle\,\mathsf{ok}$. Because we know that $\Sigma(\ell) = \mathsf{lptr}\,(x^{n'}@\pi)$, $\Psi; \Sigma \vdash_{\mathsf{st1}} \ell : \mathsf{lptr}\langle\tau\rangle\,\mathsf{ok}$ must be derived by using CONS_BINDING_LPTR. The premises of this rule state $\Psi(x^{n'}@\pi) = \tau$ and $n' \leq n''$, where n'' is the index on l. From the premises of TYPE_EXP_LOC, we get that $n'' \leq n$. Therefore, we have the two premises of TYPE_EXP_LOC, and we can derive $\Psi \vdash^{\Delta;n}_{\mathsf{exp}} x^{n'}@\pi : \tau$ as desired.
- $\Psi \subseteq_n \Psi$: By reflexivity of $\subseteq_n$.

Cases EVAL_EXP_DEREF_PTR_NULL, EVAL_EXP_DEREF_LPTR_NULL, EVAL_EXP_FLD_ERR, EVAL_EXP_METH_ERR2, EVAL_EXP_ADDR_ERROR, EVAL_EXP_DEREF_ERROR

Choose $\Psi' = \Psi$.

We must show the following:

- $\vdash^{\Delta;n}_{\mathsf{ty}} \Psi\,\mathsf{ok}$: The size of the starting expression and the resulting expression are the same, so we have this condition by assumption.
- $\Psi \vdash_{\mathsf{st}} \Sigma\,\mathsf{ok}$: By assumption.
- $\Psi \vdash^{\Delta;n}_{\mathsf{exp}} \mathsf{error} : \tau$: By TYPE_EXP_ERR.
- $\Psi \subseteq_n \Psi$: By reflexivity of $\subseteq_n$.

Cases EVAL_EXP_FLD_CONG, EVAL_EXP_METH_CONG2, EVAL_EXP_ADDR_CONG, and EVAL_EXP_DEREF_CONG

In this case explanation, we use e_1 to refer to the inner expression and e_1' to refer to that expression after it has stepped. e and e' refer to the outer expression.

We note that $|e_1| = |e|$, and we can apply the induction hypothesis immediately. We get the following about some Ψ_1:
$\vdash_{\mathsf{ty}}^{\Delta;|e_1'|+n} \Psi_1\ \mathsf{ok}$
$\Psi_1 \vdash_{\mathsf{st}} \Sigma'\ \mathsf{ok}$
$\Psi_1 \vdash_{\mathsf{exp}}^{\Delta;n} e_1' : \tau_1$ where T is defined by $\Psi \vdash_{\mathsf{exp}}^{\Delta;n} e_1 : \tau_1$
$\Psi \subseteq_n \Psi_1$

Choose $\Psi' = \Psi_1$.

We must show the following:

- $\vdash_{\mathsf{ty}}^{\Delta;|e'|+n} \Psi'\ \mathsf{ok}$: We note that $|e_1'| = |e'|$, so this condition is immediate from the IH.
- $\Psi' \vdash_{\mathsf{st}} \Sigma'\ \mathsf{ok}$: Immediate from IH.
- $\Psi' \vdash_{\mathsf{exp}}^{\Delta;n} e' : \tau$: Use $\Psi' \vdash_{\mathsf{exp}}^{\Delta;n} e_1' : \tau_1$ from the IH with the appropriate type case.
- $\Psi \subseteq_n \Psi'$: Immediate from IH.

Case EVAL_EXP_METH_CONG1

$$\frac{(\Sigma, e_1) \xrightarrow[\mathsf{exp}]{}{}^{n}_{\Delta} (\Sigma', e_1')}{(\Sigma, e_1.f(e_2)) \xrightarrow[\mathsf{exp}]{}{}^{n}_{\Delta} (\Sigma', e_1'.f(e_2))}\text{EVAL\_EXP\_METH\_CONG1}$$

$$\frac{\begin{array}{l}\Psi \vdash_{\mathsf{exp}}^{\Delta;n} e_1 : C \\ \mathsf{struct}\ C\ \{\ fldsmeths\};\ \in\ \Delta \\ \tau_1\ f(\tau_2\ x)\{vardecls; s; \mathsf{return}\ e\}\ \in\ meths \\ \Psi \vdash_{\mathsf{exp}}^{\Delta;n} e_2 : \tau_2\end{array}}{\Psi \vdash_{\mathsf{exp}}^{\Delta;n} e_1.f(e_2) : \tau_1}\text{TYPE\_EXP\_METH}$$

By the well-formedness restriction, we know $\mathsf{source}\,(e_2)$. Therefore, by lemma 26, we know that $|e_2| = 0$. Thus, $|e_1.f(e_2)| = |e_1|$. With this fact, we can now apply the induction hypothesis to get the following about some Ψ_1:
$\vdash_{\mathsf{ty}}^{\Delta;|e_1'|+n} \Psi_1\ \mathsf{ok}$
$\Psi_1 \vdash_{\mathsf{st}} \Sigma'\ \mathsf{ok}$
$\Psi_1 \vdash_{\mathsf{exp}}^{\Delta;n} e_1' : C$
$\Psi \subseteq_n \Psi_1$

Choose $\Psi' = \Psi_1$.

We must show the following:

- $\vdash_{\mathsf{ty}}^{\Delta;|e_1'.f(e_2)|+n} \Psi'\ \mathsf{ok}$: We note that $|e_1'.f(e_2)| = |e_1'|$ by the same reasoning as above, and we are done.

- $\Psi' \vdash_{\mathsf{st}} \Sigma'\,\mathsf{ok}$: Immediate from IH.
- $\Psi' \vdash^{\Delta;n}_{\mathsf{exp}} e_1'.f(e_2) : \tau$: Immediate from IH and TYPE_EXP_METH.
- $\Psi \subseteq_n \Psi'$: Immediate from IH.

Case EVAL_EXP_METH_ERR1

$$\frac{}{(\Sigma, \mathsf{error}.f(e_2)) \xrightarrow[\mathsf{exp}]{}{}^{n}_{\Delta} (\Sigma, \mathsf{error})}\text{EVAL\_EXP\_METH\_ERR1}$$

Choose $\Psi' = \Psi$.

We must show the following:

- $\vdash^{\Delta;n}_{\mathsf{ty}} \Psi\,\mathsf{ok}$: Because $\mathsf{source}\,(s_2)$, $|\mathsf{error}.f(e_2)| = 0$ (using lemma 26), and we have this by assumption.
- $\Psi \vdash_{\mathsf{st}} \Sigma\,\mathsf{ok}$: By assumption.
- $\Psi \vdash^{\Delta;n}_{\mathsf{exp}} \mathsf{error} : \tau$: By TYPE_EXP_ERR.
- $\Psi \subseteq_n \Psi$: By reflexivity of $\subseteq_n$.

Case EVAL_EXP_BODY_CONG1

$$\frac{(\Sigma, s) \xrightarrow[\mathsf{stmt}]{}{}^{n+1}_{\Delta} (\Sigma', s')}{(\Sigma, \{s;\, \mathsf{return}\ e\}) \xrightarrow[\mathsf{exp}]{}{}^{n}_{\Delta} (\Sigma', \{s';\, \mathsf{return}\ e\})}\text{EVAL\_EXP\_BODY\_CONG1}$$

First, we must ensure the premises of the induction hypothesis. That is, we must show $\vdash^{\Delta;|s|+1+n}_{\mathsf{ty}} \Psi\,\mathsf{ok}$. By lemma 26, $|\{s;\, \mathsf{return}\ e\}| = \mathsf{max}\,(|s|, |e|) + 1 = |s| + 1$, and so we have this condition by assumption.

Now, we must show that $\Psi \vdash^{\Delta;n+1}_{\mathsf{stmt}} s\,\mathsf{ok}$. This fact comes directly from inversion on TYPE_EXP_BODY.

Thus, our induction hypothesis tells us the following for some Ψ_1:
$\vdash^{\Delta;|s'|+1+n}_{\mathsf{ty}} \Psi_1\,\mathsf{ok}$
$\Psi_1 \vdash_{\mathsf{st}} \Sigma'\,\mathsf{ok}$
$\Psi_1 \vdash^{\Delta;n+1}_{\mathsf{stmt}} s'\,\mathsf{ok}$
$\Psi \subseteq_{n+1} \Psi_1$

Choose $\Psi' = \Psi_1$.

