Oxide: The Essence of Rust

AARON WEISS, Northeastern University, USA
OLEK GIERCZAK, Northeastern University, USA
DANIEL PATTERSON, Northeastern University, USA
NICHOLAS D. MATSAKIS, Mozilla Research, USA
AMAL AHMED, Northeastern University, USA

Rust claims to advance industrial programming by bridging the gap between *low-level* systems programming and *high-level* application programming. At the heart of the argument that this enables programmers to build more reliable and efficient software is the *borrow checker* — a novel approach to *ownership* that aims to balance type system expressivity with usability. And yet, to date there is no core type system that captures Rust's notion of ownership and borrowing, and hence no foundation for research on Rust to build upon.

In this work, we set out to capture the essence of this model of ownership by developing a type systems account of Rust's borrow checker. We present Oxide, a formalized programming language close to *source-level* Rust (but with fully-annotated types). This presentation takes a new view of *lifetimes* as an approximation of the *provenances* of references, and our type system is able to automatically compute this information through a substructural typing judgment. We provide the first syntactic proof of type safety for borrow checking using progress and preservation. Oxide is a simpler formulation of borrow checking — including recent features such as *non-lexical lifetimes* — that we hope researchers will be able to use as the basis for work on Rust.

1 INTRODUCTION

The Rust programming language exists at the intersection of low-level "systems" programming and high-level "applications" programming, aiming to empower the programmer with both fine-grained control over memory and performance *and* high-level abstractions that make software more reliable and quicker to produce. To accomplish this, Rust integrates decades of programming-languages research into a production system. Most notably, this includes ideas from linear and ownership types [Clarke et al. 1998; Girard 1987; Lafont 1988; Noble et al. 1998] and region-based memory management [Fluet et al. 2006; Grossman et al. 2002]. Yet, Rust goes beyond prior art in developing a particular discipline that aims to balance both *expressivity* and *usability*. As such, Rust has something interesting to teach us about making ownership *practical* for programming.

But without a platform to build upon, it is difficult for researchers to learn, understand, and investigate this new discipline. This is not a new problem though; the novelty of new languages has often encouraged their formal study to learn *precisely* what they offer. Featherweight Java [Igarashi et al. 2001] did just this — illuminating the language being studied *and* providing a foundation for future research. This has inspired our own effort, and so we endeavor in this work to distill *the essence of Rust* through our formalization, Oxide.

While there are some existing formalizations of Rust [Benitez 2016; Jung et al. 2018; Reed 2015], none capture a high-level understanding of Rust's essence (namely *ownership* and *borrowing*). The first major effort, *Patina* [Reed 2015], formalized an early version of Rust predating much of the work to simplify and streamline the language, and was unfinished. The next effort, Rusty Types [Benitez 2016], developed a formal calculus, *Metal*, which uses an algorithmic borrow checker that is less expressive than both Rust and Oxide. The most complete effort to date is RustBelt [Jung et al. 2018] which defines λ_{Rust} and takes a semantic approach to type soundness [Ahmed 2004; Ahmed et al. 2010; Milner 1978] to verify that major parts of Rust's standard library (written using `unsafe` code)

Authors' addresses: Aaron Weiss, Northeastern University, Boston, MA, 02115, USA, weiss@ccs.neu.edu; Olek Gierczak, Northeastern University, Boston, MA, 02115, USA, gierczak.o@northeastern.edu; Daniel Patterson, Northeastern University, Boston, MA, 02115, USA, dbp@dbpmail.net; Nicholas D. Matsakis, , Mozilla Research, Boston, MA, 02108, USA, nmatsakis@mozilla.com; Amal Ahmed, Northeastern University, Boston, MA, 02115, USA, amal@ccs.neu.edu.

do not violate its safety guarantees. Yet, for our purposes, λ_{Rust}'s continuation-passing style and low-level nature — closer to Rust's Mid-level Intermediate Representation (MIR) — make it difficult to use for *source-level* reasoning. Follow-on work by Jung et al. [2019] provides an operational model called *Stacked Borrows* for memory accesses that is orthogonal to our efforts.

As we will see in the rest of the paper, Oxide is a much higher-level language. Its syntax bears a close resemblance to that of Rust, and its semantics deals with an *abstract* notion of memory that does not require us to pick a specific memory layout for each type. This is significant since Rust as a language lacks a formal specification, and there are still ongoing discussions about memory layout and validity guarantees in Rust's unsafe code guidelines workgroup [2019]. Yet, Oxide also takes steps to make the semantics simpler and easier to follow. In particular, we require the types of bindings to be fully annotated in Oxide programs to avoid the orthogonal complexities of type inference. Since we are interested in *ownership*, we focus on the *safe* portion of Rust without standard library abstractions implemented using `unsafe` code. In Section 6.1, we discuss extensions to Oxide that address this, including a sketch of heap allocation support with Rust's `Vec` type.

Our efforts to develop Oxide have led us to three main contributions. First, we present Oxide as the first formal account close to source-level Rust. Second, we provide the first syntactic type safety [Wright and Felleisen 1992] result for Rust. Lastly, and most significantly, we note that while Rust's borrow-checking implementation relies on constraint generation and an algorithmic constraint solver, we provide the first inductive definition of borrow checking. Our borrow-checking definition builds on the view of lifetimes as approximations of the provenances of references, as opposed to an abstraction of the lines of code where the referenced memory is live. Oxide is a tested semantics in that we validate its faithfulness to RUSTC borrow checking on the subset of features supported by Oxide using tests from Rust's official borrow checker and non-lexical lifetimes test suites. Ultimately, Oxide has allowed us to develop a more explainable *essence* of Rust.

The rest of the paper is organized as follows: §2 describes the essence of Rust and Oxide at an intuitive level. §3 presents the formal details of Oxide including the syntax (§3.1), type system (§3.2), operational semantics (§3.3), and metatheory (§3.4). §4 provides examples and evidence that Oxide faithfully models Rust, including discussion (§4.3) of our REDUCER from Rust to Oxide and a type checker OXIDETC used to validate that Oxide typechecking matches Rust on a subset of Rust's official test suite. We discuss related work in §5 and avenues for future work built on Oxide in §6.

The technical appendices include complete definitions (§A, §B, §C, §D), typing rules (§B.4), and proofs (§E). Our implementation and test suite for our tested semantics are available on GitHub.

2 DATA THEY CAN CALL THEIR OWN

> Nothing is yours. It is to use. It is to share.
> If you will not share it, you cannot use it.
>
> *The Dispossessed*
> URSULA K. LE GUIN

The essence of Rust lies in its novel approach to *ownership* and *borrowing*, which account for the most interesting parts of the language's static semantics and the justification for its claims to *memory safety* and *data race freedom*. In this section, we explore ownership and borrowing intuitively and how they are captured in Oxide.

2.1 Ownership

Rust's notion of ownership rests atop a long lineage of work, harkening back to the early days of linear logic [Girard 1987], and especially efforts by Wadler [1991] and Baker [1992] to develop systems for functional programming *without* garbage collection. However, as noted by Wakeling and Runciman [1991], Wadler's effort relied greatly on pervasive copying. This reliance on copying

and the associated performance penalty would not suffice for real world systems programming efforts, and thus, Rust's ownership model is best understood as instead building off of Baker's work on Linear Lisp where linearity enabled efficient reuse of objects in memory [Baker 1992, 1994a,b, 1995]. The resemblance is especially strong between Rust *without* borrowing and Baker's 'use-once' variables [Baker 1995]. We illustrate these ideas at work in Rust with the following example:

```
1  struct Point(u32, u32);
2  let mut pt = Point(6, 9);
3  let mut x = pt;
4  let mut y = pt; // ERROR: pt was already moved
```

In this example, we first declare a type `Point` that consists of a pair of unsigned 32-bit integers (`u32`). Then, on line 2, we create a new `Point` bound to `pt`. Here, `mut` means that the binding for `pt` can be reassigned. We say that this new value is *owned* by this identifier `pt`. Then, on line 3, we transfer this ownership by *moving* the value from `pt` to `x`. After moving the value out of `pt`, we invalidate this old name. Subsequently, when we attempt to use it again on line 4, we encounter an error because `pt` was already moved in the previous line. With the exception of required type annotations, this program is identical in Oxide, and similarly produces an error.

2.2 Borrowing

Rust's main departure from techniques like 'use-once' variables [Baker 1995] is a softening of a rather stringent requirement: namely, that *everything* must be managed uniquely. Instead, Rust allows the programmer to locally make a decision to use unique references [Minsky 1996] with unguarded mutation *or* to use shared references without such mutation.[1] This flexibility in choosing arises at the point where the programmer creates a new reference, and draws inspiration from work on ownership types and flexible alias protection [Clarke et al. 1998; Noble et al. 1998]. We again illustrate its use in Rust with an example:

```
1  struct Point(u32, u32);
2  let mut pt = Point(6, 9);
3  let x = &pt;
4  let y = &pt; // no error, sharing is okay!
```

In the above example, we replaced the *move* expressions on lines 3 and 4 with *borrow* expressions that each create a shared reference to `pt`. As noted in the comment, this program no longer produces an error because the references allow precisely this kind of sharing. but one should note that this sharing would be *disallowed* by a standard linear or affine type system. However, unlike with just plain variable bindings (as in the previous example), we are unable to mutate through these references, and attempts to do so would result in a compile-time error. Next, we will replace our shared references with unique, `mut`-able ones instead:

```
1  struct Point(u32, u32);
2  let mut pt = Point(6, 9);
3  let x = &mut pt;
4  let y = &mut pt; // ERROR: cannot borrow pt as mutable twice
5  ... // additional code that uses x
```

In the example above, we have now chosen to create unique, rather than shared, references to `pt`. However, since our program attempts to do so twice, we encounter an error similar to the one we had in the first place — when we tried to move `pt` twice. The astute reader might notice that

[1] The use of "such" here is rather intentional as dynamically guarded mutation, e.g. using a `Mutex`, is still allowed through a shared reference. Indeed, this is precisely what makes such guards *useful* when programming.

another important change happened — we added some additional code afterward that somehow makes use of `x`. This is important because of a feature in Rust known as *non-lexical lifetimes* (or NLL for short) [Matsakis 2016a; Turon et al. 2017]. With non-lexical lifetimes and no uses of `x` in the ensuing code, the compiler would figure out that the uniqueness of unique references would not *really* be violated since `x` is never used, and thus the program is able to pass the borrow checker.

Similar to the last example, the borrow checker also prevents us from mixing **`mut`**-able references (which ought to be unique) with shared references, as in the following example:

```
1  struct Point(u32, u32);
2  let mut pt: Point = Point(6, 9);
3  let x: &'a mut Point = &mut pt;
4  let y: &'b Point = &pt;
5  //                 ^~~
6  // ERROR: cannot borrow pt while a mutable loan is live
7  ... // additional code that uses x and y
```

In this case, we've changed the borrow expression on line 4 to create a shared, rather than unique, reference. We've also chosen to add explicit type annotations to our bindings on lines 2–4. This again produces an error because Rust forbids the creation of a shared reference while a mutable *loan* exists. Here, we use the word loan to refer to the state introduced in the borrow checker (which records that the loan's uniqueness and its origin) by the creation of a reference. Regions[2] in Rust (denoted `'a`, `'b`, etc.) can be understood as collections of these loans which together statically approximate which pointers could be used dynamically at a particular reference type. This is the sense in which Rust's regions are distinct from the existing literature on region-based memory management [Fluet et al. 2006; Grossman et al. 2002; Tofte and Talpin 1994, 1997].

While we were unable to create a second reference to the same place as an existing unique reference in our past examples, Rust allows the programmer to create two unique references to disjoint paths within the same object, as in the following example:

```
1  struct Point(u32, u32);
2  let mut pt: Point = Point(6, 9);
3  let x: &'a mut u32 = &mut pt.0;
4  let y: &'b mut u32 = &mut pt.1;
5  // no error, our loans don't overlap!
```

In this example, we're borrowing from specific paths within `pt` (namely, the first and second projections respectively). Since these paths give a name to the places being referenced, we refer to them as *places*. Here, we see Rust employs a fine-grained notion of ownership that allows unique loans against non-overlapping places within *aggregate structures* (like structs, enums, and tuples). Intuitively, this is safe because the parts of memory referred to by each place (in this case, `pt.0` and `pt.1`) do not overlap, and thus they represent portions that can each be uniquely owned.

Rust allows supports an additional pattern that weakens conventional notions of flexible alias protection. In particular, Rust allows the programmer to create a unique reference by borrowing from one they already have. However, the programmer is unable to use the old reference until the *reborrowed* one ends. We can see this *reborrowing* at work in the following example:

```
1  struct Point(u32, u32);
2  let mut pt: Point = Point(6, 9);
3  let x: &'a mut u32 = &mut pt.0;
```

[2]Historically, Rust has used the term *lifetime*, rather than region, but recent efforts on a borrow checker rewrite called Polonius have transitioned to using the term region [Matsakis 2018]. We discuss Polonius further in §4.4.

```
4  let y: &'b mut u32 = &mut *x;
5  // can use y, cannot use x until we drop y
```

In this example, we borrow the first projection of `pt` (`pt.0`) and then reborrow it by creating a borrow to `*x`. We then can use `y` in the continuation, but won't be able to use `x` until `y` is dropped.