We must show the following:

- $\vdash^{\Delta;|\{s';\, \mathsf{return}\ e\}|+n}_{\mathsf{ty}} \Psi'\,\mathsf{ok}$: Using reasoning similar to that presented above, $|\{s';\, \mathsf{return}\ e\}| = |s'| + 1$. Thus, we have this condition from the IH.
- $\Psi' \vdash_{\mathsf{st}} \Sigma'\,\mathsf{ok}$: Immediate from IH.

- $\Psi' \vdash^{\Delta;n}_{\mathsf{exp}} \{s'; \mathsf{return}\ e\} : \tau$: All we need to show is $\Psi' \vdash^{\Delta;n+1}_{\mathsf{exp}} e : \tau$. We know $\Psi \vdash^{\Delta;n+1}_{\mathsf{exp}} e : \tau$. By lemma 20, (noting that we know $\mathsf{source}\,(e)$ from the well-formedness condition) we can conclude $\Psi' \vdash^{\Delta;n+1}_{\mathsf{exp}} e : \tau$ as desired.
- $\Psi \subseteq_n \Psi'$: Immediate from lemma 18.

Case EVAL_EXP_BODY_CONG2

$$\frac{(\Sigma, e) \xrightarrow[\mathsf{exp}]{}^{n+1}_{\Delta} (\Sigma', e')}{(\Sigma, \{\mathsf{skip}; \mathsf{return}\ e\}) \xrightarrow[\mathsf{exp}]{}^{n}_{\Delta} (\Sigma', \{\mathsf{skip}; \mathsf{return}\ e'\})}\text{EVAL\_EXP\_BODY\_CONG2}$$

Using reasoning like that of the previous case, we find that $|\{\mathsf{skip}; \mathsf{return}\ e\}| = |e| + 1$. We can then use the induction hypothesis to get the following for some Ψ_1:
Ψ_1: $\vdash^{\Delta;|e'|+1+n}_{\mathsf{ty}} \Psi_1\ \mathsf{ok}$
$\Psi_1 \vdash_{\mathsf{st}} \Sigma'\ \mathsf{ok}$
$\Psi_1 \vdash^{\Delta;n+1}_{\mathsf{exp}} e' : \tau$
$\Psi \subseteq_{n+1} \Psi_1$

Choose $\Psi' = \Psi_1$.

We must show the following:

- $\vdash^{\Delta;|\{\mathsf{skip}; \mathsf{return}\ e'\}|+n}_{\mathsf{ty}} \Psi'\ \mathsf{ok}$: For reasons similar to those above, $|\{\mathsf{skip}; \mathsf{return}\ e'\}| = |e'| + 1$, making this statement immediate from IH.
- $\Psi' \vdash_{\mathsf{st}} \Sigma'\ \mathsf{ok}$: Immediate from IH.
- $\Psi' \vdash^{\Delta;n}_{\mathsf{exp}} \{\mathsf{skip}; \mathsf{return}\ e'\} : \tau$: We know $\Psi' \vdash^{\Delta;n+1}_{\mathsf{stmt}} \mathsf{skip}\ \mathsf{ok}$ from TYPE_STMT_SKIP, and we know $\Psi' \vdash^{\Delta;n+1}_{\mathsf{exp}} e' : \tau$ from the IH. Noting that τ has not changed (and by inversion on $\Psi \vdash^{\Delta;n}_{\mathsf{exp}} \{\mathsf{skip}; \mathsf{return}\ e\} : \tau$ does not equal $\mathsf{lptr}\langle\tau'\rangle$), we can apply TYPE_EXP_BODY.
- $\Psi \subseteq_n \Psi'$: Immediate from lemma 18.

Cases EVAL_EXP_BODY_ERR1 **and** EVAL_EXP_BODY_ERR2 Choose $\Psi' = \Psi$.

We must show the following:

- $\vdash^{\Delta;n}_{\mathsf{ty}} \Psi\ \mathsf{ok}$: We note that $|\mathsf{error}| \leq |\{\mathsf{error}; \mathsf{return}\ e\}|$ and $|\mathsf{error}| \leq |\{\mathsf{skip}; \mathsf{return}\ \mathsf{error}\}|$. We use lemma 18.
- $\Psi \vdash_{\mathsf{st}} \Sigma\ \mathsf{ok}$: By assumption.
- $\Psi \vdash^{\Delta;n}_{\mathsf{exp}} \mathsf{error} : \tau$: By TYPE_EXP_ERR.
- $\Psi \subseteq_n \Psi$: By reflexivity of $\subseteq_n$.

□

Corollary (Pointer lifetime invariant is preserved)**.** *If* $\mathsf{wf}\,(\Delta, \Psi, \Sigma, k)$, $\Psi \vdash^{\Delta;n}_{\mathsf{stmt}} s\ \mathsf{ok}$, *and* $(\Sigma, s) \xrightarrow[\mathsf{stmt}]{}^{n}_{\Delta} (\Sigma', s')$, *where* k *is the maximum stack height used within the statement* s, *then the pointer lifetime invariant holds in the store* Σ'.

Proof. The Preservation theorem says that if a statement is well-formed in a consistent context $\Psi; \Sigma$, then the context after evaluation $\Psi'; \Sigma'$ is also consistent. In particular, the judgment $\Psi'; \Sigma' \vdash_{\mathsf{st1}} \ell : \tau\,\mathsf{ok}$ must hold for every binding in Ψ'. By the statement of the pointer lifetime invariant, we care only about non-null pointers; thus, the following two rules are the only ones of interest:

$$\frac{\Sigma(x^n@\,) = \mathsf{lptr}\,(x'^{n'}@\pi') \qquad n' \leq n \qquad \Psi(x'^{n'}@\pi') = \tau}{\Psi; \Sigma \vdash_{\mathsf{st1}} x^n@\, : \mathsf{lptr}\langle\tau\rangle\,\mathsf{ok}}\text{CONS\_BINDING\_LPTR}$$

$$\frac{\Sigma(x^n@\pi) = \mathsf{ptr}\,(x'^{n'}@\pi') \qquad n' = 0 \qquad \Psi(x'^{n'}@\pi') = \tau}{\Psi; \Sigma \vdash_{\mathsf{st1}} x^n@\pi : \mathsf{ptr}\langle\tau\rangle\,\mathsf{ok}}\text{CONS\_BINDING\_PTR}$$

By inspection, we can see that these enforce the pointer lifetime invariant. Thus, the pointer lifetime invariant is maintained during evaluation, as desired. □

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[2] C. DeLozier, R. Eisenberg, S. Nagarakatte, P.-M. Osera, M. M. K. Martin, and S. Zdancewic. Ironclad C++: A library-augmented type-safe subset of C++. In *Proceedings of the ACM international conference on Object-oriented programming, systems, languages, and applications (OOPSLA)*, Oct. 2013.

[3] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *Proceedings of the SIGPLAN 2002 Conference on Programming Language Design and Implementation*, June 2002.

[4] A. Igarashi, B. Pierce, and P. Wadler. Featherweight java: a minimal core calculus for Java and GJ. In *Proceedings of the 14th SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Application (OOPSLA)*, Oct. 1999.

[5] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proceedings of the 2002 USENIX Annual Technical Conference*, June 2002.

[6] J. Jonathan G. Rossie and D. P. Friedman. An algebraic semantics of subobjects. In *Proceedings of the 17th SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Application (OOPSLA)*, Nov. 2002.

[7] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. Softbound: Highly compatible and complete spatial memory safety for C. In *Proceedings of the SIGPLAN 2009 Conference on Programming Language Design and Implementation*, June 2009.

[8] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. CETS: Compiler enforced temporal safety for C. In *Proceedings of the 2010 International Symposium on Memory Management*, June 2010.

[9] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy software. *ACM Trans. Prog. Lang. Syst.*, 27(3), May 2005.

[10] T. Ramananandro, G. D. Reis, and X. Leroy. Formal verification of object layout for C++ multiple inheritance. In *Proceedings of The 38th ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages (POPL)*, Jan. 2011.

[11] D. Wasserrab, T. Nipkow, G. Snelting, and F. Tip. An operational semantics and type safety proof for multiple inheritance in C++. In *Proceedings of the 21st SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Application (OOPSLA)*, Oct. 2006.