2.3 Formalizing Rust

Notably, in Oxide, these programs are largely unchanged. The main differences from Rust are threefold. First, we annotate the type of every binding, including adding bindings and annotations for local lifetime variables. Second, acknowledging that **`mut`** plays two distinct roles in Rust, we focus on its essential use (as a qualifier for the uniqueness of a reference), and removed the syntactic restriction on reassigning a binding. That is, while Rust allows the programmer to mark let bindings as **`mut`** to enable the bound variable to be reassigned, we omit this annotation, and allow all bindings to be mutated when it is safe to do so. Finally, we shift the conceptual terminology we use to discuss regions/lifetimes. In particular, as regions here approximate the origin of references, we choose a more precise term, *approximate provenances*, and refer to their variable form (`'a`, `'b`, etc.) as *provenance variables*. Translating our last example into Oxide gives us the following:

```
1  struct Point(u32, u32);
2  letprov<'x, 'y> {
3    let pt: Point = Point(6, 9);
4    let x: &'x uniq u32 = &'x uniq pt.0;
5    let y: &'y uniq u32 = &'y uniq pt.1;
6  }
7  // no error, our loans don't overlap
```

Here, our type annotations on lines 3–5 (i.e. for each let binding) are now required, and we replaced **`mut`** and the lack of an annotation with two qualifiers `uniq` and `shrd` respectively. We also annotate the references with the local provenance `'x` and `'y` which themselves are introduced by the new `letprov` binding on line 2. The remainder of the program is otherwise unchanged. As in the Rust version, the Oxide version type checks without error because the origins of the loans are disjoint. That is, `x` can only have originated from `pt.0` and `y` only from `pt.1`.

During type-checking, Oxide will track sets of loans for the provenance variables bound with `letprov` (i.e., `'x` and `'y`). Specifically, after line 4, `'x` will be mapped to the loan set $\{\ {}^{\texttt{uniq}}\texttt{pt.0}\ \}$ and, after line 5, `'y` will be mapped to $\{\ {}^{\texttt{uniq}}\texttt{pt.1}\ \}$. Moreover, when type-checking each borrow expression, Oxide looks at the existing loans in its environment to determine that all of the live loans (e.g., between line 4 and 5, the live loans are just the loans that `'x` maps to) are disjoint from the place being borrowed (which on line 5 is `pt.1`). At runtime, the program evaluates with a stack σ that satisfies an environment Γ. That is, σ maps variables to values whose types are given in Γ.

Information Loss. Though all the examples we've discussed thus far have a precise origin for every reference, provenances are, in general, approximate due to join points in the program. For example, in an if expression, we might create some new set of loans in one branch, and a different set of loans in the other branch. To ensure that the system is sound, we need to be conservative and act as if *both* sets of loans are live. As such, we combine the environments from each side of the branch. We will come back to this with a more formal treatment in §3.2.

2.4 Oxide, More Formally

We've now seen enough to start to describe Oxide in more formal detail. First, we note that since information about loans must flow between expressions, we must somehow be able to track this flow in our type system. To do so, we use a typing judgment in an environment-passing style. The

$$\textsc{T-Move}\quad \frac{\Delta;\ \Gamma \vdash_{\texttt{uniq}} \pi \Rightarrow \{\ {}^{\texttt{uniq}}\pi\ \} \qquad \Gamma(\pi) = \tau^{\text{SI}} \qquad \text{noncopyable}_{\Sigma}\ \tau^{\text{SI}}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\pi} : \tau^{\text{SI}} \Rightarrow \Gamma[\pi \mapsto \tau^{\text{SI}^{\dagger}}]}$$

$$\textsc{T-Borrow}\quad \frac{\Gamma(r) = \emptyset \qquad \Delta;\ \Gamma \vdash_{\omega} p \Rightarrow \{\ \overline{\ell}\ \} \qquad \Delta;\ \Gamma \vdash_{\omega} p : \tau^{\text{XI}}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\&r\ \omega\ p} : \&r\ \omega\ \tau^{\text{XI}} \Rightarrow \Gamma[r \mapsto \{\ \overline{\ell}\ \}]}$$

Fig. 1. The essence of Oxide.

shape of our judgment is $\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e} : \tau \Rightarrow \Gamma'$, where Σ is the global environment denoting the top-level function definitions of the program, Δ is the type environment tracking in-scope type and provenance variables and their kinds, and Γ is the stack typing mapping both variables to their types and provenances to their associated loan sets. The output environment Γ' denotes the stack typing to use when type checking expressions *after* this one. This is essential in capturing the substructural aspects of Oxide as e may "consume resources", changing ownership/borrowing state from Γ to Γ'.

Places and Place Expressions. Before we can look at some typing rules, there is one remaining piece to understand: the distinction between places π and place expressions p. While places give a name to a precise part of memory (e.g., `pt` or `pt.0`, which gives a path through a struct), place expressions also include dereferences of places (e.g., `*x` or `(*x).1`. This dereferencing is the source of the gap in their meaning — since a reference's provenance can only be statically known *approximately*, a place expression during type-checking has to be thought of as representing potentially many places. We'll see later that places alone aren't enough to describe all abstract addresses in Oxide that arise at runtime — we will need to generalize places to a richer address abstraction that we call *referents* (§3.3) which subsume places. Each reference will be represented as a pointer to a specific referent, and so place expressions p at runtime will always be directly evaluated to a single referent $\mathcal{R}$.

Borrow Checking. In Figure 1, we see two typing rules that capture the essence of how Oxide models Rust's ownership semantics, but to understand them, we'll need to understand the crucial *ownership safety* judgment in their premises: $\Delta;\ \Gamma \vdash_{\omega} p \Rightarrow \{\ \overline{{}^{\omega}p}\ \}$. We can read it as saying "in the environments Δ and Γ, it is safe to use the place expression p ω-ly," where ω is either `uniq` or `shrd`. That is, if we have a derivation where ω is `uniq`, we know that we can use the place expression p uniquely because we have a proof that there are no live loans against the section(s) of memory that p represents. T-Move checks if π is `uniq`-ly safe because we know that it is only safe to move a value out of the environment when there are no aliases to it.

Further, when we have a derivation where ω is `shrd`, we know that we can use the place expression p sharedly because we have a proof that there are no live *unique* loans against the section(s) of memory that p represents. In the case of borrowing (as in T-Borrow), these two meanings correspond exactly to the intuition behind when an ω borrow is safe, and the loan-set output from the derivation (called a *borrow chain*) tells us what loans this use of p will create.

Since it is precisely this ownership safety judgment that captures the essence of Rust's ownership semantics, we understand Rust's borrow checking system as ultimately being a system for statically building a proof that data in memory is either *uniquely owned* (and thus able to allow unguarded mutation) or *collectively shared*, but not both. To do so, intuitively, ownership safety looks at all of the provenances found within Γ, and ensures that all the loans they contain are not in conflict with the place expression p in question. For a `uniq` borrow, a conflict occurs if *any* loan maps to an overlapping place, but for a `shrd` borrow, a conflict occurs only when a `uniq` loan maps to an overlapping place. Now, this intuition provides only a partial picture since ownership safety must do more in order to support reborrowing; we return to these details in §3.2.

2.5 Non-Lexical Lifetimes in Oxide

In Oxide, we allow non-lexical lifetimes through a restricted form of *weakening* provided by the rule T-Drop. To illustrate how T-Drop in combination with our environment-passing typing judgment enables non-lexical lifetimes, let us consider a variant of an earlier example where we produce two unique pointers, but our first unique pointer is instead *not* used in the remainder of the program:

```
1  struct Point(u32, u32);
2  letprov<'x, 'y> {
3    let pt: Point = Point(6, 9);
4    let x: &'x uniq Point = &'x uniq pt;
5    let y: &'y uniq Point = &'y uniq pt;
6    ... // additional code using y, but not x
7  }
```

$$\textsc{T-Drop}\quad \frac{\Gamma(\pi) = \tau_\pi^{\mathrm{SI}} \qquad \Sigma;\ \Delta;\ \Gamma[\pi \mapsto \tau_\pi^{\mathrm{SI}^\dagger}] \vdash \boxed{e} : \tau^{\mathrm{SX}} \Rightarrow \Gamma_f}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e} : \tau^{\mathrm{SX}} \Rightarrow \Gamma_f}$$

Normally, when we attempt to type check the expression being bound to `y` with `x` still bound, we are unable to type check the expression for `y` since it would produce a second ostensibly "unique" reference to `pt`. However, we can drop `x` from our context by marking it *dead*, ending all of the loans in `'x`, and allowing us to successfully proceed with the rest of the program. This corresponds to a bottom-up reading of our T-Drop rule, shown to the right of the example. The notation $\Gamma[\pi \mapsto \tau_\pi^{\mathrm{SI}^\dagger}]$ in the premise captures marking the type of π as dead After applying this rule, if `x` *was* used in the rest of the program, we would encounter a new point where we cannot make a typing derivation since our typing rules do not allow us to use variables at dead types. However, since `x` is not used, the rest of the program will succeed since it necessarily did not depend on `x`'s existence. Intuitively, this rule says references that are no longer used may as well not exist.

In the rest of the paper, we explore the formalism in more detail along with the possibilities and consequences of this new model for Rust.

3 OXIDE

We now present Oxide formally. The remaining formal details are present in the appendices, including the syntax (§A), statics (§B), including all typing rules (§B.4), well-formedness rules (§B.1) and additional judgments (§B.5), metafunctions (§C), and the complete operational semantics (§D).

3.1 Syntax

Figure 2 presents most of the syntax of Oxide. In Oxide, we annotate references with ownership qualifiers ω, indicating whether the reference is shared (`shrd`) or unique (`uniq`). We use these rather than their equivalents in Rust (no annotation and **`mut`** respectively) because the terms more accurately reflect the semantic focus on *aliasing*, rather than *mutation*. Indeed, in Rust, a value of the type `&&mut u32` *cannot* be mutated (because we have a shared reference to a unique reference), and a value of the type `&Cell<u32>`[3] *can* be mutated through the method `Cell::set`. In this sense, the official name **`mut`** in Rust should be thought of as an *accident of history*, rather than something that appropriately reflects intuitions about how the language works.

Places and Place Expressions. As discussed at a high-level in §2.2, place expressions p and places π are names for paths from a particular variable to a particular part of the object stored there, whether that be a field of a struct, or a projection of a tuple. One might think of place expressions as a sort of syntactic generalization of variables. They are analogous to what are called `lvalues` in C. Place expression contexts $p^\square$ are used in various parts of the formalism to decompose place expressions p into an innermost dereferenced place, $*\pi$, and an outer context $p^\square$.

[3] `Cell<T>` is a Rust standard library type that provides a "mutable memory location" that allows mutation in its API.

Variables	x	Functions	f	Type Vars.	α	Frame Vars.	φ
Concrete Prov.	r	Abstract Prov.	ϱ	Strings	$\texttt{str}$	Naturals	m, n, k

Path	q	::=	$\epsilon \mid n.q$
Places	π	::=	$x.q$
Place Expressions	p	::=	$x \mid *p \mid p.n$
Place Expression Contexts	$p^{\square}$	::=	$\square \mid *p^{\square} \mid p^{\square}.n$
Provenances	ρ	::=	$\varrho \mid r$
Ownership Qualifiers	ω	::=	$\texttt{shrd} \mid \texttt{uniq}$
Loans	ℓ	::=	${}^{\omega}p$
Kinds	κ	::=	$\star \mid \texttt{PRV} \mid \texttt{FRM}$
Base Types	τ^{B}	::=	$\texttt{bool} \mid \texttt{u32} \mid \texttt{unit}$
Sized Types	τ^{SI}	::=	$\tau^{\text{B}} \mid \alpha \mid \&\rho\ \omega\ \tau^{\text{XI}} \mid [\tau^{\text{SI}}; n] \mid (\tau_1^{\text{SI}}, \ldots, \tau_n^{\text{SI}})$
		$\mid$	$\forall <\overline{\varphi}, \overline{\varrho}, \overline{\alpha}> (\tau_1^{\text{SI}}, \ldots, \tau_n^{\text{SI}}) \xrightarrow{\Phi} \tau_r^{\text{SI}}\ \texttt{where}\ \overline{\varrho_1 : \varrho_2}$
Maybe Unsized Types	τ^{XI}	::=	$\tau^{\text{SI}} \mid [\tau^{\text{SI}}]$
Dead Types	τ^{SD}	::=	$\tau^{\text{SI}^{\dagger}} \mid (\tau_1^{\text{SD}}, \ldots, \tau_n^{\text{SD}})$
Maybe Dead Types	τ^{SX}	::=	$\tau^{\text{SI}} \mid \tau^{\text{SD}} \mid (\tau_1^{\text{SX}}, \ldots, \tau_n^{\text{SX}})$
Types	τ	::=	$\tau^{\text{XI}} \mid \tau^{\text{SX}}$
Constants	c	::=	$() \mid n \mid \texttt{true} \mid \texttt{false}$
Expressions	e	::=	$c \mid p \mid \&r\ \omega\ p \mid \&r\ \omega\ p[e] \mid \&r\ \omega\ p[\hat{e}_1..\hat{e}_2] \mid p := e$
		$\mid$	$\texttt{letprov} <r> \{ e \} \mid \texttt{let}\ x : \tau^{\text{SI}} = e_1; e_2 \mid e_1; e_2$
		$\mid$	$\lvert x_1 : \tau_1^{\text{SI}}, \ldots, x_n : \tau_n^{\text{SI}} \rvert \rightarrow \tau_r^{\text{SI}} \{ e \} \mid e_f ::<\overline{\Phi}, \overline{\rho}, \overline{\tau^{\text{SI}}}> (\hat{e}_1, \ldots, \hat{e}_n)$
		$\mid$	$\texttt{if}\ e_1 \{ e_2 \}\ \texttt{else} \{ e_3 \} \mid [\hat{e}_1, \ldots, \hat{e}_n] \mid (\hat{e}_1, \ldots, \hat{e}_n)$
		$\mid$	$p[e] \mid \texttt{for}\ x\ \texttt{in}\ e_1 \{ e_2 \} \mid \texttt{while}\ e_1 \{ e_2 \} \mid \texttt{abort!(str)}$
Frame Expressions	Φ	::=	$\varphi \mid \mathcal{F}$
Global Environment	Σ	::=	$\bullet \mid \Sigma, \varepsilon$
Global Entries	ε	::=	$\texttt{fn}\ f <\overline{\varphi}, \overline{\varrho}, \overline{\alpha}> (x_1 : \tau_1^{\text{SI}}, \ldots, x_n : \tau_n^{\text{SI}}) \rightarrow \tau_r^{\text{SI}}\ \texttt{where}\ \overline{\varrho_1 : \varrho_2} \{ e \}$
Type Environment	Δ	::=	$\bullet \mid \Delta, \alpha : \star \mid \Delta, \varrho : \texttt{PRV} \mid \Delta, \varphi : \texttt{FRM} \mid \Delta, \varrho :> \varrho'$
Frame Typing	$\mathcal{F}$	::=	$\bullet \mid \mathcal{F}, x : \tau^{\text{SX}} \mid \mathcal{F}, r \mapsto \{ \overline{\ell} \}$
Stack Typing	Γ	::=	$\bullet \mid \Gamma \natural \mathcal{F}$

Fig. 2. Syntax of Oxide

Provenances. Provenances ρ have two forms: abstract provenances ϱ (pronounced *var-rho*) and local provenances r. Abstract provenances correspond to lifetime variables `'a`, `'b`, etc. in Rust, and are used polymorphically in function types to indicate that they are agnostic to the particular provenances of references. Local provenances, by contrast, carry concrete information in the stack typing Γ consisting of a set of loans. A loan ${}^{\omega}p$ indicates a possible origin (p), qualified by whether the loan is unique or shared (ω). Intuitively, each loan tells us a single possible origin for a reference, while a provenance maps to *all* possible origins. As we will see in §3.2, provenances are essential to enabling our type system to guarantee the correct use of unique and shared references.