A. Core Ironclad Language Definition

A.1. Syntax

Types	τ	$::=$	$\mathsf{ptr}\langle\tau\rangle \mid \mathsf{lptr}\langle\tau\rangle \mid C$
Surface exprs	e	$::=$	$x \mid \mathsf{null} \mid e.x \mid e_1.f(e_2) \mid \mathsf{new}\, C() \mid \&e \mid *e$
Internal exprs	e	$::=$	$\ell \mid \{s; \mathsf{return}\, e\} \mid \mathsf{error}$
Statements	s	$::=$	$e_1 = e_2 \mid s_1; s_2 \mid \mathsf{skip} \mid \mathsf{error}$
Class decls	cls	$::=$	$\mathsf{struct}\, C\, \{\, fldsmeths\};$
Fields	$flds$	$::=$	$\tau_1\, x_1;\, ..\, \tau_m\, x_m;$
Methods	$meth$	$::=$	$\tau_1\, f(\tau_2\, x)\{\overline{\tau_i\, x_i}^{\,i}; s; \mathsf{return}\, e\}$
Class context	Δ	$::=$	$\{cls_1\, ..\, cls_m\}$
Programs	$prog$	$::=$	$\Delta; \mathsf{void\ main}()\ \{s\}$
Locations	ℓ	$::=$	$x^n @ y_1\, ..\, y_m$
Pointer values	pv	$::=$	$\ell \mid \mathsf{bad\_ptr}$
Values	v	$::=$	$\mathsf{ptr}\,(pv) \mid \mathsf{lptr}\,(pv) \mid C$
Store	Σ	$::=$	$\cdot \mid \Sigma\,[\ell \mapsto v]$
Store typing	Ψ	$::=$	$\cdot \mid \Psi\,[\ell : \tau]$

A.2. Auxiliary Definitions

$\boxed{\Sigma = \mathsf{default}_\Delta\langle\tau\rangle(\ell)}$ Default store

$$\frac{}{[\ell \mapsto \mathsf{ptr}\,(\mathsf{bad\_ptr})] = \mathsf{default}_\Delta\langle\mathsf{ptr}\langle\tau\rangle\rangle(\ell)}\,\text{AUX\_DEFAULT\_PTR}$$

$$\frac{}{[\ell \mapsto \mathsf{lptr}\,(\mathsf{bad\_ptr})] = \mathsf{default}_\Delta\langle\mathsf{lptr}\langle\tau\rangle\rangle(\ell)}\,\text{AUX\_DEFAULT\_LPTR}$$

$$\frac{\begin{array}{l}\mathsf{struct}\ C\ \{\ fldsmeths\};\in\ \Delta \\ \overline{\tau_i\ x_i;}^{\,i} = flds \\ \overline{\Sigma_i = \mathsf{default}_\Delta\langle\tau_i\rangle(x^n@\pi +\!\!+ x_i)}^{\,i} \\ \Sigma = [x^n@\pi \mapsto C]\,\overline{\Sigma_i}^{\,i}\end{array}}{\Sigma = \mathsf{default}_\Delta\langle C\rangle(x^n@\pi)}\text{AUX\_DEFAULT\_CLASS}$$

$\boxed{\Psi = \mathsf{build\_types}_\Delta\langle\tau\rangle(\ell)}$ Default store typing

$$\frac{}{[\ell : \mathsf{lptr}\langle\tau\rangle] = \mathsf{build\_types}_\Delta\langle\mathsf{lptr}\langle\tau\rangle\rangle(\ell)}\text{AUX\_BUILD\_TYPES\_LPTR}$$

$$\frac{}{[\ell : \mathsf{ptr}\langle\tau\rangle] = \mathsf{build\_types}_\Delta\langle\mathsf{ptr}\langle\tau\rangle\rangle(\ell)}\text{AUX\_BUILD\_TYPES\_PTR}$$

$$\frac{\begin{array}{l}\mathsf{struct}\ C\ \{\ fldsmeths\};\in\ \Delta \\ \overline{\tau_i\ x_i;}^{\,i} = flds \\ \overline{\Psi_i = \mathsf{build\_types}_\Delta\langle\tau_i\rangle(x^n@\pi +\!\!+ x_i)}^{\,i} \\ \Psi = [x^n@\pi : C]\,\overline{\Psi_i}^{\,i}\end{array}}{\Psi = \mathsf{build\_types}_\Delta\langle C\rangle(x^n@\pi)}\text{AUX\_BUILD\_TYPES\_CLASS}$$

$\boxed{\Sigma' = \mathsf{copy\_store}(\Sigma, \ell, base)}$ Copy store

$$\frac{}{\cdot = \mathsf{copy\_store}(\cdot, \ell, base)}\text{AUX\_COPY\_STORE\_EMPTY}$$

$$\frac{\Sigma' = \mathsf{copy\_store}(\Sigma, {x_1}^{n_1}@\pi_1, {x_2}^{n_2})}{\Sigma'\,[{x_2}^{n_2}@\pi \mapsto v] = \mathsf{copy\_store}(\Sigma\,[{x_1}^{n_1}@\pi_1 +\!\!+ \pi \mapsto v], {x_1}^{n_1}@\pi_1, {x_2}^{n_2})}\text{AUX\_COPY\_STORE\_BIND\_MATCH}$$

$$\frac{\begin{array}{l}\Sigma' = \mathsf{copy\_store}(\Sigma, x^n@\pi, base) \\ x^n@\pi \neq {x'}^{n'}@\pi' \qquad \pi \text{ is not a prefix of } \pi' +\!\!+ \pi''\end{array}}{\Sigma' = \mathsf{copy\_store}(\Sigma\,[{x'}^{n'}@\pi' +\!\!+ \pi'' \mapsto v], x^n@\pi, base)}\text{AUX\_COPY\_STORE\_BIND\_NO\_MATCH}$$

$\boxed{\Psi' = \mathsf{copy\_types}(\Psi, \ell, base)}$ Copy types

$$\frac{}{\cdot = \mathsf{copy\_types}(\cdot, \ell, base)}\text{AUX\_COPY\_TYPES\_EMPTY}$$

$$\frac{\Psi' = \mathsf{copy\_types}(\Psi, {x_1}^{n_1}@\pi_1, {x_2}^{n_2})}{\Psi'\left[{x_2}^{n_2}@\pi : \tau\right] = \mathsf{copy\_types}(\Psi\left[{x_1}^{n_1}@\pi_1 +\!\!+ \pi : \tau\right], {x_1}^{n_1}@\pi_1, {x_2}^{n_2})}\text{AUX\_COPY\_TYPES\_BIND\_MATCH}$$

$$\frac{\begin{array}{l}\Psi' = \mathsf{copy\_types}(\Psi, x^n@\pi, base) \\ x^n@\pi \neq {x'}^{n'}@\pi' \qquad\qquad \pi \text{ is not a prefix of } \pi' +\!\!+ \pi''\end{array}}{\Psi' = \mathsf{copy\_types}(\Psi\left[{x'}^{n'}@\pi' +\!\!+ \pi'' : \tau\right], x^n@\pi, base)}\text{AUX\_COPY\_TYPES\_BIND\_NO\_MATCH}$$

$\boxed{k = |s|}$ Size of statements

$$\frac{k_1 = |e_1| \qquad k_2 = |e_2| \qquad k' = \mathsf{max}\left(k_1, k_2\right)}{k' = |e_1 = e_2|}\text{AUX\_STMT\_SIZEASSIGN}$$

$$\frac{k_1 = |s_1| \qquad k_2 = |s_2| \qquad k' = \mathsf{max}\left(k_1, k_2\right)}{k' = |s_1; s_2|}\text{AUX\_STMT\_SIZESEQ}$$

$$\frac{}{0 = |\mathsf{skip}|}\text{AUX\_STMT\_SIZESKIP}$$

$$\frac{}{0 = |\mathsf{error}|}\text{AUX\_STMT\_SIZEERROR}$$

$\boxed{k = |e|}$ Size of expressions

$$\frac{}{0 = |x|}\text{AUX\_EXP\_SIZE\_VAR}$$

$$\frac{}{0 = |\mathsf{null}|}\text{AUX\_EXP\_SIZE\_NULL}$$

$$\frac{k = |e|}{k = |e.x|}\text{AUX\_EXP\_SIZE\_FLD}$$

$$\frac{k_1 = |e_1| \qquad k_2 = |e_2| \qquad k' = \mathsf{max}\left(k_1, k_2\right)}{k' = |e_1.f\,(e_2)|}\text{AUX\_EXP\_SIZE\_METH}$$

$$\frac{}{0 = |\mathsf{new}\ C()|}\text{AUX\_EXP\_SIZE\_NEW}$$

$$\frac{k = |e|}{k = |\&e|}\text{AUX\_EXP\_SIZE\_ADDR}$$

$$\frac{k = |e|}{k = |*e|}\text{AUX\_EXP\_SIZE\_DEREF}$$

$$\frac{}{0 = |\mathsf{error}|}\text{AUX\_EXP\_SIZE\_ERROR}$$

$$\frac{}{0 = |\ell|}\text{AUX\_EXP\_SIZE\_LOC}$$

$$\frac{\begin{array}{l}k_1 = |s| \\ k_2 = |e| \\ k' = \mathsf{max}\,(k_1, k_2) + 1\end{array}}{k' = |\{s;\ \mathsf{return}\ e\}|}\text{AUX\_EXP\_SIZE\_BODY}$$