Types and Kinds. Oxide has three kinds κ: the kind of ordinary types $\star$, the kind of provenances PRV, and the kind of *frame typings* FRM. (Frame typings are relevant for closures, as we'll see below.) We abstract over variables of each kind in Oxide and, to aid the reader, we have separate syntax for each: α, ϱ, and φ, respectively. Since Rust is a fairly low-level language, Oxide makes a syntactic

distinction[4] between a few different sorts of types: sized & initialized types τ^{SI}, maybe-unsized & initialized types τ^{XI}, sized & dead types τ^{SD}, and sized & maybe-dead types τ^{SX}. Note that slices $[\tau^{\text{SI}}]$ are the only unsized type as they represent dynamically-sized segments of an array.

The bulk of types in Oxide are sized & initialized types, which include base types τ^{B}, type variables α, arrays $[\tau^{\text{SI}}; n]$, tuple types $(\tau_1^{\text{SI}}, \ldots, \tau_n^{\text{SI}})$, reference types $\&\rho\ \omega\ \tau^{\text{XI}}$, and function types $\forall <\overline{\varphi}, \overline{\varrho}, \overline{\alpha}> (\tau_1^{\text{SI}}, \ldots, \tau_n^{\text{SI}}) \xrightarrow{\Phi} \tau_r^{\text{SI}}$ `where` $\overline{\varrho_1 : \varrho_2}$. With the exception of references, any types that occur within these types are themselves required to be both sized and initialized.

The two interesting types here are reference and function types. For reference types $\&\rho\ \omega\ \tau^{\text{XI}}$, we include both the provenance ρ and ownership qualifier ω in the type which allow us to understand statically both a reference's origin as well as its aliasing requirements. We allow potentially unsized types under references since the reference itself will always have a fixed size regardless of what it points to (e.g. 64-bit on a 64-bit machine). For function types, there are three notable features. First, each function type can possibly include a frame expression Φ over the arrow indicating what bindings, if any, were caught up in the closed environment (when nothing is captured, we put nothing over the arrow). Next, functions are polymorphic in type and provenance variables, as well as in frame variables φ to enable the use of higher-order functions. Finally, functions can relate types with abstract provenances using outlives bounds, where $\varrho_1 : \varrho_2$ means ϱ_1 outlives ϱ_2.

Expressions. Expressions e in Oxide are numerous, but largely standard. For example, our constants c consist of the unit value ( ), unsigned 32-bit integers n, and boolean values `true` and `false`. The most interesting expressions in Oxide are the ones we've already seen by example: place expression usage (written simply p) and borrowing (with several forms that we will explain shortly). The former should be thought of like variable expressions that behave linearly (removing the place from the environment after use) for non-copyable types, and traditionally for copyable types. There are three borrowing forms overall, and all work fundamentalley the same — they are each used as introduction forms for references. The simplest case is written $\&r\ \omega\ p$, introducing an ω-reference with provenance r directly to the place π that the place expression p evaluates to. The next form borrows from $p[e]$ instead of simply p, and is used to borrow an element out of an array or slice p at the index given by e. The final form borrows from $p[e_1..e_2]$, and is used to borrow a slice of p using the range given by e_1 and e_2.

In these last two cases, one might wonder "why are indexing and slicing not places themselves?" The answer comes in two parts: (1) indexing and slicing take arbitrary expressions, while places are entirely static, and (2) unlike tuple projections which have a fine-grained notion of ownership, indexing and slicing affect the ownership of the array or slice overall. This second part means that while you can create two unique references to different projections of the same tuple, you cannot create two unique references to different indices of an array.

The remainder of our expressions are standard or discussed already, but we will draw attention to a few points of note. Our closure syntax follows the syntax of Rust, and thus uses vertical bars to denote the closure's parameters. As in Rust, closures are not polymorphic; only global functions may be polymorphic and specify outlives bounds. We use function application when applying closures as well as global functions. Hence, function application additionally includes polymorphic instantiation written using Rust's turbofish syntax (`::<>`). Finally, `abort!(str)` indicates irrecoverable failure, and thus terminates the program with the given string as a diagnostic message.

Several expressions have subexpressions $\hat{e}$, which denote *sequenceless expressions* (grammar elided). These are identical to expressions except that they may not contain any (nested) sequencing

[4] One may wonder why these types don't all have their own kinds. Since type polymorphism is restricted to sized & initialized types, we have not needed additional kinds. We could support more rich type polymorphism by enriching the kind system.

or `let` expressions. This allows us to control the effects that happen to our stack typing between expressions that should be thought of statically as evaluating at the same time.

Environments. As we have already seen in §2.4, we have three environments for type-checking in Oxide. First, we have a global environment Σ that consists of top-level function definitions. Next, we have a type environment Δ that contains type variables α, provenance variables ϱ, and frame variables φ, with their respective kinds. Additionally, it tracks outlives relations $\varrho :> \varrho'$, which correspond directly to the outlives bounds in function definitions and types.

Finally, we have a stack typing Γ which is organized as a sequence of frame typings $\mathcal{F}$. Frame typings track in-scope variable bindings x and their types, as well as in-scope local provenances r and their corresponding loan sets. The information in the stack and frame typing together describes the shape and validity of the stack σ at runtime. The separation into frames is useful for closures: when typing a closure in environment Γ, we add a new frame with appropriate bindings to the end of Γ to type-check the body of the closure. We assume that all variables x and provenances r in Γ are unique (no reuse/shadowing allowed, and we assume implicit alpha-renaming to enforce name uniqueness). More significantly, Γ and $\mathcal{F}$ are *ordered*. We rely on the ordering and on frames in several ways as we'll point out — e.g., when deciding if one provenance r outlives another and to ensure we only drop places in the current frame.

3.2 Type System

Figure 3 presents a selection of Oxide typing rules. In every rule, we highlight the expression being typechecked with a framebox. As described in §2.4, the shape of our typing judgement is $\Sigma; \Delta; \Gamma \vdash \boxed{e} : \tau \Rightarrow \Gamma'$: we type-check e in environments Σ, Δ, and Γ, producing Γ' which is the stack typing for the continuation of e. These rules rely on the subtyping and outlives judgments (Figure 5) and the ownership safety judgment (Figure 4), which we'll discuss below. We elide the various well-formedness judgments (for types, stack typings, etc.); see the appendix (§B.1).

Moving. The T-Move rule, which was first introduced in §2.4, type-checks place usage that must *move* the value out of π. Here, we are restricted to places, rather than the more general place expressions because Rust disallows moving values from under references. As such, it requires three things: (1) π must be able to be used `uniq`-ly (checked using the ownership safety judgment in Figure 4, discussed later); (2) π must have a sized type τ^{SI} in Γ; and (3) the type of π is `noncopyable`. Requirement (1) is needed to ensure that we do not invalidate any existing references to π by moving it. Requirement (3) says we should use T-Copy, which is more permissive, when π has a `copyable` type. When the premises hold, the output environment updates the type of π to include a dagger — marking it dead — reflecting that it has been moved after type-checking the expression.

If the type *is* instead `copyable`,[5] we use T-Copy which requires that p is safe to use as `shrd`. We leave the output environment unchanged since the value will be copied from the stack at runtime.

Borrowing. In §2.4, we also introduced T-Borrow, which requires that the place expression p be safe to use (again checked with ownership safety, Figure 4), and if so, updates the loan set for r to incorporate the new borrow chain $\{ \overline{\ell} \}$ from ownership safety. It is important to note that although simple uses of T-Borrow (such as directly borrowing a newly-bound variable) only introduce trivial provenances — in general, these provenances are *approximate*. We will see below how they are combined via subtyping (Figure 5) and when type-checking branches to yield larger loan sets.

[5] We've elided definitions of `copyable` and `noncopyable`, but they're straightforward. Intuitively, a type is safe to copy if none of its constituent parts are unique. Thus, all types that don't contain a unique reference are copyable. Generic types are always non-copyable. In Rust, `copyable` is actually the `Copy` trait, but `copyable` can be thought of as special casing it.

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

T-Move

$$\frac{\Delta;\ \Gamma \vdash_{\mathsf{uniq}} \pi \Rightarrow \{\ {}^{\mathsf{uniq}}\pi\ \} \qquad \Gamma(\pi) = \tau^{\mathrm{SI}} \qquad \mathsf{noncopyable}_{\Sigma}\ \tau^{\mathrm{SI}}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\pi} : \tau^{\mathrm{SI}} \Rightarrow \Gamma[\pi \mapsto \tau^{\mathrm{SI}^{\dagger}}]}$$

T-Copy

$$\frac{\Delta;\ \Gamma \vdash_{\mathsf{shrd}} p \Rightarrow \{\ \overline{\ell}\ \} \qquad \Delta;\ \Gamma \vdash_{\mathsf{shrd}} p : \tau^{\mathrm{SI}} \qquad \mathsf{copyable}_{\Sigma}\ \tau^{\mathrm{SI}}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{p} : \tau^{\mathrm{SI}} \Rightarrow \Gamma}$$

T-Borrow

$$\frac{\Gamma(r) = \emptyset \qquad \Delta;\ \Gamma \vdash_{\omega} p \Rightarrow \{\ \overline{\ell}\ \} \qquad \Delta;\ \Gamma \vdash_{\omega} p : \tau^{\mathrm{XI}}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\&r\ \omega\ p} : \&r\ \omega\ \tau^{\mathrm{XI}} \Rightarrow \Gamma[r \mapsto \{\ \overline{\ell}\ \}]}$$

T-Branch

$$\frac{\begin{array}{c}\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e_1} : \mathtt{bool} \Rightarrow \Gamma_1 \qquad \Sigma;\ \Delta;\ \Gamma_1 \vdash \boxed{e_2} : \tau_2^{\mathrm{SI}} \Rightarrow \Gamma_2 \\ \Sigma;\ \Delta;\ \Gamma_1 \vdash \boxed{e_3} : \tau_3^{\mathrm{SI}} \Rightarrow \Gamma_3 \qquad \tau^{\mathrm{SI}} = \tau_2^{\mathrm{SI}} \vee \tau^{\mathrm{SI}} = \tau_3^{\mathrm{SI}} \\ \Delta;\ \Gamma_2 \vdash \tau_2^{\mathrm{SI}} \lesssim \tau^{\mathrm{SI}} \Rightarrow \Gamma_2' \qquad \Delta;\ \Gamma_3 \vdash \tau_3^{\mathrm{SI}} \lesssim \tau^{\mathrm{SI}} \Rightarrow \Gamma_3' \qquad \Gamma_2' \Cup \Gamma_3' = \Gamma'\end{array}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\mathtt{if}\ e_1\ \{\ e_2\ \}\ \mathtt{else}\ \{\ e_3\ \}} : \tau^{\mathrm{SI}} \Rightarrow \Gamma'}$$

T-Seq

$$\frac{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e_1} : \tau_1^{\mathrm{SI}} \Rightarrow \Gamma_1 \qquad \Sigma;\ \Delta;\ \mathsf{gc\text{-}loans}(\Gamma_1) \vdash \boxed{e_2} : \tau_2^{\mathrm{SI}} \Rightarrow \Gamma_2}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e_1;\ e_2} : \tau_2^{\mathrm{SI}} \Rightarrow \Gamma_2}$$

T-Let

$$\frac{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e_1} : \tau_1^{\mathrm{SI}} \Rightarrow \Gamma_1 \qquad \Delta;\ \Gamma_1 \vdash \tau_1^{\mathrm{SI}} \lesssim \tau_a^{\mathrm{SI}} \Rightarrow \Gamma_1' \qquad \Sigma;\ \Delta;\ \mathsf{gc\text{-}loans}(\Gamma_1',\ x\ :\ \tau_a^{\mathrm{SI}}) \vdash \boxed{e_2} : \tau_2^{\mathrm{SI}} \Rightarrow \Gamma_2,\ x\ :\ \tau^{\mathrm{SD}}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\mathtt{let}\ x : \tau_a^{\mathrm{SI}} = e_1;\ e_2} : \tau_2^{\mathrm{SI}} \Rightarrow \Gamma_2}$$

T-LetProv

$$\frac{\Sigma;\ \Delta;\ \Gamma,\ r \mapsto \{\} \vdash \boxed{e} : \tau^{\mathrm{SI}} \Rightarrow \Gamma',\ r \mapsto \{\overline{\ell}\}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\mathtt{letprov}\ \mathtt{<}r\mathtt{>}\ \{\ e\ \}} : \tau^{\mathrm{SI}} \Rightarrow \Gamma'}$$

T-Assign

$$\frac{\begin{array}{c}\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e} : \tau^{\mathrm{SI}} \Rightarrow \Gamma_1 \qquad \Gamma_1(\pi) = \tau^{\mathrm{SX}} \\ (\tau^{\mathrm{SX}} = \tau^{\mathrm{SD}} \vee \Delta;\ \Gamma_1 \vdash_{\mathsf{uniq}} \pi \Rightarrow \{\ {}^{\mathsf{uniq}}\pi\ \}) \\ \Delta;\ \Gamma_1 \vdash \tau^{\mathrm{SI}} \lesssim \tau^{\mathrm{SX}} \Rightarrow \Gamma'\end{array}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\pi \coloneqq e} : \mathtt{unit} \Rightarrow \Gamma'[\pi \mapsto \tau^{\mathrm{SI}}] \Subset \pi}$$

T-AssignDeref

$$\frac{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e} : \tau_n^{\mathrm{SI}} \Rightarrow \Gamma_1 \qquad \Delta;\ \Gamma_1 \vdash_{\mathsf{uniq}} p : \tau_o^{\mathrm{SI}} \qquad \Delta;\ \Gamma_1 \vdash_{\mathsf{uniq}} p \Rightarrow \{\ \overline{\ell}\ \} \qquad \Delta;\ \Gamma_1 \vdash \tau_n^{\mathrm{SI}} \lesssim \tau_o^{\mathrm{SI}} \Rightarrow \Gamma'}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{p \coloneqq e} : \mathtt{unit} \Rightarrow \Gamma' \Subset p}$$

T-Closure

$$\frac{\mathsf{free\text{-}vars}(e) \setminus \overline{x} = \overline{x_f} \qquad \mathsf{free\text{-}nc\text{-}vars}_{\Gamma}(e) = \overline{x_{nc}} \qquad \overline{r} = \overline{\mathsf{free\text{-}provs}(\Gamma(x_f))},\ \mathsf{free\text{-}provs}(e) \qquad \mathcal{F}_c = \overline{r \mapsto \Gamma(r)},\ \overline{x_f\ :\ \Gamma(x_f)} \qquad \Sigma;\ \Delta;\ \Gamma[\overline{x_{nc} \mapsto \Gamma(x_{nc})^{\dagger}}] \uplus \mathcal{F}_c,\ x_1\ :\ \tau_1^{\mathrm{SI}},\ \ldots,\ x_n\ :\ \tau_n^{\mathrm{SI}} \vdash \boxed{e} : \tau_r^{\mathrm{SI}} \Rightarrow \Gamma' \uplus \mathcal{F}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{|x_1 : \tau_1^{\mathrm{SI}},\ \ldots,\ x_n : \tau_n^{\mathrm{SI}}|\ \rightarrow\ \tau_r^{\mathrm{SI}}\ \{\ e\ \}} : (\tau_1^{\mathrm{SI}},\ \ldots,\ \tau_n^{\mathrm{SI}}) \xrightarrow{\mathcal{F}_c} \tau_r^{\mathrm{SI}} \Rightarrow \Gamma'}$$

T-App

$$\frac{\begin{array}{c}\overline{\Sigma;\ \Delta;\ \Gamma \vdash \Phi} \qquad \overline{\Delta;\ \Gamma \vdash \rho} \qquad \overline{\Sigma;\ \Delta;\ \Gamma \vdash \tau^{\mathrm{SI}}} \\ \Sigma;\ \Delta;\ \Gamma \vdash \boxed{\hat{e}_f} : \forall \mathtt{<}\overline{\varphi},\ \overline{\varrho},\ \overline{\alpha}\mathtt{>}(\tau_1^{\mathrm{SI}},\ \ldots,\ \tau_n^{\mathrm{SI}}) \xrightarrow{\Phi_c} \tau_f^{\mathrm{SI}}\ \mathtt{where}\ \overline{\varrho_1 : \varrho_2} \Rightarrow \Gamma_0 \\ \forall i \in \{\ 1\ \ldots\ n\ \}.\ \Sigma;\ \Delta;\ \Gamma_{i-1} \vdash \boxed{\hat{e}_i} : \tau_i^{\mathrm{SI}}\ \overline{[\Phi/_{\varphi}]}\ \overline{[\rho/_{\varrho}]}\ \overline{[\tau^{\mathrm{SI}}/_{\alpha}]} \Rightarrow \Gamma_i \qquad \Delta;\ \Gamma_n \vdash \overline{\varrho_2\ \overline{[\rho/_{\varrho}]} :> \varrho_1\ \overline{[\rho/_{\varrho}]}} \Rightarrow \Gamma_b\end{array}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\hat{e}_f ::\mathtt{<}\overline{\Phi},\ \overline{\rho},\ \overline{\tau^{\mathrm{SI}}}\mathtt{>}(\hat{e}_1,\ \ldots,\ \hat{e}_n)} : \tau_f^{\mathrm{SI}}\ \overline{[\Phi/_{\varphi}]}\ \overline{[\rho/_{\varrho}]}\ \overline{[\tau^{\mathrm{SI}}/_{\alpha}]} \Rightarrow \Gamma_b}$$

T-True

$$\frac{}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\mathtt{true}} : \mathtt{bool} \Rightarrow \Gamma}$$

T-False

$$\frac{}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\mathtt{false}} : \mathtt{bool} \Rightarrow \Gamma}$$

T-u32

$$\frac{}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{n} : \mathtt{u32} \Rightarrow \Gamma}$$

T-Tuple

$$\frac{\forall i \in \{\ 1\ \ldots\ n\ \}.\ \Sigma;\ \Delta;\ \Gamma_{i-1} \vdash \boxed{\hat{e}_i} : \tau_i^{\mathrm{SI}} \Rightarrow \Gamma_i}{\Sigma;\ \Delta;\ \Gamma_0 \vdash \boxed{(\hat{e}_1,\ \ldots,\ \hat{e}_n)} : (\tau_1^{\mathrm{SI}},\ \ldots,\ \tau_n^{\mathrm{SI}}) \Rightarrow \Gamma_n}$$

T-While

$$\frac{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e_1} : \mathtt{bool} \Rightarrow \Gamma_1 \qquad \Sigma;\ \Delta;\ \Gamma_1 \vdash \boxed{e_2} : \mathtt{unit} \Rightarrow \Gamma_2 \qquad \Sigma;\ \Delta;\ \Gamma_2 \vdash \boxed{e_1} : \mathtt{bool} \Rightarrow \Gamma_2 \qquad \Sigma;\ \Delta;\ \Gamma_2 \vdash \boxed{e_2} : \mathtt{unit} \Rightarrow \Gamma_2}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\mathtt{while}\ e_1\ \{\ e_2\ \}} : \mathtt{unit} \Rightarrow \Gamma_2}$$

T-Abort

$$\frac{}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\mathtt{abort!(str)}} : \tau^{\mathrm{SX}} \Rightarrow \Gamma'}$$

T-Drop

$$\frac{\Gamma(\pi) = \tau_{\pi}^{\mathrm{SI}} \qquad \Sigma;\ \Delta;\ \Gamma[\pi \mapsto \tau_{\pi}^{\mathrm{SI}^{\dagger}}] \vdash \boxed{e} : \tau^{\mathrm{SX}} \Rightarrow \Gamma_f}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e} : \tau^{\mathrm{SX}} \Rightarrow \Gamma_f}$$

Fig. 3. Selected Oxide Typing Rules

$\boxed{\Delta;\ \Gamma \vdash^{\overline{\pi}}_{\omega} p \Rightarrow \{\ \overline{^{\omega}p}\ \}}$ where $\Delta;\ \Gamma \vdash_{\omega} p \Rightarrow \{\ \overline{^{\omega}p}\ \}$ means $\Delta;\ \Gamma \vdash^{\bullet}_{\omega} p \Rightarrow \{\ \overline{^{\omega}p}\ \}$.

O-SafePlace

$$\frac{\begin{array}{c}\forall r' \mapsto \{\ \overline{\ell}\ \} \in \Gamma.\ (\forall\, ^{\omega'}p^{\square}[\pi'] \in \{\ \overline{\ell}\ \}.(\omega = \texttt{uniq} \vee \omega' = \texttt{uniq}) \implies \pi' \# \pi) \\ \vee\ (\exists \pi' : \&r'\ \omega'\ \tau' \in \Gamma \wedge (\forall \pi' : \&r'\ \omega'\ \tau' \in \Gamma.\ \pi' \in \{\ \overline{\pi_e}\ \}))\end{array}}{\Delta;\ \Gamma \vdash^{\overline{\pi_e}}_{\omega} \pi \Rightarrow \{\ ^{\omega}\pi\ \}}$$

O-Deref

$$\frac{\begin{array}{c}\Gamma(\pi) = \&r\ \omega_\pi\ \tau_\pi \qquad \Gamma(r) = \{\ \overline{^{\omega'}p_i}\ \} \qquad \overline{p_i = p_i^{\square}[\pi_i]} \qquad \omega \lesssim \omega_\pi \\ \forall i \in \{1 \ldots n\}.\ \Delta;\ \Gamma \vdash^{\overline{\pi_e}, \overline{\pi_i}, \pi}_{\omega} p^{\square}[p_i] \Rightarrow \{\ \overline{^{\omega}p'_i}\ \} \\ \forall r' \mapsto \{\ \overline{\ell}\ \} \in \Gamma.\ (\forall\, ^{\omega'}p \in \{\ \overline{\ell}\ \}.(\omega = \texttt{uniq} \vee \omega' = \texttt{uniq}) \implies p \# p^{\square}[*\pi]) \\ \vee\ (\exists \pi' : \&r'\ \omega'\ \tau' \in \Gamma \wedge (\forall \pi' : \&r'\ \omega'\ \tau' \in \Gamma.\ \pi' \in \{\ \overline{\pi_e}, \overline{\pi_i}, \pi\ \}))\end{array}}{\Delta;\ \Gamma \vdash^{\overline{\pi_e}}_{\omega} p^{\square}[*\pi] \Rightarrow \{\ \overline{^{\omega}p'_1}, \ \ldots\ \overline{^{\omega}p'_n}, \ ^{\omega}p^{\square}[*\pi]\ \}}$$

O-DerefAbs

$$\frac{\begin{array}{c}\Gamma(\pi) = \&\varrho\ \omega_\pi\ \tau_\pi \qquad \Delta;\ \Gamma \vdash_{\omega} p^{\square}[*\pi] : \tau \qquad \omega \lesssim \omega_\pi \\ \forall r' \mapsto \{\ \overline{\ell}\ \} \in \Gamma.\ (\forall\, ^{\omega'}p \in \{\ \overline{\ell}\ \}.(\omega = \texttt{uniq} \vee \omega' = \texttt{uniq}) \implies p \# p^{\square}[*\pi]) \\ \vee\ (\exists \pi' : \&r'\ \omega'\ \tau' \in \Gamma \wedge (\forall \pi' : \&r'\ \omega'\ \tau' \in \Gamma.\ \pi' \in \{\ \overline{\pi_e}, \pi\ \}))\end{array}}{\Delta;\ \Gamma \vdash^{\overline{\pi_e}}_{\omega} p^{\square}[*\pi] \Rightarrow \{\ ^{\omega}p^{\square}[*\pi]\ \}}$$

Fig. 4. Ownership Safety in Oxide

Ownership Safety. Figure 4 presents the rules for ownership safety, which we've already explained at a high level in §2.4. What we did not explain earlier is how ownership safety handles reborrowing. The full form of the judgment is $\Delta;\ \Gamma \vdash^{\overline{\pi}}_{\omega} p \Rightarrow \{\ \overline{^{\omega}p}\ \}$, which says that p is ω-safe under Δ and Γ, with *reborrow exclusion list* $\overline{\pi}$, and may point to any of the loans in $\overline{^{\omega}p}$ (called the borrow chain). The first rule, O-SafePlace, checks if a place π is ω-safe by looking at each loan in every provenance r' in Γ and either (1) making sure that if either that loan or ω is `uniq` then π does not overlap with the loan; or (2) checking that all references in Γ with provenance r' are in the reborrow exclusion list (meaning we need not check if there is overlap with π).

The next two rules check if a place expression p is ω-safe, decomposing the place expression into a place expression context $p^{\square}$ with $*\pi$ in the hole. The last two lines of premises for both essentially ensure that either (1) or (2) holds, but each one adds to the incoming reborrow exclusion list when checking (2) by collecting any additional places dereferenced in p. Both rules also check $\omega \lesssim \omega_\pi$ (defined as the reflexive closure of $\texttt{shrd} \lesssim \texttt{uniq}$) in order to ensure that the reference has sufficient permission to be used, preventing a dereference of a `uniq` reference in a `shrd` context.

Unlike O-DerefAbs, O-Deref is dereferencing a reference π with a concrete provenance r. As such, we can look at the loans present for r in the stack typing. These loans consist of both direct loans to places π_i which correspond to a possible origin for the reference, and indirect loans to place expressions p_i which capture how this reference was reborrowed from other references. As such, when we recursively check for the safety of these origins, we append the reborrow origins (the π_i prefixes of these p_i) to the reborrow exclusion list. This means that they will not be considered as possible conflicts in the rest of ownership safety. At the end, we union together the borrow chains from all the possible origins to determine our final borrow chain. We also include an additional loan $^{\omega}p^{\square}[*\pi]$ to indicate that this use was reborrowed from $*\pi$.