$\boxed{\Sigma' = \Sigma \backslash n}$ Store quotienting

$$\frac{}{\cdot = \cdot \backslash n}\text{AUX\_STORE\_QUOT\_EMPTY}$$

$$\frac{\Sigma' = \Sigma \backslash n}{\Sigma' = \Sigma\,[x^n @ \pi \mapsto v] \backslash n}\text{AUX\_STORE\_QUOT\_REMOVE\_VALUE}$$

$$\frac{\Sigma' = \Sigma \backslash n}{\Sigma' = \Sigma\,[x^n @ \pi \mapsto C] \backslash n}\text{AUX\_STORE\_QUOT\_REMOVE\_TAG}$$

$$\frac{\Sigma' = \Sigma \backslash n \qquad n \neq n'}{\Sigma'\,[x^{n'} @ \pi \mapsto v] = \Sigma\,[x^{n'} @ \pi \mapsto v] \backslash n}\text{AUX\_STORE\_QUOT\_KEEP\_VALUE}$$

$$\frac{\Sigma' = \Sigma \backslash n \qquad n \neq n'}{\Sigma' \left[x^{n'} @ \pi \mapsto C\right] = \Sigma \left[x^{n'} @ \pi \mapsto C\right] \backslash n} \text{AUX\_STORE\_QUOT\_KEEP\_TAG}$$

$\boxed{\Psi' = \Psi \backslash n}$ Type context quotienting

$$\frac{}{\cdot = \cdot \backslash n} \text{AUX\_TYPE\_QUOT\_EMPTY}$$

$$\frac{\Psi' = \Psi \backslash n}{\Psi' = \Psi \left[x^{n} @ \pi : \tau\right] \backslash n} \text{AUX\_TYPE\_QUOT\_REMOVE}$$

$$\frac{\Psi' = \Psi \backslash n \qquad n \neq n'}{\Psi' \left[x^{n'} @ \pi : \tau\right] = \Psi \left[x^{n'} @ \pi : \tau\right] \backslash n} \text{AUX\_TYPE\_QUOT\_KEEP}$$

A.3. Store Consistency

$\boxed{\mathsf{wf}\,(\Delta, \Psi, \Sigma, n)}$ Well-formedness summary judgement

$$\frac{\vdash_{\mathsf{env}} \Delta\,\mathsf{ok} \qquad \vdash_{\mathsf{ty}}^{\Delta;n} \Psi\,\mathsf{ok} \qquad \Psi \vdash_{\mathsf{st}} \Sigma\,\mathsf{ok}}{\mathsf{wf}\,(\Delta, \Psi, \Sigma, n)} \text{CONS\_WFSUMMARY\_WF}$$

$\boxed{\vdash_{\mathsf{ty}}^{\Delta;n} \Psi\,\mathsf{ok}}$ Type context consistency

$$\frac{\dfrac{\Psi = \overline{[\ell_i : \tau_i]}^{\,i}}{\Psi \vdash_{\mathsf{ty1}}^{\Delta;n} \ell_i : \tau_i\,\mathsf{ok}}\,i}{\vdash_{\mathsf{ty}}^{\Delta;n} \Psi\,\mathsf{ok}} \text{CONS\_TYPE\_BINDINGS}$$

$\boxed{\Psi \vdash_{\mathsf{ty1}}^{\Delta;n} \ell : \tau\,\mathsf{ok}}$ Type binding consistency

$$\frac{n' \leq n}{\Psi \vdash_{\mathsf{ty1}}^{\Delta;n} x^{n'} @ \; : \mathsf{lptr}\langle\tau\rangle\,\mathsf{ok}} \text{CONS\_TYPE\_BINDING\_LPTR}$$

$$\frac{\begin{array}{l}\Psi(x^{n'} @ \pi) = C \\ \mathsf{struct}\; C\,\{\,fldsmeths\}; \in\; \Delta \\ \mathsf{ptr}\langle\tau\rangle\, x'; \in\; flds\end{array}}{\Psi \vdash_{\mathsf{ty1}}^{\Delta;n} x^{n'} @ \pi +\!\!+ x' : \mathsf{ptr}\langle\tau\rangle\,\mathsf{ok}} \text{CONS\_TYPE\_BINDING\_FIELD\_PTR}$$

$$\frac{n' \leq n}{\Psi \vdash^{\Delta;n}_{\mathsf{ty1}} x^{n'}@ \ : \mathsf{ptr}\langle\tau\rangle\ \mathsf{ok}}\text{CONS\_TYPE\_BINDING\_LOCAL\_PTR}$$

$$\frac{\begin{array}{l}\Psi(x^{n'}@\pi) = C \\ \mathsf{struct}\ C\ \{\ fldsmeths\}; \in\ \Delta \\ C'\ x'; \in\ flds \\ \mathsf{struct}\ C'\ \{\ flds'meths'\}; \in\ \Delta \\ \overline{\tau_i\ y_i;}^{\,i} = flds' \\ \pi' = \pi +\!\!+ x' \qquad \overline{\Psi(x^{n'}@\pi' +\!\!+ y_i) = \tau_i}^{\,i}\end{array}}{\Psi \vdash^{\Delta;n}_{\mathsf{ty1}} x^{n'}@\pi +\!\!+ x' : C'\ \mathsf{ok}}\text{CONS\_TYPE\_BINDING\_FIELD\_CLASS}$$

$$\frac{\begin{array}{l}n' \leq n \\ \mathsf{struct}\ C\ \{\ fldsmeths\}; \in\ \Delta \\ \overline{\tau_i\ x_i;}^{\,i} = flds \\ \overline{\Psi(x^{n'}@x_i) = \tau_i}^{\,i}\end{array}}{\Psi \vdash^{\Delta;n}_{\mathsf{ty1}} x^{n'}@\ : C\ \mathsf{ok}}\text{CONS\_TYPE\_BINDING\_LOCAL\_CLASS}$$

$\boxed{\Psi \vdash_{\mathsf{st}} \Sigma\ \mathsf{ok}}$ Store consistency

$$\frac{\begin{array}{l}\Psi = \overline{[\ell_i : \tau_i]}^{\,i} \\ \overline{\Psi; \Sigma \vdash_{\mathsf{st1}} \ell_i : \tau_i\ \mathsf{ok}}^{\,i} \\ dom\,(\Sigma)\ \subseteq\ dom\,(\Psi)\end{array}}{\Psi \vdash_{\mathsf{st}} \Sigma\ \mathsf{ok}}\text{CONS\_STORE\_BINDINGS}$$

$\boxed{\Psi; \Sigma \vdash_{\mathsf{st1}} \ell : \tau\ \mathsf{ok}}$ Binding consistency

$$\frac{\Sigma(x^n@\,) = \mathsf{lptr}\,(\mathsf{bad\_ptr})}{\Psi; \Sigma \vdash_{\mathsf{st1}} x^n@\ : \mathsf{lptr}\langle\tau\rangle\ \mathsf{ok}}\text{CONS\_BINDING\_LPTR\_NULL}$$

$$\frac{\begin{array}{ll}\Sigma(x^n@\,) = \mathsf{lptr}\,(x'^{n'}@\pi') & n' \leq n \\ \Psi(x'^{n'}@\pi') = \tau & \end{array}}{\Psi; \Sigma \vdash_{\mathsf{st1}} x^n@\ : \mathsf{lptr}\langle\tau\rangle\ \mathsf{ok}}\text{CONS\_BINDING\_LPTR}$$

$$\frac{\Sigma(x^n@\pi) = \mathsf{ptr}\,(\mathsf{bad\_ptr})}{\Psi; \Sigma \vdash_{\mathsf{st1}} x^n@\pi : \mathsf{ptr}\langle\tau\rangle\ \mathsf{ok}}\text{CONS\_BINDING\_PTR\_NULL}$$

$$\frac{\Sigma(x^n@\pi) = \mathsf{ptr}\,(x'^{n'}@\pi') \qquad n' = 0 \qquad \Psi(x'^{n'}@\pi') = \tau}{\Psi; \Sigma \vdash_{\mathsf{st1}} x^n@\pi : \mathsf{ptr}\langle\tau\rangle\,\mathsf{ok}}\text{CONS\_BINDING\_PTR}$$

$$\frac{\Sigma(x^n@\pi) = C}{\Psi; \Sigma \vdash_{\mathsf{st1}} x^n@\pi : C\,\mathsf{ok}}\text{CONS\_BINDING\_CLS}$$