Subtyping and Outlives. We examine subtyping next (Figure 5) since some of the typing rules discussed below require it. The subtyping judgment $\Delta;\ \Gamma \vdash \tau_1 \lesssim \tau_2 \Rightarrow \Gamma'$ says τ_1 is a subtype of τ_2 under Δ and Γ, producing Γ'. We produce an output Γ' with updated provenances to be used

when typing the continuation after an appeal to subtyping. Note that our subtyping judgment is *not* allowed arbitrarily, but instead is used specifically in T-Let and T-Branch.

Subtyping is largely standard excepting the output stack typing: it is reflexive and transitive; covariant for arrays, slices, tuples, and shared references; non-variant for unique references which support mutation. A reference type is a subtype of another if the provenance on the subtype *outlives* the provenance on the supertype (S-SharedRef and S-UniqueRef).

The outlives judgment (Figure 5) $\Delta;\ \Gamma \vdash \rho_1 :> \rho_2 \Rightarrow \Gamma'$ says ρ_1 outlives ρ_2 under Δ and Γ, producing Γ'. Every provenance outlives itself (reflexivity). An abstract provenance outlives another if there's a corresponding outlives relation in Δ (OL-AbstractProvenances) or if we can transitively put together outlives relations from Δ (OL-Trans). [6] OL-LocalProvenances says that r_1 outlives r_2 if it occurs earlier than r_2 in Γ. It also requires that there not exist any references with the provenance r_1 which have been reborrowed ($\forall \pi : \&r_1\ \omega\ \tau \in \Gamma.\ \nexists r'.\ {}^{\omega} * \pi\ \in \Gamma(r')$).

The last two rules say when a local provenance outlives an abstract one and vice versa. In essence, a local provenance r can only outlive an abstract provenance ϱ (OL-LocalProvAbsProv) if r was *reborrowed*. The first two premises check for reborrowing: r's loan set must be non-empty (otherwise there is no reborrow), and must consist solely of place expressions $\overline{p}$ (since place expressions, unlike places, contain dereferences, which identifies this as a reborrow instead of a borrow). The third premise collects all the provenances $\overline{\rho_i}$ that annotate any references dereferenced in each place expression p_i (see the place-expression type-computation judgment $\Delta;\ \Gamma \vdash_{\omega} p : \tau,\ \{\ \overline{\rho}\ \}$ in the appendix (§B.5)), while the last premise ensures that all of these outlive ϱ. The final rule, OL-AbsProvLocalProv, says that an abstract provenance *always* outlives a local provenance. This is subtle but makes sense because any abstract provenance ϱ is bound in a top-level function (recall that closures don't abstract over provenances), while a local provenance r must be bound by `letprovs` *inside* the function body. Ultimately, any local provenance r' that gets substituted for ϱ upon application will already exist before r (even for recursive calls), which means it outlives r.

Branching and Sequencing. The next two rules illustrate how stack typings are threaded through larger programs since the form of our typing judgment requires each rule to specify its continuation's stack typing. T-Branch uses the stack typing Γ_1 that we get from typing the conditional e_1 when typing each of the two branches. The type τ^{SI} ascribed to the overall expression must be a supertype of the types of both branches and equal to one of them. Additionally, branching uses a union operation $\Cup$ to combine the output stack typings from each branch to produce the final stack typing Γ' for the overall expression. This union requires that types of bound variables x in the two stack typings be equal (which potentially demands use of T-Drop and T-Subsumption when typing the branches), and unions the loan sets for each provenance r from both stack typings. Note that we only need to union stack typings with identical domains — we type-check both branches under Γ_1 so they produce output stack typings with the same domains (since `let` and `letprov` are the only means for introducing variables and provenances, but both are lexically scoped), and subtyping does not change the domain of stack typings from input to output.

When typing $e_1; e_2$, we type-check e_2 under the stack typing Γ_1 we got from type-checking e_1. But, importantly, we apply a metafunction gc-loans$(\cdot)$ to Γ_1 to empty out the loan sets of provenances not used in the stack typing before typing e_2 because e_1 may have been a unique reference that is thrown away at runtime before moving on to e_2. Without garbage collecting loans, Oxide would reject programs that are safe and accepted by Rust. Specifically, gc-loans(Γ) empties out the loan set of each r that does not appear in any types in Γ or in τ.

[6] We do not need transitivity for concrete provenances beyond what we can already conclude from the remaining OL rules.

$\boxed{\Delta;\ \Gamma \vdash \tau_1 \lesssim \tau_2 \Rightarrow \Gamma'}$

$$\textsc{S-Refl}\ \frac{}{\Delta;\ \Gamma \vdash \tau_1 \lesssim \tau_1 \Rightarrow \Gamma} \qquad \textsc{S-Trans}\ \frac{\Delta;\ \Gamma \vdash \tau_1 \lesssim \tau_2 \Rightarrow \Gamma' \qquad \Delta;\ \Gamma' \vdash \tau_2 \lesssim \tau_3 \Rightarrow \Gamma''}{\Delta;\ \Gamma \vdash \tau_1 \lesssim \tau_3 \Rightarrow \Gamma''} \qquad \textsc{S-Array}\ \frac{\Delta;\ \Gamma \vdash \tau_1 \lesssim \tau_2 \Rightarrow \Gamma'}{\Delta;\ \Gamma \vdash [\tau_1;\ n] \lesssim [\tau_2;\ n] \Rightarrow \Gamma'}$$

$$\textsc{S-Slice}\ \frac{\Delta;\ \Gamma \vdash \tau_1 \lesssim \tau_2 \Rightarrow \Gamma'}{\Delta;\ \Gamma \vdash [\tau_1] \lesssim [\tau_2] \Rightarrow \Gamma'} \qquad \textsc{S-Tuple}\ \frac{\forall i \in \{\,1 \ldots n\,\}.\ \Delta;\ \Gamma_{n-1} \vdash \tau_i \lesssim \tau_i' \Rightarrow \Gamma_i}{\Delta;\ \Gamma \vdash (\tau_1 \ \ldots \ \tau_n) \lesssim (\tau_1' \ \ldots \ \tau_n') \Rightarrow \Gamma_n} \qquad \textsc{S-Uninit}\ \frac{\Delta;\ \Gamma \vdash \tau_1^{\mathrm{SI}} \lesssim \tau_2^{\mathrm{SI}} \Rightarrow \Gamma'}{\Delta;\ \Gamma \vdash \tau_1^{\mathrm{SI}} \lesssim \tau_2^{\mathrm{SI}^{\dagger}} \Rightarrow \Gamma}$$

$$\textsc{S-SharedRef}\ \frac{\Delta;\ \Gamma \vdash \rho_1 :> \rho_2 \Rightarrow \Gamma' \qquad \Delta;\ \Gamma' \vdash \tau_1 \lesssim \tau_2 \Rightarrow \Gamma''}{\Delta;\ \Gamma \vdash \&\rho_1\ \mathtt{shrd}\ \tau_1 \lesssim \&\rho_2\ \mathtt{shrd}\ \tau_2 \Rightarrow \Gamma''} \qquad \textsc{S-UniqueRef}\ \frac{\Delta;\ \Gamma \vdash \rho_1 :> \rho_2 \Rightarrow \Gamma' \qquad \Delta;\ \Gamma' \vdash \tau_1 \lesssim \tau_2 \Rightarrow \Gamma'' \qquad \Delta;\ \Gamma' \vdash \tau_2 \lesssim \tau_1 \Rightarrow \Gamma''}{\Delta;\ \Gamma \vdash \&\rho_1\ \mathtt{uniq}\ \tau_1 \lesssim \&\rho_2\ \mathtt{uniq}\ \tau_2 \Rightarrow \Gamma''}$$

$\boxed{\Delta;\ \Gamma \vdash \rho_1 :> \rho_2 \Rightarrow \Gamma'}$

$$\textsc{OL-Refl}\ \frac{}{\Delta;\ \Gamma \vdash \rho :> \rho \Rightarrow \Gamma} \qquad \textsc{OL-AbstractProvenances}\ \frac{\varrho_1 : \mathsf{PRV} \in \Delta \qquad \varrho_2 : \mathsf{PRV} \in \Delta \qquad \varrho_1 :> \varrho_2 \in \Delta}{\Delta;\ \Gamma \vdash \varrho_1 :> \varrho_2 \Rightarrow \Gamma} \qquad \textsc{OL-Trans}\ \frac{\Delta;\ \Gamma \vdash \varrho_1 :> \varrho_2 \Rightarrow \Gamma' \qquad \Delta;\ \Gamma' \vdash \varrho_2 :> \varrho_3 \Rightarrow \Gamma''}{\Delta;\ \Gamma \vdash \varrho_1 :> \varrho_3 \Rightarrow \Gamma''}$$

$$\textsc{OL-LocalProvenances}\ \frac{\forall \pi : \&r_1\ \omega\ \tau \in \Gamma.\ \nexists r'.\ {}^{\omega} * \pi\ \in \Gamma(r') \qquad r_1\ \mathtt{occurs\ before}\ r_2\ \mathtt{in}\ \Gamma}{\Delta;\ \Gamma \vdash r_1 :> r_2 \Rightarrow \Gamma[r_2 \mapsto \{\,\Gamma(r_1) \cup \Gamma(r_2)\,\}]} \qquad \textsc{OL-AbsProvLocalProv}\ \frac{\varrho : \mathsf{PRV} \in \Delta \qquad r \in \mathrm{dom}(\Gamma)}{\Delta;\ \Gamma \vdash \varrho :> r \Rightarrow \Gamma}$$

$$\textsc{OL-LocalProvAbsProv}\ \frac{\Gamma_{1,0}(r) = \{\ \overline{{}^{\omega} p}^{\,n}\ \} \neq \emptyset \qquad \overline{\forall \pi.\ p \neq \pi} \qquad \forall i \in \{\,1 \ldots n\,\}.\ \Delta;\ \Gamma_0 \vdash_{\mathsf{shrd}} p_i : \_,\ \overline{\rho_i}^{m_i} \qquad \varrho : \mathsf{PRV} \in \Delta \qquad \forall i \in \{\,1 \ldots n\,\}.\forall j \in \{\,1 \ldots m_i\,\}.\ \Delta;\ \Gamma_{i,j-1} \vdash \rho_{i,j} :> \varrho \Rightarrow \Gamma_{i,j}}{\Delta;\ \Gamma_{1,0} \vdash r :> \varrho \Rightarrow \Gamma_{n,m_n}}$$

Fig. 5. Subtyping and Outlives Relations in Oxide

Binding. In Oxide, T-Let is interesting in two ways. Similar to sequencing, T-Let uses the meta-function gc-loans$(\cdot)$ to eliminate any loans that might be unnecessary as a result of e_1 potentially being promoted to the annotated type τ_a^{SI}. Additionally, in the output stack typing from e_2, we see that our binding for x must have a dead type τ^{SD} with the whole binding being dropped in the overall stack typing Γ_2 output from T-Let (since the scope of x ends at that point). The requirement that the type be dead means we must have either used T-Move to move out of that binding or we must have explicitly used T-Drop on x in the derivation for e_2.

Assignment. Assignment is interesting in a few ways. First, assignment is broken up into two rules T-Assign and T-AssignDeref where the former is able to assign to a place π that is dead, and the latter is able to assign to a place through a reference (i.e. by using dereferencing). Otherwise, the two rules are essentially the same: they type-check the expression we're assigning, they compute the type of the place or place expression we're assigning to, they check that that place or place expression is safe to use uniquely, and they check that the two types are compatible with subtyping. Finally, we use the operation $\Gamma \Subset p$ to remove any loans prefixed by $*p$ from all loan sets in Γ.

Closures and Application. Closures in Oxide correspond to *move closures* in Rust which move or copy their free variables from the outer environment into the closure[7]. As such, T-Closure must

[7]Rust's standard closures implicitly introduce borrowed temporaries for all the free variables, and so we can recover this behavior via a simple, local transformation to move closures.

$$
\begin{array}{llll}
\text{Referent} & \mathcal{R} & ::= & x \mid \mathcal{R}.n \mid \mathcal{R}[n] \mid \mathcal{R}[n_1..n_2] \\
\text{Expressions} & e & ::= & \ldots \mid [\![v_1, \ldots, v_n]\!] \mid \texttt{dead} \mid \texttt{framed}\, e \mid \texttt{shift}\, e \mid \texttt{ptr}\, \mathcal{R} \\
 & & \mid & \langle \varsigma,\ |x_1 : \tau_1^{\mathrm{SI}}, \ldots, x_n : \tau_n^{\mathrm{SI}}|\ \rightarrow\ \tau_r^{\mathrm{SI}}\ \{\ e\ \}\ \rangle \\
\text{Values} & v & ::= & c \mid (v_1, \ldots, v_n) \mid [v_1, \ldots, v_n] \mid [\![v_1, \ldots, v_n]\!] \mid f \mid \texttt{dead} \mid \texttt{ptr}\, \mathcal{R} \\
 & & \mid & \langle \varsigma,\ |x_1 : \tau_1^{\mathrm{SI}}, \ldots, x_n : \tau_n^{\mathrm{SI}}|\ \rightarrow\ \tau_r^{\mathrm{SI}}\ \{\ e\ \}\ \rangle \\
\text{Value Contexts} & \mathcal{V} & ::= & \square \mid (v_1, \ldots, \mathcal{V}, \ldots, v_n) \mid [v_1, \ldots, \mathcal{V}_1, \ldots, \mathcal{V}_m, \ldots, v_n] \\
\text{Stacks} & \sigma & ::= & \bullet \mid \sigma \natural \varsigma \\
\text{Stack Frame} & \varsigma & ::= & \bullet \mid \varsigma,\ x \mapsto v
\end{array}
$$

Fig. 6. Oxide Syntax Extensions for Dynamics

compute the captured frame by looking at the free variables (and free provenances) of the closure's body, and it must mark dead (add daggers to the types of) any variables in the stack typing with *non-copyable* types. The captured frame is suspended over the arrow in the function type to keep track of the fact that the data caught up in the closure is ultimately still alive (and thus must be considered in ownership safety). We elide the simple rule for top-level function definitions, which gives a function named f the type that f is annotated with in Σ, relying on the well-formedness of Σ to know that this is okay.

The rule for application (T-App) is essentially an ordinary function application rule in environment-passing style, with two exceptions: (1) frame, type, and provenance variables are substituted in all the types since functions are polymorphic, and (2) application must check that the outlives relation (defined in Figure 5) holds for all the bounds specified in the function type.

Values and Aggregates. The typing rules for base types (T-u32, T-True, T-False, etc.) are standard, and leave the type environment unchanged in their output. Aggregate structures like tuples check the types of their components while threading through the environments in left-to-right order. This left-to-right ordering for type-checking corresponds to the ordering implemented by Rust's type checker and borrow checker. The formalism of Oxide omits a specific treatment of structs, but we note that they are essentially the same as tuples, only featuring a tag that must also be checked. Our implementation which we discuss in §4.3 relies on exactly this approach to support structs.

Remaining Rules. The remaining rules in Figure 3 are straightforward or covered earlier. Elided typing rules all concern arrays and are given in the technical appendix (§B.4).

3.3 Operational Semantics

For our operational semantics, we extend the syntax of Oxide in Figure 6 with terms that only arise at runtime. First, to be able to specify what "address" a pointer points to, we introduce an abstract form of memory addresses called *referents*. Referents $\mathcal{R}$ essentially record what the offsets are from a variable on the stack in order to specify a precise "memory address," (e.g., a particular element of an array or tuple, or a particular slice of an array). Next, we introduce value forms including pointers to referents, and closures packaged with their environment ς. Additionally, we include some administrative forms: (1) `dead` (the dead value), (2) `framed` e which is evaluated under and drops the top stack frame when eliminated, (3) `shift` e which is similar but drops the last binding when eliminated, and (4) $[\![v_1, \ldots, v_n]\!]$ which is a dynamically-sized slice of an array. Figure 6 also includes stacks σ which are a sequence of stack frames ς, and value contexts $\mathcal{V}$ which allow array values to be decomposed with multiple holes when dealing with slices.

In Figure 7, we present a selection of our small-step operational semantics which is defined using Felleisen and Hieb [1992]-style left-to-right evaluation contexts over configurations of the form $(\sigma;\ \boxed{e}\,)$. Since our semantics uses referents $\mathcal{R}$ as an abstract version of memory addresses, some of

$\boxed{\sigma \vdash p \Downarrow \mathcal{R} \mapsto v}$ $\quad \sigma \vdash p^{\square}[x] \Downarrow \mathcal{R} \mapsto v \stackrel{\text{def}}{=} \sigma \vdash p^{\square} \times x \Downarrow \mathcal{R} \mapsto v.$

$\boxed{\sigma \vdash p^{\square} \times \mathcal{R} \Downarrow \mathcal{R}' \mapsto v}$ read: "$\mathcal{R}$ in a context $p^{\square}$ computes to $\mathcal{R}'$ which maps to v in σ."

P-Referent
$$\frac{\sigma \vdash \mathcal{R} \Downarrow \_ \times v}{\sigma \vdash \square \times \mathcal{R} \Downarrow \mathcal{R} \mapsto v}$$

P-Proj
$$\frac{\sigma \vdash p^{\square} \times \mathcal{R}_1 \Downarrow \mathcal{R}_2 \mapsto (v_0, \ldots, v_i, \ldots, v_n)}{\sigma \vdash p^{\square}[\square.i] \times \mathcal{R}_1 \Downarrow \mathcal{R}_2.i \mapsto v_i}$$

P-DerefPtr
$$\frac{\sigma \vdash \square \times \mathcal{R}_1 \Downarrow \_ \mapsto \texttt{ptr}\ \pi \qquad \sigma \vdash p^{\square} \times \pi \Downarrow \mathcal{R}_2 \mapsto v}{\sigma \vdash p^{\square}[*\square] \times \mathcal{R}_1 \Downarrow \mathcal{R}_2 \mapsto v}$$

$\boxed{\Sigma \vdash (\sigma; \boxed{e}) \to (\sigma'; \boxed{e'})}$

E-Move
$$\frac{\sigma \vdash \pi \Downarrow \_ \mapsto v}{\Sigma \vdash (\sigma; \boxed{\pi}) \to (\sigma[\pi \mapsto \text{dead}]; \boxed{v})}$$

E-Copy
$$\frac{\sigma \vdash p \Downarrow \_ \mapsto v}{\Sigma \vdash (\sigma; \boxed{p}) \to (\sigma; \boxed{v})}$$

E-Borrow
$$\frac{\sigma \vdash p \Downarrow \mathcal{R} \mapsto \_}{\Sigma \vdash (\sigma; \boxed{\&r\ \omega\ p}) \to (\sigma; \boxed{\texttt{ptr}\ \mathcal{R}})}$$

E-Seq
$$\frac{}{\Sigma \vdash (\sigma; \boxed{v; e}) \to (\sigma; \boxed{e})}$$

E-LetProv
$$\frac{}{\Sigma \vdash (\sigma; \boxed{\texttt{letprov <}r\texttt{> \{ } v \texttt{ \}}}) \to (\sigma; \boxed{v})}$$

E-Assign
$$\frac{\sigma \vdash p \Downarrow \mathcal{V} \qquad p = p^{\square}[x]}{\Sigma \vdash (\sigma; \boxed{p := v}) \to (\sigma[x \mapsto \mathcal{V}[v]]; \boxed{()})}$$

E-Let
$$\frac{}{\Sigma \vdash (\sigma; \boxed{\texttt{let}\ x : \tau_a^{\text{SI}} = v; e}) \to (\sigma, x \mapsto v; \boxed{\texttt{shift}\ e})}$$

E-Shift
$$\frac{}{\Sigma \vdash (\sigma, x \mapsto v'; \boxed{\texttt{shift}\ v}) \to (\sigma; \boxed{v})}$$

E-Closure
$$\frac{\text{free-vars}(e) = \overline{x_f} \qquad \text{free-nc-vars}_{\sigma}(e) = \overline{x_{nc}} \qquad \varsigma_c = \sigma \mid_{\overline{x_f}}}{\Sigma \vdash (\sigma; \boxed{|x_1 : \tau_1^{\text{s}}, \ldots, x_n : \tau_n^{\text{s}}| \ \to\ \tau_r^{\text{s}}\ \{\ e\ \}}) \to (\sigma[\overline{x_{nc}} \mapsto \text{dead}]; \boxed{\langle \varsigma_c,\ |x_1 : \tau_1^{\text{s}}, \ldots, x_n : \tau_n^{\text{s}}| \ \to\ \tau_r^{\text{s}}\ \{\ e\ \} \rangle})}$$

E-AppClosure
$$\frac{v_f = \langle \varsigma_c,\ |x_1 : \tau_1^{\text{s}}, \ldots, x_n : \tau_n^{\text{s}}| \ \to\ \tau_r^{\text{s}}\ \{\ e\ \} \rangle}{\Sigma \vdash (\sigma; \boxed{v_f(v_1, \ldots, v_n)}) \to (\sigma \natural \varsigma_c, x_1 \mapsto v_1, \ldots, x_n \mapsto v_n; \boxed{\texttt{framed}\ e})}$$

E-Framed
$$\frac{}{\Sigma \vdash (\sigma \natural \varsigma; \boxed{\texttt{framed}\ v}) \to (\sigma; \boxed{v})}$$

E-While
$$\frac{}{\Sigma \vdash (\sigma; \boxed{\texttt{while}\ e_1\ \{\ e_2\ \}}) \to (\sigma; \boxed{\texttt{if}\ e_1\ \{\ e_2; \texttt{while}\ e_1\ \{\ e_2\ \}\ \}\ \texttt{else}\ \{\ ()\ \}})}$$

Fig. 7. Selected Place-expression Evaluation Rules (top) and Reduction Rules (bottom)

our rules rely on a notion of place-expression evaluation, $\sigma \vdash p \Downarrow \mathcal{R} \mapsto v$ (Figure 7, top), which should be read as: p computes to $\mathcal{R}$, which maps to v in σ.

The evaluation rules are straightforward: E-Move returns a value by moving it off of the stack σ, replacing it with dead. E-Copy copies the value from the stack. E-Borrow creates a pointer value to the referent $\mathcal{R}$. Branching is completely standard, hence elided. Assignment, similar to E-Copy and E-Borrow, uses a place-expression evaluation rule but a slightly different "get-context" version, $\sigma \vdash p \Downarrow \mathcal{V}$ (elided), which just returns the context $\mathcal{V}$ surrounding the value at our desired address as it sits on the stack bound to a variable. Then, assignment updates the stack by maintaining this context when it updates x (mapping it to $\mathcal{V}[v]$).

Binding and the Stack. Bindings are interesting in that they introduce our two administrative forms, `framed` e and `shift` e. For instance, in E-Let, we step to `shift` e rather than e alone in order to ensure that the binding for x is well-scoped and ends when it should (seen in E-Shift). In E-AppClosure, we similarly step to `framed` e to ensure that after evaluating the body of the closure

we drop the stack frame from that function call (seen in E-Framed). Both E-Shift and E-Framed rely crucially on the fact that our stack σ is ordered — they must match the most recent entry.

3.4 Well-typed Oxide programs won't go wrong!

We prove syntactic type safety for Oxide using progress and preservation [Wright and Felleisen 1992]. The proofs of these lemmas are fairly standard — using structural induction on the typing derivation in both cases — but rely on additional formal machinery to address the fact that evaluation can make provenances more precise as it, for instance, follows one particular side of a branch.

Lemma 3.1 (Progress). *If* $\Sigma; \bullet; \Gamma \vdash \boxed{e} : \tau^{SI} \Rightarrow \Gamma'$ *and* $\Sigma \vdash \sigma : \Gamma$ *, then either e is a value, e is an* `abort!` *(…) , or* $\exists\, \sigma', e'.\ \Sigma \vdash (\sigma; \boxed{e}) \rightarrow (\sigma'; \boxed{e'})$.

The Progress lemma says that if we can type-check e under a valid global environment Σ and stack typing Γ *and* we have a stack σ that satisfies this stack typing Γ, then either e is a value, an `abort!` expression, or we can take a step. We've elided the $\Sigma \vdash \sigma : \Gamma$ judgment as it's straightforward. The proof proceeds by structural induction on the typing derivation for e, and relies on lemmas that tell us that we can find values for places and place expressions at runtime when our σ is well-formed.

Lemma 3.2 (Preservation). *If* $\Sigma; \bullet; \Gamma \vdash \boxed{e} : \tau_1^{SI} \Rightarrow \Gamma_f$ *and* $\Sigma \vdash \sigma : \Gamma$ *and* $\Sigma \vdash (\sigma; \boxed{e}) \rightarrow (\sigma'; \boxed{e'})$ *, then there exists* Γ_i *such that* $\Sigma \vdash \sigma' : \Gamma_i$ *and* $\Sigma; \bullet; \Gamma_i \vdash \boxed{e'} : \tau_2^{SI} \Rightarrow \Gamma_f'$ *and* $\bullet;\ \Gamma_f' \vdash \tau_2^{SI} \lesssim \tau_1^{SI} \Rightarrow \Gamma_s$ *and there exists* Γ_o *such that* $\Gamma_f = \Gamma_s \Cup \Gamma_o$.

The Preservation lemma says that if e has type τ_1^{SI} under a valid global environment Σ and stack typing Γ, have a stack σ that satisfies the stack typing Γ, and can take a step to an updated configuration $(\sigma'; \boxed{e'})$, then there exists some intermediate stack typing Γ_i that our updated stack σ' satisfies and under which the expression e' type-checks with a potentially more-specific type τ_2^{SI} and output stack typing Γ_f'.

With Lemma 3.1 and Lemma 3.2 in hand, we can prove the following type safety theorem (Theorem 3.3) by interleaving the usage of progress and preservation. Full proofs of these and all supporting lemmas are included in our technical appendix.

Theorem 3.3 (Type Safety). *If* $\Sigma; \bullet; \bullet \vdash \boxed{e} : \tau^{SI} \Rightarrow \Gamma$ *and* $\vdash \Sigma$ *then,* $\Sigma \vdash (\bullet; \boxed{e}) \rightarrow^* (\sigma'; \boxed{v})$ *or the evaluation of e steps to an abort expression or otherwise diverges.*

Notice that Type Safety, Progress, and Preservation all restrict their attention to expressions with sized & initialized types τ^{SI}. This is because expressions only ever have sized & initialized types, while values of unsized or dead types only exist as part of the stack and other machinery.

4 (IRON) OXIDE IS RUST

To show that Oxide is a faithful formal model for the (core) Rust borrow checker, we work through a number of example programs in Rust, and their corresponding form in Oxide. Then, in §4.3, we describe a prototype type-checker for Oxide and how we've used it to test our semantics against the official borrow checker. Finally, in §4.4, we draw connections between Oxide and Polonius, a new streamlined implementation of Rust's borrow checker using techniques from logic programming.