A.4. Well-formedness of Terms

$\boxed{\mathsf{source}\,(s)}$ Statement source predicate

$$\frac{\mathsf{source}\,(e_1) \qquad \mathsf{source}\,(e_2)}{\mathsf{source}\,(e_1 = e_2)}\text{SSRC\_ASSIGN}$$

$$\frac{\mathsf{source}\,(s_1) \qquad \mathsf{source}\,(s_2)}{\mathsf{source}\,(s_1; s_2)}\text{SSRC\_SEQ}$$

$$\frac{}{\mathsf{source}\,(\mathsf{error})}\text{SSRC\_ERR}$$

$$\frac{}{\mathsf{source}\,(\mathsf{skip})}\text{SSRC\_SKIP}$$

$\boxed{\mathsf{source}\,(e)}$ Expression source predicate

$$\frac{}{\mathsf{source}\,(x)}\text{ESRC\_VAR}$$

$$\frac{}{\mathsf{source}\,(\mathsf{null})}\text{ESRC\_NULL}$$

$$\frac{\mathsf{source}\,(e)}{\mathsf{source}\,(e.x)}\text{ESRC\_FLD}$$

$$\frac{\mathsf{source}\,(e_1) \qquad \mathsf{source}\,(e_2)}{\mathsf{source}\,(e_1.f(e_2))}\text{ESRC\_METH}$$

$$\frac{}{\mathsf{source}\,(\mathsf{new}\ C())}\text{ESRC\_NEW}$$

$$\frac{\mathsf{source}\,(e)}{\mathsf{source}\,(\&e)}\text{ESRC\_ADDR}$$

$$\frac{\mathsf{source}\,(e)}{\mathsf{source}\,(*e)}\text{ESRC\_DEREF}$$

$$\frac{}{\mathsf{source}\,(\mathsf{error})}\text{ESRC\_ERR}$$

$\boxed{\Delta\,\mathsf{wf}}$ Well-formed class context

$$\frac{\overline{cls_i\,\mathsf{wf}}^{\,i}}{\{\,\overline{cls_i}^{\,i}\,\}\,\mathsf{wf}}\text{DWF\_LIST}$$

$\boxed{cls\,\mathsf{wf}}$ Well-formed class

$$\frac{\overline{meth_i\,\mathsf{wf}}^{\,i}}{\mathsf{struct}\ C\,\{\,flds\overline{meth_i}^{\,i}\};\,\mathsf{wf}}\text{CLSWF\_DECL}$$

$\boxed{meth\,\mathsf{wf}}$ Well-formed method

$$\frac{\mathsf{source}\,(s) \qquad \mathsf{source}\,(e)}{\tau_1\ f(\tau_2\ x)\{vardecls;\,s;\mathsf{return}\ e\}\,\mathsf{wf}}\text{METHWF\_DECL}$$

$\boxed{s\,\mathsf{wf}}$ Well-formed statement

$$\frac{\begin{array}{l}e_1 \neq \ell_1\\ \mathsf{source}\,(e_2)\end{array}}{e_1 = e_2\,\mathsf{wf}}\text{SWF\_ASSIGN1}$$

$$\frac{}{\ell_1 = e_2\,\mathsf{wf}}\text{SWF\_ASSIGN2}$$

$$\frac{\mathsf{source}\,(s_2)}{s_1;\, s_2\;\mathsf{wf}}\,\text{SWF\_SEQ}$$

$$\frac{}{\mathsf{skip}\;\mathsf{wf}}\,\text{SWF\_SKIP}$$

$$\frac{}{\mathsf{error}\;\mathsf{wf}}\,\text{SWF\_ERR}$$

$\boxed{e\;\mathsf{wf}}$ Well-formed expression

$$\frac{}{x\;\mathsf{wf}}\,\text{EWF\_VAR}$$

$$\frac{}{\mathsf{null}\;\mathsf{wf}}\,\text{EWF\_NULL}$$

$$\frac{}{e.x\;\mathsf{wf}}\,\text{EWF\_FLD}$$

$$\frac{e_1 \neq \ell_1 \quad \mathsf{source}\,(e_2)}{e_1.f\,(e_2)\;\mathsf{wf}}\,\text{EWF\_METH1}$$

$$\frac{}{\ell_1.f\,(e_2)\;\mathsf{wf}}\,\text{EWF\_METH2}$$

$$\frac{}{\mathsf{new}\;C()\;\mathsf{wf}}\,\text{EWF\_NEW}$$

$$\frac{}{\&e\;\mathsf{wf}}\,\text{EWF\_ADDR}$$

$$\frac{}{*e\;\mathsf{wf}}\,\text{EWF\_DEREF}$$

$$\frac{}{\mathsf{error}\ \mathsf{wf}}\text{EWF\_ERR}$$

$$\frac{}{\ell\ \mathsf{wf}}\text{EWF\_LOC}$$

$$\frac{\begin{array}{l} s \neq \mathsf{skip} \\ \mathsf{source}\,(e) \end{array}}{\{s;\ \mathsf{return}\ e\}\ \mathsf{wf}}\text{EWF\_BODY1}$$

$$\frac{}{\{\mathsf{skip};\ \mathsf{return}\ e\}\ \mathsf{wf}}\text{EWF\_BODY2}$$

A.5. Typing

$\boxed{\vdash^{\Delta}_{\mathsf{twf}} \tau}$ Type well-formedness

$$\frac{\vdash^{\Delta}_{\mathsf{twf}} \tau}{\vdash^{\Delta}_{\mathsf{twf}} \mathsf{lptr}\langle\tau\rangle}\text{TYPE\_WELL\_FORMED\_LPTR}$$

$$\frac{\vdash^{\Delta}_{\mathsf{twf}} \tau}{\vdash^{\Delta}_{\mathsf{twf}} \mathsf{ptr}\langle\tau\rangle}\text{TYPE\_WELL\_FORMED\_PTR}$$

$$\frac{\mathsf{struct}\ C\ \{\ fldsmeths\};\ \in\ \Delta}{\vdash^{\Delta}_{\mathsf{twf}} C}\text{TYPE\_WELL\_FORMED\_CLASS}$$

$\boxed{\tau_1 \Leftarrow \tau_2}$ Assignable

$$\frac{}{\mathsf{ptr}\langle\tau\rangle \Leftarrow \mathsf{ptr}\langle\tau\rangle}\text{TYPE\_ASSIGNABLE\_PTR\_PTR}$$

$$\frac{}{\mathsf{lptr}\langle\tau\rangle \Leftarrow \mathsf{ptr}\langle\tau\rangle}\text{TYPE\_ASSIGNABLE\_LPTR\_PTR}$$

$$\frac{}{\mathsf{ptr}\langle\tau\rangle \Leftarrow \mathsf{lptr}\langle\tau\rangle}\text{TYPE\_ASSIGNABLE\_PTR\_LPTR}$$

$$\frac{}{\mathsf{lptr}\langle\tau\rangle \Leftarrow \mathsf{lptr}\langle\tau\rangle}\text{TYPE\_ASSIGNABLE\_LPTR\_LPTR}$$