4.1 Liveness

One of the primary goals of Rust's borrow checker is to statically ensure that there are no use-after-free errors for references since they are a common class of bugs and even security vulnerabilities when doing systems programming in C. To see how it works, we'll look at a small example:

```
1  let msg = {
2    let m = ("Howdy", "Pals");
3    &m.0 // ERROR: m.0 does not live long enough
4  };
5  msg
```

In the block spanning lines 1–4, we declare a tuple of one element on line 2, and then create a reference to it on line 3. Since Rust is largely an expression-based (rather than statement-based) language, when we evalute this block, it will return the value we get from `&m.0`. However, after doing so, `m` drops out of scope, and since it is on the stack, it is then necessarily destroyed. If this program was allowed, we would then have a dead pointer *forward* on the stack, which would be very bad. Fortunately, Rust's borrow checker detects this, and instead reports an error — protecting us from our mistake! Let's look at how the same program would work in Oxide:

```
1  // Γ0 = •
2  letprov<'msg, 'm> {
3    // Γ1 = 'msg ↦ {}, 'm ↦ {}
4    let msg: &'msg shrd String = {
5      // Γ2 = Γ1
6      let m: (String, String) = ("Howdy", "Pals");
7      // Γ3 = Γ2, m : (String, String)
8      &'m shrd m.0 // ERROR. τ = &'m shrd String
9      // Γ4 = 'msg ↦ {}, 'm ↦ { shrd m.0 }
10   }; // 'm is no longer valid at this point.
11   msg
12 }
```

To translate to Oxide, we again made the usual set of changes — annotating bindings with types, and adding explicit `letprov` and `shrd` qualifiers. To aid comprehension, we also added comments that describe the state of the stack typing Γ while type-checking the program. Like the Rust version, the Oxide version statically produces an error, but to understand why we must cover a few facts. First, recall that our rule for let binding (T-Let) removes bound variables from the environments at the end of the binding (seen on line 9). Further, note that the type we derive for `&'m shrd` $m.0$ has provenance `'m` mapped to { shrd $m.0$ }. Then, since type well-formedness requires that the places present in the loan sets for each provenance be bound, we are unable to prove that the type of the expression being bound for `msg` is valid. That is, •; •; $\Gamma_4 \vdash$ `&'m shrd String` does not hold.

4.2 Conditional Control Flow

It is also important for the borrow checker to be able to deal appropriately with conditional control flow. As mentioned in §2.3, it is essential to treat conditional loans as live in order to have a sound analysis. To see how Oxide handles conditional control flow, we will look at two examples in Rust and Oxide— one that type-checks and one that does not. We'll start with the former:

```
1  struct Point(u32, u32);
2  let mut pt: Point = Point(3, 2);
3  if cond {
4      let x = &mut pt.0;
5      *x = 4;
6  } else {
7      let p = &mut pt;
8      (*p).1 = 5;
9  }
```

In Rust, we declare a mutable binding `pt` to a `Point` value. Then, we branch on an unknown boolean variable `cond`, and in one case uniquely borrow the first projection of `pt` before assigning it a new value. In the other case, we uniquely borrow the whole of `pt`, and then mutate its second projection through this reference. Since Rust identifies that only one of these will happen in any program, it is okay for the two unique references to refer to overlapping parts of memory. The program is largely the same in Oxide (though we have again included comments marking the state of the environments during type-checking):

```
1  struct Point(u32, u32);
2  // Γ0 = •
3  letprov<'a, 'b> {
4    // Γ1 = 'a ↦ {}, 'b ↦ {}
5    let pt: Point = Point(3, 2);
6    // Γ2 = Γ1, pt : Point
7    if cond { // Γ3 = Γ2
8        let x: &'a uniq u32 = &'a uniq pt.0;
9        // Γ4 = 'a ↦ { uniq pt.0 }, 'b ↦ {}, pt : Point, x : &'a uniq u32
10       *x = 4;
11       // Γ5 = Γ4
12       () // Γ6 = Γ2
13   } else { // Γ7 = Γ2
14       let p: &'b uniq Point = &'b uniq pt;
15       // Γ8 = 'a ↦ {}, 'b ↦ { uniq pt }, pt : Point, p : &'b uniq Point
16       (*p).1 = 5;
17       // Γ9 = Γ8
18       () // Γ10 = Γ2
19   } // Γ11 = (Γ10 ∪ Γ6) = Γ2
20 }
```

As usual, **`mut`** has been replaced with the more appropriate `uniq`. We can now see more formally how this example type-checks. In particular, when we get to the branch on line 7, according to T-Branch, we check the type of each side of the branch under the same environment Γ_2 (visible in the annotations on lines 8 and 16). Since they have the same input environment, they are each able to create their own unique reference to parts of `pt` (lines 9 and 17). Then, the bindings to `x` and `p` both end at the end of their respective branch before returning unit (lines 13 and 21). This means that when we union the *output* environments of each branch on line 21, we get exactly their input environments Γ_2, meaning that the loans in each branch have necessarily ended.

However, it's also possible for loans to outlive the scope they are created in. We will explore this kind of situation in our next example:

```
let mut m: u32 = 6;
let mut n: u32 = 5;
let x: &u32 = &n;
if false {
  x = &m;
}
&mut m; // ERROR: cannot borrow m mutably while already borrowed
... // additional code using x
```

In this example, we declare two mutable bindings `m` and `n` on lines 1 and 2. Then, on line 3, we create a shared reference to `n` and bind it to `x`. On line 4, we branch, and assign to `x` a shared reference to `m` instead. Then, after the branch ends, we try to mutably borrow `m`. Even though we

can see that the branch is dead code (since the condition is always **false**), the borrow checker will not inspect the value and will instead give us an error saying that we cannot borrow `m` mutably twice. The program is again similar in Oxide (and again annotated with environment Γ):

```
1  letprov<'a, 'b, 'c> {
2    // Γ0 = 'a ↦ {}, 'b ↦ {}, 'c ↦ {}
3    let mut m: u32 = 6;
4    // Γ1 = Γ0, m : u32
5    let mut n: u32 = 5;
6    // Γ2 = Γ1, n : u32
7    let x: &'a shrd u32 = &'a shrd n;
8    // Γ3 = 'a ↦ { shrd n }, 'b ↦ {}, 'c ↦ {}, m : u32, n : u32, x : &'a shrd u32
9    if false { // Γ4 = Γ3
10     x := &'b shrd m;
11     // Γ5 = 'a ↦ { shrd m }, 'b ↦ { shrd m }, 'c ↦ {}, m : u32, n : u32, x : &'a shrd u32
12     () // Γ6 = Γ5
13   } else { // Γ7 = Γ3
14     () // Γ8 = Γ7
15   } // Γ9 = Γ6 ⊍ Γ8 =
16     // 'a ↦ { shrd m, shrd n }, 'b ↦ { shrd m }, 'c ↦ {}, m : u32, n : u32, x : &'a shrd u32
17   &'c uniq m; // ERROR: cannot borrow m uniquely while already borrowed
18   ... // additional code using x
19 }
```

Now, using the Oxide version of the example, we can explain more formally why the program fails to type-check. On line 7, when we borrow from `n`, we produce a reference of type `&'a shrd u32` and add it to our stack typing as the type of `x` (line 7). Then, in the first half of the branch, we assign to `x` a shared reference to `m` (line 11). According to OL-OverrideLocalProvenances (via T-Assign), this will cause us to replace the loans associated with `'a` with the loans associated with `'b` (namely { shrd m }), but in the other side of the branch, we don't change `'a` and so it remains the same (lines 12 and 18 respectively). When we exit the branch, in T-Branch, we will combine the two stack typings from each side, resulting in the unification of the loan sets associated with each provenance in each side of the branch. The result (as seen on line 19) is that `'a` is { shrd m , shrd n }. Thus, when we attempt to derive ownership safety in T-Borrow for the borrow expression on line 21, we find an overlapping shared loan against `m` in the loan set for `'a` and yield an error.

passing		disqualified						
borrowck	nll	heap	out-of-scope library	enums	statics & consts	traits	uninitialized variables	misc.
89	119	63	40	50	40	93	40	81

Fig. 8. Tested Semantics Results

4.3 Tested Semantics

We set out at the onset to solve a particular problem — there is no high-level specification of the Rust programming language and its borrowchecker. If there were, this would be the point where we might present a proof that every expression that type checks in Oxide also type checks in Rust and vice versa. Since doing that is not possible, we follow Guha et al. [2010] in developing a *tested semantics* of Oxide typechecking. We have built an implementation of our Oxide type-checking algorithm, OxideTC, alongside a compiler, called Reducer, from a subset of Rust (with a small number of additional annotations) to Oxide. In addition to the features described in §3, our implementation supports Rust's structs by treating them as tagged tuples or records. The combined

Reducer-OxideTC tooling has allowed us to use tests from the official borrow checker (`borrowck`) and non-lexical lifetime (`nll`) test suites to validate Oxide's faithfulness as a model of Rust against its implementation, rustc. The results of this testing is summarized in Figure 8.

For the 208 passing tests, we can compile the test case into Oxide with Reducer and then use OxideTC to either successfully type check the program or to produce a type error. We compare this type checking result to the expected behavior according to the rustc test suite. All 208 tests either type check when rustc does so, or produce an error corresponding to the error produced by rustc.

The remaining 407 tests were taken out of consideration on the basis of being out-of-scope for this work. There were 20 categories for exclusion, the majority of which had fewer than 10 applicable tests. Figure 8 includes the 6 largest categories: (1) heap allocation, (2) out-of-scope libraries, (3) enumerations, (4) statics and constants, (5) traits, and (6) uninitialized variables. One specialized category (multithreading) was folded into out-of-scope libraries in this table, with the miscellaneous column aggregating the remaining smaller categories: control flow, casting, first-class constructors, compiler internals dumping, function mutability, inline assembly, macros, slice patterns, two-phase borrows (discussed in §6.2), uninitialized variables, universal function-call syntax, unsafe, and variable mutability (discussed in §2.3).

Combined, heap allocation and out-of-scope libraries (of which the former is a specialization of the latter) make up for the largest excluded category with 103 tests, and is the most immediate avenue for future work as we will discuss in §6.1. The next largest category, traits, accounts for 93 tests. Though the trait system is in some ways novel, the bulk of its design is rooted in the work on Haskell typeclasses and their extensions. As such, we feel that they are not an *essential* part of Rust, though exploring the particularities of their design may be a fruitful avenue for future work on typeclasses. We are working on extending our implementation with sums to support enumerations. Many of the other categories describe features (e.g., macros, control flow, casting, first-class constructors, statics, and constants) that are well-studied in the programming languages literature, and in which we believe Rust has made relatively standard design choices.

The last issue to discuss involving the tested semantics is the aforementioned annotation burden. This burden comes directly out of the syntactic differences between Oxide and Rust as seen in §3.1, and so are fairly minor. The most immediately apparent need is to provide a provenance annotation on borrow expressions, which we handle using Rust's compiler annotation support. In our tests, a borrow expression like `&'a uniq x` appears as `#[lft="a"] &mut x`. However, we reduce the need for this by automatically generating a fresh local provenance for borrow expressions without an annotation. This suffices for the majority of expressions without change. Relatedly, one might also expect to see the introduction of `letprov` throughout. To alleviate the need for this, our implementation automatically binds free provenances at the beginning of each function body.

The other main change we had to make relates to the use of explicit environment polymorphism in Oxide. In Rust, every closure has a unique type without a syntax for writing it down. To work with higher-order functions, these closures implement one of three special language-defined traits (`Fn`, `FnMut`, and `FnOnce`) which can be used as bounds in higher-order functions. We compile the use of these trait bounds to environment polymorphism in a straight-forward manner (turning instances of the same `Fn`-bound polymorphic type into uses of function types with the same environment variable), but need to introduce a way of writing down which environment to use at instantiation time. We use a compiler annotation (`#[envs(c1, ..., cn)]`) on applications which says to instantiate the environment variables with the captured environments of the types of these bindings. If the bindings are unbound or not at a function type, we produce an error indicating as much.

Aside from these two changes, there are a handful of smaller changes that we made by hand to keep the implementations of Reducer and OxideTC simpler, though the need for these could be obviated with more work. Our implementation does not support method call syntax, and so

we translate method definitions (which take `self`, `&self`, or `&mut self` as their first argument) into ordinary function definitions with a named first argument at the method receiver's type. Relatedly, some of the tests used traits in a trivial way to define methods polymorphic in their receiver type. Much as with other methods, we translated these into ordinary function definitions and used a polymorphic type for the receiver. Further, RUSTC allows for a number of convenient programming patterns (like borrowing from a constant, e.g. `&0`) which are not supported by our implementation. To deal with these cases, we manually introduced temporaries (a process that RUSTC does automatically). As a simplification for the type checker, OXIDETC only reports the first error that occurs in the program. To ensure that we find a correspondence between all errors, we split up test files with multiple errors into one file per test.

Finally, an earlier version of our implementation required type annotations on all let bindings, and so currently the majority of tests include fully-annotated types. We later came to the realization that our typing judgment is very-nearly a type *synthesis* judgment as in bidirectional typechecking, and so the implementation now supports unannotated let bindings by giving the name the type synthesized from the expression being bound. This works for all expressions except `abort!` which can produce any type and thus requires an annotation. Further, if the programmer wishes to give the binding a broader type via subtyping, they must provide it with an annotated type.

4.4 Polonius

Polonius [Matsakis 2018] is a new alias-based formulation of Rust's borrow checker that uses information from the Rust compiler as input facts for a logic program that checks the safety of borrows in a program. Much as we have done with Oxide, Polonius shifts the view of *lifetimes* to a model of *regions* as sets of loans. Similar to Oxide's provenances, Polonius' regions are a mechanism for approximating the possible provenances of a given reference, and as described by Matsakis [2018], a reference is no longer valid when any of the region's constituent loans are invalidated. In Oxide, we take an analogous view: a reference type is valid only when its constituent loans are bound in the stack typing Γ. Though we have not formally explored the connection, based on the commonality between both new views on lifetimes, we feel that Oxide corresponds to a sort of type-systems formulation of Polonius.

5 RELATED WORK

5.1 Semantics for Rust

Patina. Reed [2015] developed *Patina*, a formal semantics for an early version of Rust (pre-1.0) focused on proving memory safety for a language with a syntactic version of borrow checking and unique pointers. Unfortunately, the design of the language was not yet stable, and the language overall has drifted from their model. Also, unlike Oxide, Patina made concrete decisions about memory layout and validity which is problematic as Rust itself has not yet made such commitments.