$\boxed{\Psi \vdash^{\Delta;n}_{\mathsf{exp}} e : \tau}$ Expression typing

$$\frac{\Psi(x^n@\,) = \tau}{\Psi \vdash^{\Delta;n}_{\mathsf{exp}} x : \tau}\text{TYPE\_EXP\_VAR}$$

$$\frac{\begin{array}{l}\Psi \vdash^{\Delta;n}_{\mathsf{exp}} e : C \\ \mathsf{struct}\ C\ \{\ fldsmeths\};\ \in\ \Delta \\ \tau\ x;\ \in\ flds\end{array}}{\Psi \vdash^{\Delta;n}_{\mathsf{exp}} e.x : \tau}\text{TYPE\_EXP\_FLD}$$

$$\frac{\Psi \vdash^{\Delta;n}_{\mathsf{exp}} e : \mathsf{ptr}\langle\tau\rangle}{\Psi \vdash^{\Delta;n}_{\mathsf{exp}} *e : \tau}\text{TYPE\_EXP\_DEREF\_PTR}$$

$$\frac{\Psi \vdash^{\Delta;n}_{\mathsf{exp}} e : \mathsf{lptr}\langle\tau\rangle}{\Psi \vdash^{\Delta;n}_{\mathsf{exp}} *e : \tau}\text{TYPE\_EXP\_DEREF\_LPTR}$$

$$\frac{}{\Psi \vdash^{\Delta;n}_{\mathsf{exp}} \mathsf{error} : \tau}\text{TYPE\_EXP\_ERR}$$

$$\frac{\Psi(x^{n'}@\pi) = \tau \qquad n' \leq n}{\Psi \vdash^{\Delta;n}_{\mathsf{exp}} x^{n'}@\pi : \tau}\text{TYPE\_EXP\_LOC}$$

$$\frac{}{\Psi \vdash^{\Delta;n}_{\mathsf{exp}} \mathsf{null} : \mathsf{ptr}\langle\tau\rangle}\text{TYPE\_EXP\_NULL}$$

$$\frac{\begin{array}{l}\Psi \vdash^{\Delta;n}_{\mathsf{exp}} e_1 : C \\ \mathsf{struct}\ C\ \{\ fldsmeths\};\ \in\ \Delta \\ \tau_1\ f(\tau_2\ x)\{vardecls; s; \mathsf{return}\ e\}\ \in\ meths \\ \Psi \vdash^{\Delta;n}_{\mathsf{exp}} e_2 : \tau_2\end{array}}{\Psi \vdash^{\Delta;n}_{\mathsf{exp}} e_1.f(e_2) : \tau_1}\text{TYPE\_EXP\_METH}$$

$$\frac{\vdash^{\Delta}_{\mathsf{twf}} C}{\Psi \vdash^{\Delta;n}_{\mathsf{exp}} \mathsf{new}\ C() : \mathsf{ptr}\langle C\rangle}\text{TYPE\_EXP\_NEW}$$

$$\frac{\Psi \vdash^{\Delta;n}_{\mathsf{exp}} e : \tau}{\Psi \vdash^{\Delta;n}_{\mathsf{exp}} \&e : \mathsf{lptr}\langle\tau\rangle}\text{TYPE\_EXP\_ADDR}$$

$$\frac{\begin{array}{l}\forall\tau'.\tau \neq \mathsf{lptr}\langle\tau'\rangle \\ \Psi \vdash^{\Delta;n+1}_{\mathsf{stmt}} s\ \mathsf{ok} \qquad \Psi \vdash^{\Delta;n+1}_{\mathsf{exp}} e : \tau\end{array}}{\Psi \vdash^{\Delta;n}_{\mathsf{exp}} \{s;\ \mathsf{return}\ e\} : \tau}\text{TYPE\_EXP\_BODY}$$

$\boxed{\Psi \vdash^{\Delta;n}_{\mathsf{stmt}} s\ \mathsf{ok}}$ Statement consistency

$$\frac{\begin{array}{l}\Psi \vdash^{\Delta;n}_{\mathsf{exp}} e_1 : \tau_1 \\ \Psi \vdash^{\Delta;n}_{\mathsf{exp}} e_2 : \tau_2 \\ \tau_1 \Leftarrow \tau_2\end{array}}{\Psi \vdash^{\Delta;n}_{\mathsf{stmt}} e_1 = e_2\ \mathsf{ok}}\text{TYPE\_STMT\_ASSIGN}$$

$$\frac{\begin{array}{l}\Psi \vdash^{\Delta;n}_{\mathsf{stmt}} s_1\ \mathsf{ok} \\ \Psi \vdash^{\Delta;n}_{\mathsf{stmt}} s_2\ \mathsf{ok}\end{array}}{\Psi \vdash^{\Delta;n}_{\mathsf{stmt}} s_1; s_2\ \mathsf{ok}}\text{TYPE\_STMT\_SEQ}$$

$$\frac{}{\Psi \vdash^{\Delta;n}_{\mathsf{stmt}} \mathsf{skip}\ \mathsf{ok}}\text{TYPE\_STMT\_SKIP}$$

$$\frac{}{\Psi \vdash^{\Delta;n}_{\mathsf{stmt}} \mathsf{error}\ \mathsf{ok}}\text{TYPE\_STMT\_ERROR}$$

$\boxed{C \vdash^{\Delta}_{\mathsf{meth}} \mathit{meth}\ \mathsf{ok}}$ Method consistency

$$\frac{\begin{array}{l}\overline{x_i}^{\,i}, x, this \textbf{ distinct} \\ \vdash^{\Delta}_{\mathsf{twf}} \tau_2 \qquad \overline{\vdash^{\Delta}_{\mathsf{twf}} \tau_i}^{\,i} \\ \Psi_1 = \mathsf{build\_types}_{\Delta}\langle\tau_2\rangle(x^0@) \\ \Psi_2 = \mathsf{build\_types}_{\Delta}\langle\mathsf{lptr}\langle C\rangle\rangle(this^0@) \\ \overline{\Psi_{3\,i} = \mathsf{build\_types}_{\Delta}\langle\tau_i\rangle(x_i{}^0@)}^{\,i} \\ \Psi' = \Psi_1\,\Psi_2\,\overline{\Psi_{3\,i}}^{\,i} \qquad \Psi' \vdash^{\Delta;0}_{\mathsf{stmt}} s\,\mathsf{ok} \qquad \Psi' \vdash^{\Delta;0}_{\mathsf{exp}} e : \tau_1 \\ \forall \tau_1'.\tau_1 \neq \mathsf{lptr}\langle\tau_1'\rangle \qquad \mathsf{source}\,(s) \qquad \mathsf{source}\,(e)\end{array}}{C \vdash^{\Delta}_{\mathsf{meth}} \tau_1\, f(\tau_2\, x)\{\overline{\tau_i\, x_i}^{\,i}; s; \mathsf{return}\, e\}\,\mathsf{ok}}\text{TYPE\_METH\_DECL}$$

$\boxed{\vdash^{\Delta}_{\mathsf{class}} cls\,\mathsf{ok}}$ Class consistency

$$\frac{\begin{array}{l}flds = \overline{\tau_i\, x_i;}^{\,i} \\ \overline{\vdash^{\Delta}_{\mathsf{twf}} \tau_i}^{\,i} \\ \overline{\tau_i = \mathsf{ptr}\langle\tau_i'\rangle \vee \tau_i = C_i}^{\,i} \\ meths = \overline{meth_j}^{\,j} \\ \overline{C \vdash^{\Delta}_{\mathsf{meth}} meth_j\,\mathsf{ok}}^{\,j}\end{array}}{\vdash^{\Delta}_{\mathsf{class}} \mathsf{struct}\, C\,\{\, fldsmeths\};\,\mathsf{ok}}\text{TYPE\_CLS\_DECL}$$

$\boxed{\vdash_{\mathsf{env}} \Delta\,\mathsf{ok}}$ Class environment consistency

$$\frac{\begin{array}{l}\Delta = \{\,\overline{cls_i}^{\,i}\,\} \\ \overline{\vdash^{\Delta}_{\mathsf{class}} cls_i\,\mathsf{ok}}^{\,i} \\ \text{all names in } \Delta \text{ are unique within their scope} \\ \text{classes in } \Delta \text{ contain no cycles}\end{array}}{\vdash_{\mathsf{env}} \Delta\,\mathsf{ok}}\text{TYPE\_ENV\_DELTA}$$

$\boxed{\vdash_{\mathsf{prog}} prog\,\mathsf{ok}}$ Program typing

$$\frac{\begin{array}{l}\vdash_{\mathsf{env}} \Delta\,\mathsf{ok} \\ \cdot \vdash^{\Delta;0}_{\mathsf{stmt}} s\,\mathsf{ok}\end{array}}{\vdash_{\mathsf{prog}} \Delta; \mathsf{void\ main}()\ \{s\}\,\mathsf{ok}}\text{TYPE\_PROG\_MAIN}$$

A.6. Evaluation

$\boxed{(\Sigma, s) \underset{\mathsf{stmt}}{\longrightarrow}{}^{n}_{\Delta} (\Sigma', s')}$ Statement evaluation

$$\frac{\begin{array}{l}\Sigma(\ell_1) = \mathsf{ptr}\,(pv_1) \qquad \Sigma(\ell_2) = \mathsf{ptr}\,(pv_2) \\ \Sigma' = \Sigma\,[\ell_1 \mapsto \mathsf{ptr}\,(pv_2)]\end{array}}{(\Sigma, \ell_1 = \ell_2) \underset{\mathsf{stmt}}{\longrightarrow}^{n}_{\Delta} (\Sigma', \mathsf{skip})}\text{EVAL\_STMT\_ASSIGN\_PTR\_PTR}$$

$$\frac{\begin{array}{l}\Sigma(\ell_1) = \mathsf{lptr}\,(pv_1) \qquad \Sigma(\ell_2) = \mathsf{ptr}\,(pv_2) \\ \Sigma' = \Sigma\,[\ell_1 \mapsto \mathsf{lptr}\,(pv_2)]\end{array}}{(\Sigma, \ell_1 = \ell_2) \underset{\mathsf{stmt}}{\longrightarrow}^{n}_{\Delta} (\Sigma', \mathsf{skip})}\text{EVAL\_STMT\_ASSIGN\_LPTR\_PTR}$$

$$\frac{\begin{array}{l}\Sigma(\ell_1) = \mathsf{ptr}\,(pv_1) \qquad \Sigma(\ell_2) = \mathsf{lptr}\,(\mathsf{bad\_ptr}) \\ \Sigma' = \Sigma\,[\ell_1 \mapsto \mathsf{ptr}\,(\mathsf{bad\_ptr})]\end{array}}{(\Sigma, \ell_1 = \ell_2) \underset{\mathsf{stmt}}{\longrightarrow}^{n}_{\Delta} (\Sigma', \mathsf{skip})}\text{EVAL\_STMT\_ASSIGN\_PTR\_LPTR\_NULL}$$

$$\frac{\begin{array}{l}\Sigma(\ell_1) = \mathsf{ptr}\,(pv_1) \qquad \Sigma(\ell_2) = \mathsf{lptr}\,(x^0@\pi) \\ \Sigma' = \Sigma\,[\ell_1 \mapsto \mathsf{ptr}\,(x^0@\pi)]\end{array}}{(\Sigma, \ell_1 = \ell_2) \underset{\mathsf{stmt}}{\longrightarrow}^{n}_{\Delta} (\Sigma', \mathsf{skip})}\text{EVAL\_STMT\_ASSIGN\_PTR\_LPTR}$$

$$\frac{\begin{array}{l}\Sigma(\ell_1) = \mathsf{ptr}\,(pv_1) \qquad n' \neq 0 \\ \Sigma(\ell_2) = \mathsf{lptr}\,({x^{n'}}@\pi_2)\end{array}}{(\Sigma, \ell_1 = \ell_2) \underset{\mathsf{stmt}}{\longrightarrow}^{n}_{\Delta} (\Sigma, \mathsf{error})}\text{EVAL\_STMT\_ASSIGN\_PTR\_LPTR\_ERR}$$

$$\frac{\begin{array}{l}\Sigma(\ell_1) = \mathsf{lptr}\,(pv_1) \qquad \Sigma(\ell_2) = \mathsf{lptr}\,(\mathsf{bad\_ptr}) \\ \Sigma' = \Sigma\,[\ell_1 \mapsto \mathsf{lptr}\,(\mathsf{bad\_ptr})]\end{array}}{(\Sigma, \ell_1 = \ell_2) \underset{\mathsf{stmt}}{\longrightarrow}^{n}_{\Delta} (\Sigma', \mathsf{skip})}\text{EVAL\_STMT\_ASSIGN\_LPTR\_LPTR\_NULL}$$

$$\frac{\begin{array}{l}\Sigma({x_1}^{n_1}@\pi_1) = \mathsf{lptr}\,(pv_1) \\ \Sigma(\ell_2) = \mathsf{lptr}\,({x_2}^{n_2}@\pi_2) \\ \Sigma' = \Sigma\,[{x_1}^{n_1}@\pi_1 \mapsto \mathsf{lptr}\,({x_2}^{n_2}@\pi_2)] \qquad n_2 \leq n_1\end{array}}{(\Sigma, {x_1}^{n_1}@\pi_1 = \ell_2) \underset{\mathsf{stmt}}{\longrightarrow}^{n}_{\Delta} (\Sigma', \mathsf{skip})}\text{EVAL\_STMT\_ASSIGN\_LPTR\_LPTR}$$

$$\frac{\begin{array}{l}\Sigma({x_1}^{n_1}@\pi_1) = \mathsf{lptr}\,(pv_1) \qquad n_2 \not\leq n_1 \\ \Sigma(\ell_2) = \mathsf{lptr}\,({x_2}^{n_2}@\pi_2)\end{array}}{(\Sigma, {x_1}^{n_1}@\pi_1 = \ell_2) \underset{\mathsf{stmt}}{\longrightarrow}^{n}_{\Delta} (\Sigma, \mathsf{error})}\text{EVAL\_STMT\_ASSIGN\_LPTR\_LPTR\_ERR}$$

$$\frac{}{(\Sigma, \mathsf{skip}; s) \underset{\mathsf{stmt}}{\longrightarrow}^n_\Delta (\Sigma, s)}\text{EVAL\_STMT\_SEQ\_SKIP}$$

$$\frac{(\Sigma, e_1) \underset{\mathsf{exp}}{\longrightarrow}^n_\Delta (\Sigma', e_1')}{(\Sigma, e_1 = e_2) \underset{\mathsf{stmt}}{\longrightarrow}^n_\Delta (\Sigma', e_1' = e_2)}\text{EVAL\_STMT\_ASSIGN\_CONG1}$$

$$\frac{(\Sigma, e_2) \underset{\mathsf{exp}}{\longrightarrow}^n_\Delta (\Sigma', e_2')}{(\Sigma, \ell_1 = e_2) \underset{\mathsf{stmt}}{\longrightarrow}^n_\Delta (\Sigma', \ell_1 = e_2')}\text{EVAL\_STMT\_ASSIGN\_CONG2}$$

$$\frac{}{(\Sigma, \mathsf{error} = e_2) \underset{\mathsf{stmt}}{\longrightarrow}^n_\Delta (\Sigma, \mathsf{error})}\text{EVAL\_STMT\_ASSIGN\_ERR1}$$

$$\frac{}{(\Sigma, \ell_1 = \mathsf{error}) \underset{\mathsf{stmt}}{\longrightarrow}^n_\Delta (\Sigma, \mathsf{error})}\text{EVAL\_STMT\_ASSIGN\_ERR2}$$

$$\frac{(\Sigma, s_1) \underset{\mathsf{stmt}}{\longrightarrow}^n_\Delta (\Sigma', s_1')}{(\Sigma, s_1; s_2) \underset{\mathsf{stmt}}{\longrightarrow}^n_\Delta (\Sigma', s_1'; s_2)}\text{EVAL\_STMT\_SEQ\_CONG}$$

$$\frac{}{(\Sigma, \mathsf{error}; s_2) \underset{\mathsf{stmt}}{\longrightarrow}^n_\Delta (\Sigma, \mathsf{error})}\text{EVAL\_STMT\_SEQ\_ERROR}$$

$\boxed{(\Sigma, e) \underset{\mathsf{exp}}{\longrightarrow}^n_\Delta (\Sigma', e')}$ Expression evaluation

$$\frac{}{(\Sigma, x) \underset{\mathsf{exp}}{\longrightarrow}^n_\Delta (\Sigma, x^n@\,)}\text{EVAL\_EXP\_VAR}$$

$$\frac{\begin{array}{l} x^n \textbf{ fresh for } \Sigma \\ \Sigma' = \Sigma\,[x^n@ \mapsto \mathsf{ptr}\,(\mathsf{bad\_ptr})] \end{array}}{(\Sigma, \mathsf{null}) \underset{\mathsf{exp}}{\longrightarrow}^n_\Delta (\Sigma', x^n@\,)}\text{EVAL\_EXP\_NULL}$$

$$\frac{}{(\Sigma, (x^{n'}@\pi).x') \xrightarrow[\mathsf{exp}]{}{}^{n}_{\Delta} (\Sigma, x^{n'}@\pi +\!\!+ x')}\text{EVAL\_EXP\_FLD}$$

$$\frac{\begin{array}{l}\Sigma(\ell_1) = C \\ \mathsf{struct}\ C\ \{\ fldsmeths\}; \in\ \Delta \\ \tau_1\ f(\tau_2\ x)\{\overline{\tau_i\ x_i}^{\,i}; s; \mathsf{return}\ e\}\ \in\ meths \\ \Sigma_2 = \mathsf{copy\_store}(\Sigma, {x_2}^{n_2}@\pi_2, x^{n+1}) \\ \Sigma_3 = [{this}^{n+1}@\ \mapsto \mathsf{lptr}\ (\ell_1)] \\ \overline{\Sigma_{4\,i} = \mathsf{default}_\Delta\langle\tau_i\rangle({x_i}^{n+1}@\,)}^{\,i} \\ \Sigma' = \Sigma\,\Sigma_2\,\Sigma_3\,\overline{\Sigma_{4\,i}}^{\,i}\end{array}}{(\Sigma, \ell_1.f({x_2}^{n_2}@\pi_2)) \xrightarrow[\mathsf{exp}]{}{}^{n}_{\Delta} (\Sigma', \{s;\ \mathsf{return}\ e\})}\text{EVAL\_EXP\_METH}$$

$$\frac{\begin{array}{l}x^n\ \textbf{fresh for}\ \Sigma \\ \Sigma_2 = \mathsf{copy\_store}(\Sigma, \ell, x^n) \\ \Sigma' = \Sigma\,\Sigma_2 \backslash (n+1)\end{array}}{(\Sigma, \{\mathsf{skip};\ \mathsf{return}\ \ell\}) \xrightarrow[\mathsf{exp}]{}{}^{n}_{\Delta} (\Sigma', x^n@\,)}\text{EVAL\_EXP\_BODY\_RET}$$

$$\frac{\begin{array}{l}x^n, y^0\ \textbf{fresh for}\ \Sigma \\ \Sigma_1 = \mathsf{default}_\Delta\langle C\rangle(y^0@\,) \qquad\qquad \Sigma' = \Sigma\,\Sigma_1\,[x^n@\ \mapsto \mathsf{ptr}\ (y^0@\,)]\end{array}}{(\Sigma, \mathsf{new}\ C()) \xrightarrow[\mathsf{exp}]{}{}^{n}_{\Delta} (\Sigma', x^n@\,)}\text{EVAL\_EXP\_NEW}$$

$$\frac{\begin{array}{l}y^n\ \textbf{fresh for}\ \Sigma \\ \Sigma' = \Sigma\,[y^n@\ \mapsto \mathsf{lptr}\ (x^{n'}@\pi)]\end{array}}{(\Sigma, \&x^{n'}@\pi) \xrightarrow[\mathsf{exp}]{}{}^{n}_{\Delta} (\Sigma', y^n@\,)}\text{EVAL\_EXP\_ADDR}$$

$$\frac{\Sigma(\ell) = \mathsf{ptr}\ (x^{n'}@\pi)}{(\Sigma, *\ell) \xrightarrow[\mathsf{exp}]{}{}^{n}_{\Delta} (\Sigma, x^{n'}@\pi)}\text{EVAL\_EXP\_DEREF\_PTR}$$

$$\frac{\Sigma(\ell) = \mathsf{lptr}\ (x^{n'}@\pi)}{(\Sigma, *\ell) \xrightarrow[\mathsf{exp}]{}{}^{n}_{\Delta} (\Sigma, x^{n'}@\pi)}\text{EVAL\_EXP\_DEREF\_LPTR}$$

$$\frac{\Sigma(\ell) = \mathsf{ptr}\ (\mathsf{bad\_ptr})}{(\Sigma, *\ell) \xrightarrow[\mathsf{exp}]{}{}^{n}_{\Delta} (\Sigma, \mathsf{error})}\text{EVAL\_EXP\_DEREF\_PTR\_NULL}$$

$$\frac{\Sigma(\ell) = \mathsf{lptr}\,(\mathsf{bad\_ptr})}{(\Sigma, *\ell) \xrightarrow[\mathsf{exp}]{}{}^{n}_{\Delta}(\Sigma, \mathsf{error})}\text{EVAL\_EXP\_DEREF\_LPTR\_NULL}$$

$$\frac{(\Sigma, e) \xrightarrow[\mathsf{exp}]{}{}^{n}_{\Delta}(\Sigma', e')}{(\Sigma, e.x) \xrightarrow[\mathsf{exp}]{}{}^{n}_{\Delta}(\Sigma', e'.x)}\text{EVAL\_EXP\_FLD\_CONG}$$

$$\frac{}{(\Sigma, \mathsf{error}.x) \xrightarrow[\mathsf{exp}]{}{}^{n}_{\Delta}(\Sigma, \mathsf{error})}\text{EVAL\_EXP\_FLD\_ERR}$$

$$\frac{(\Sigma, e_1) \xrightarrow[\mathsf{exp}]{}{}^{n}_{\Delta}(\Sigma', e_1')}{(\Sigma, e_1.f(e_2)) \xrightarrow[\mathsf{exp}]{}{}^{n}_{\Delta}(\Sigma', e_1'.f(e_2))}\text{EVAL\_EXP\_METH\_CONG1}$$

$$\frac{(\Sigma, e_2) \xrightarrow[\mathsf{exp}]{}{}^{n}_{\Delta}(\Sigma', e_2')}{(\Sigma, \ell_1.f(e_2)) \xrightarrow[\mathsf{exp}]{}{}^{n}_{\Delta}(\Sigma', \ell_1.f(e_2'))}\text{EVAL\_EXP\_METH\_CONG2}$$

$$\frac{}{(\Sigma, \mathsf{error}.f(e_2)) \xrightarrow[\mathsf{exp}]{}{}^{n}_{\Delta}(\Sigma, \mathsf{error})}\text{EVAL\_EXP\_METH\_ERR1}$$

$$\frac{}{(\Sigma, \ell_1.f(\mathsf{error})) \xrightarrow[\mathsf{exp}]{}{}^{n}_{\Delta}(\Sigma, \mathsf{error})}\text{EVAL\_EXP\_METH\_ERR2}$$

$$\frac{(\Sigma, s) \xrightarrow[\mathsf{stmt}]{}{}^{n+1}_{\Delta}(\Sigma', s')}{(\Sigma, \{s;\ \mathsf{return}\ e\}) \xrightarrow[\mathsf{exp}]{}{}^{n}_{\Delta}(\Sigma', \{s';\ \mathsf{return}\ e\})}\text{EVAL\_EXP\_BODY\_CONG1}$$

$$\frac{(\Sigma, e) \xrightarrow[\mathsf{exp}]{}{}^{n+1}_{\Delta}(\Sigma', e')}{(\Sigma, \{\mathsf{skip};\ \mathsf{return}\ e\}) \xrightarrow[\mathsf{exp}]{}{}^{n}_{\Delta}(\Sigma', \{\mathsf{skip};\ \mathsf{return}\ e'\})}\text{EVAL\_EXP\_BODY\_CONG2}$$

$$\frac{}{(\Sigma, \{\mathsf{error};\ \mathsf{return}\ e\}) \xrightarrow[\mathsf{exp}]{}{}^{n}_{\Delta}(\Sigma, \mathsf{error})}\text{EVAL\_EXP\_BODY\_ERR1}$$

$$\frac{}{(\Sigma, \{\mathsf{skip};\ \mathsf{return\ error}\}) \xrightarrow[\mathsf{exp}]{}{}^{n}_{\Delta} (\Sigma, \mathsf{error})}\ \text{EVAL\_EXP\_BODY\_ERR2}$$

$$\frac{(\Sigma, e) \xrightarrow[\mathsf{exp}]{}{}^{n}_{\Delta} (\Sigma', e')}{(\Sigma, \&e) \xrightarrow[\mathsf{exp}]{}{}^{n}_{\Delta} (\Sigma', \&e')}\ \text{EVAL\_EXP\_ADDR\_CONG}$$

$$\frac{}{(\Sigma, \&\mathsf{error}) \xrightarrow[\mathsf{exp}]{}{}^{n}_{\Delta} (\Sigma, \mathsf{error})}\ \text{EVAL\_EXP\_ADDR\_ERROR}$$

$$\frac{(\Sigma, e) \xrightarrow[\mathsf{exp}]{}{}^{n}_{\Delta} (\Sigma', e')}{(\Sigma, *e) \xrightarrow[\mathsf{exp}]{}{}^{n}_{\Delta} (\Sigma', *e')}\ \text{EVAL\_EXP\_DEREF\_CONG}$$

$$\frac{}{(\Sigma, *\mathsf{error}) \xrightarrow[\mathsf{exp}]{}{}^{n}_{\Delta} (\Sigma, \mathsf{error})}\ \text{EVAL\_EXP\_DEREF\_ERROR}$$