Rusty Types. Benitez [2016] developed *Metal*, a formal calculus that, by their characterization, has a Rust-like type system using an *algorithmic* borrow-checking formulation. Their model relies on capabilities as in the Capability Calculus of Crary et al. [1999], but manages them indirectly (compared to the first-class capabilities of Crary et al. [1999] or Morrisett et al. [2007]). Compared to Rust and our work on Oxide, Metal is unable to deal with the proper LIFO ordering for object destruction and their algorithmic formulation is less expressive than our declarative formulation.

RustBelt. In the RustBelt project, Jung et al. [2018] developed a formal semantics called λ_{Rust} for a continuation-passing style intermediate language in the Rust compiler known as MIR. They

mechanized this formal semantics in Iris [Jung et al. 2017] and used it to verify the extrinsic safety of important Rust standard library abstractions that make extensive use of `unsafe` code. Their goal was distinct from ours in that we instead wish to reason about how programs work at the source-level, and our goals are fortunately complementary. While we argue in Sec. 6.1 that we can treat `unsafe` code in the standard library as an implementation detail of the language, the work by Jung et al. on RustBelt provides further justification by allowing us to say that what we model as primitives can be compiled to their verified MIR implementations.

5.2 Practical Substructural Programming

As a practical programming language with substructural typing, Rust does not exist in a vacuum. There have been numerous efforts in the programming languages community to produce languages that rely on substructurality. Though different in their design from Rust, these languages sit in the same broader design space, finding a balance between usability and expressivity.

Mezzo. Pottier and Protzenko [2013] developed Mezzo, an ML-family language with a static discipline of duplicable and affine permissions to control aliasing and ownership. Similar to Rust, Mezzo is able to have types refer directly to values, rather than always requiring indirection as in work on ownership types [Clarke et al. 1998; Noble et al. 1998]. However, unlike Rust, Mezzo uses a permissions system that works as a sort of type-system formulation of separation logic [Reynolds 2002]. By contrast, Rust relies on a borrow checking analysis to ensure that its guarantees about aliasing and ownership are maintained. In Oxide, we formalized this analysis as the ownership safety judgment which determines if it is safe to use a place uniquely or sharedly in a given context.

Alms. Tov and Pucella [2011] developed Alms as an effort to make affine types *practical* for programming. Unlike Rust, Alms more closely follows the ML tradition, and relies on an interesting module system to design resource-aware abstractions. Within Alms module signatures, the programmer can annotate abstract types with kinds that denote whether or not they should be affine. They use abstract affine types in modules to build explicit capabilities into the function signatures within the module which enforce correct use. By contrast, in Rust, everything is affine and unrestricted types are approximated through the use of the `Copy` trait.

Resource Polymorphism for OCaml. Munch-Maccagnoni [2018] has recently proposed a backwards-compatible model of resource management for OCaml. Though not yet a part of OCaml, the proposal is promising and aims to integrate ideas from Rust and C++ (like ownership and so-called "resource acquisition is initialization" [Stroustrup 1994]) with a garbage-collected runtime system for a functional language. Similar to our efforts in understanding Rust, they note the relationship that Baker's work on Linear Lisp [Baker 1994a,b, 1995] has to modern efforts for practical substructural programming. As Munch-Maccagnoni note themselves, there is much to be learned from Rust in these kinds of efforts, and we hope that Oxide provides a stronger footing for doing so.

Cyclone. Grossman et al. [2002] developed Cyclone, whose goal was to be a safe C alternative. To do so, they rely on techniques from region-based memory management [Tofte and Talpin 1994, 1997]. However unlike Rust, regions in Cyclone indicate where an object is in memory (for example, if it is on the stack or the heap). As noted early on in §2.2, the meaning of regions in Rust (and Oxide) is different. Provenances correspond to static approximations of a reference's possible origins, without requiring any realization to a particular memory model. Similar to our effort to develop Oxide, Grossman et al. [2002] and Fluet et al. [2006] developed formal semantics to build an understanding of the essence of Cyclone.

$$\begin{array}{llll} \text{Sized Types} & \tau^{\text{SI}} & ::= & \ldots \mid \texttt{Vec<}\tau\texttt{>} \\ \text{Expressions} & e & ::= & \ldots \mid \texttt{Vec::<}\tau\texttt{>::new()} \mid e_1\texttt{.push(}\hat{e}_2\texttt{)} \mid e\texttt{.pop()} \mid e_1\texttt{.swap(}\hat{e}_2, \hat{e}_3\texttt{)} \mid e\texttt{.len()} \end{array}$$

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

T-VecNew

$$\frac{}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\texttt{Vec::<}\tau\texttt{>::new()}} : \texttt{Vec<}\tau\texttt{>} \Rightarrow \Gamma}$$

T-VecPush

$$\frac{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e_1} : \&\rho\ \texttt{uniq Vec<}\tau\texttt{>} \Rightarrow \Gamma_1 \qquad \Sigma;\ \Delta;\ \Gamma_1 \vdash \boxed{\hat{e}_2} : \tau \Rightarrow \Gamma_2}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e_1\texttt{.push(}\hat{e}_2\texttt{)}} : \texttt{unit} \Rightarrow \Gamma_2}$$

T-VecPop

$$\frac{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e} : \&\rho\ \texttt{uniq Vec<}\tau\texttt{>} \Rightarrow \Gamma'}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e\texttt{.pop()}} : \tau \Rightarrow \Gamma'}$$

T-VecLen

$$\frac{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e} : \&\rho\ \texttt{shrd Vec<}\tau\texttt{>} \Rightarrow \Gamma'}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e\texttt{.len()}} : \texttt{u32} \Rightarrow \Gamma'}$$

T-VecSwap

$$\frac{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e_1} : \&\rho\ \texttt{uniq Vec<}\tau\texttt{>} \Rightarrow \Gamma_1 \qquad \Sigma;\ \Delta;\ \Gamma_1 \vdash \boxed{\hat{e}_2} : \texttt{u32} \Rightarrow \mathscr{L}_2\Gamma_2 \qquad \Sigma;\ \Delta;\ \Gamma_2 \vdash \boxed{\hat{e}_3} : \texttt{u32} \Rightarrow \Gamma_3}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e_1\texttt{.swap(}\hat{e}_2, \hat{e}_3\texttt{)}} : \texttt{unit} \Rightarrow \Gamma_3}$$

Fig. 9. Extending Oxide with Vectors

6 DISCUSSION AND FUTURE WORK

6.1 A Tower of Languages

Following the proposal by Weiss et al. [2018], we take the view that, although Rust's standard library contains a great deal of **`unsafe`** code, this reliance on **`unsafe`** is ultimately an *implementation detail* of the language. In many other languages, key data structures like hash maps are implemented as built-in types within the interpeter or compiler. In Rust's case, `HashMap` happens to be implemented using **`unsafe`** code, but it is no less safe than such built-ins. Bugs within this code are taken seriously as the library is relied upon by millions of lines of code. Instead, what is essential to the soundness of Rust overall is that the API that these standard library abstractions present are safe at the types they are given. To that end, we wish to build on Oxide with extensions for individual abstractions that ultimately increase the *expressive power* [Felleisen 1991] of the language.

Following Matsakis [2016b] and Weiss et al. [2018], we consider the most important of these abstractions to be `Vec`, the type of dynamically-sized vectors (which adds support for heap allocation), `Rc`, the type of reference-counted pointers (which adds support for runtime-checked sharing), and `RefCell`, the type of mutable reference cells (which adds support for runtime-guarded mutation). Though these extensions are beyond the scope of this paper, we show a sketch of an extension for heap allocation in Figure 9, which adds support for `Vec` to Oxide. We leave the full extensions and their metatheory to future work.

The extension comes in a few parts. First, we extend the grammar of types to include a polymorphic vector type `Vec<`τ`>`. Then, we extend the grammar of expressions with some of the key operations on vectors. `Vec::<`τ`>::new()` is used to create a new empty vector with the element type τ. Then, e_1`.push(`e_2`)` and e`.pop()` are used to add and remove elements from the vector, while e_1`.swap(`e_2, e_3`)` is used to swap the values in the vector at indices e_2 and e_3. Finally, of course, e`.len()` yields the current number of elements stored within the vector. Notably, the typing rules in our sketch directly follow the types as defined in Rust's `Vec` API, suggesting that they are essentially special cases of Oxide's rule for function application (T-App).

6.2 Two-Phase Borrows

In working on non-lexical lifetimes, Matsakis [2017] introduced a proposal for two-phase mutable borrows in Rust. The goal of these two-phase borrows is to resolve a long-standing usability issue,

referred to as the "nested method call" problem, where Rust's borrow checker might force the programmer to introduce temporaries to prove that code like `vec.push(vec.len())` is safe. To understand where the problem comes from, we will have to look at how method calls expand in Rust. For example, `vec.push(vec.len())` desugars to the code on the left below:

```
1  let tmp0 = &mut vec;
2  let tmp1 = &vec;
3  let tmp2 = Vec::len(tmp1);
4  Vec::push(tmp0, tmp2);
```

```
1  let tmp1 = &vec;
2  let tmp2 = Vec::len(tmp1);
3  let tmp0 = &mut vec;
4  Vec::push(tmp0, tmp2);
```

Without two-phase borrows, the example on the left behaves like one of our early examples in §2.2. That is, we cannot create an immutable reference on line 2 because the mutable loan from line 1 is still live. Further, non-lexical lifetimes are no help — the loan on line 1 *needs* to be live until line 4. However, intuitively, we know that this code is safe since the mutable loan is not *actually* needed until line 4. We could resolve the problem here by desugaring to the code on the right.

Unfortunately, this desugaring in general is subtle. Still, the idea of reordering suggests a weakening of the type system to make the two expansions equivalent to the borrowchecker. This weakening is precisely two-phase borrows. When a mutable borrow occurs for a method receiver, the loan is marked *reserved*. Reserved loans then act as if they are shared *until* the method is applied.

While Oxide does not currently support two-phase borrows, we could imagine extending our grammar for ownership quantifiers ω with a new form `reserved`, which behaves precisely like a `shrd`-loan until the program requires uniqueness at which point it is raised to a `uniq`-loan. However, this would likely require some additional machinery in order for the ownership safety judgment to make these transitions at the use site of values with `reserved` loans, complicating our type system.

6.3 A Rusty Future

Oxide gives a formal framework for reasoning about the behavior of source-level Rust programs. This reasoning opens up a number of promising avenues for future work on Rust using Oxide.

Mechanized Metatheory for Oxide. Though we have paper proofs in our technical appendix (§3.4) for all the theorems presented here, we have begun an effort to mechanize the semantics in Coq. This has a number of advantages. First, as with most efforts for mechanized metatheory, we can establish even more confidence in our current results. Further, we can expand the mechanization to incorporate other important theorems. Finally, other researchers can use the mechanization as a starting point for their work and know that their changes have not violated type safety.

Formal Verification. One of the unfortunate gaps in Rust programming today is the lack of effective tools for proving properties (such as functional correctness) of Rust programs. There are some early efforts already to try to improve this situation [Astrauskas et al. 2018; Baranowski et al. 2018; Toman et al. 2015; Ullrich 2016], but without a semantics the possibilities are limited. For example, the work by Astrauskas et al. [2018] builds verification support for Rust into Viper [Müller et al. 2016], but uses an ad-hoc subset without support for shared references. We believe that our work on Oxide can help extend such work and will enable further verification techniques like those seen in $F^\star$ [Swamy et al. 2016] and Liquid Haskell [Vazou et al. 2014].

Verified Compilation. Rust's memory safety guarantees lend themselves well to security-critical applications. However, the existing compiler toolchain (leveraging LLVM [Lattner and Adve 2004]) does not lend itself well to preserving these kinds of guarantees. As such, another avenue for future work using Oxide would be to build an alternative verified compiler toolchain, perhaps by compilation to Vellvm [Zhao et al. 2012] or CompCert's Clight [Blazy and Leroy 2009].

Security. We also view Oxide as an enabler for future work on extending techniques from the literature on language-based security to Rust. In particular, one could imagine building support for dynamic or static information-flow control atop Oxide as a formalization (for which we can actually prove theorems about these extensions) alongside a practical implementation for the official Rust compiler. Further, we would like to prove parametricity for Oxide to develop support for relaxed noninterference through type abstraction as done in recent work by Cruz et al. [2017].

7 CONCLUSION

We have presented Oxide, a formal model of *the essence of Rust*. Oxide features a novel presentation of ownership and borrowing from the perspective of Rust, and reformulates Rust's algorithmic borrow-checker as a declarative substructural type system. We proved type safety for Oxide using syntactic techniques (§3.4). We implemented the Oxide type checker in OCaml along with a compiler from Rust to Oxide, and validated our semantics against a suite of over two-hundred tests from the official RUSTC test suite. As alluded to in Sections 1 and 6, we hope Oxide will serve as a basis for further research using Rust, and more broadly on safe and correct systems programming.

REFERENCES

Amal Ahmed. 2004. *Semantics of Types for Mutable State.* Ph.D. Dissertation. Princeton University.

Amal Ahmed, Andrew W. Appel, Christopher D. Richards, Kedar N. Swadi, Gang Tan, and Daniel C. Wang. 2010. Semantic Foundations for Typed Assembly Languages. *ACM Transactions on Programming Languages and Systems* 32, 3 (March 2010), 1–67.

Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 2018. *Leveraging Rust Types for Modular Specification and Verification.* Technical Report. Eidgenössische Technische Hochschule Zürich.

Henry G. Baker. 1992. Lively Linear Lisp — 'Look Ma, No Garbage!'. *SIGPLAN Notices* (1992).

Henry G. Baker. 1994a. Linear Logic and Permutation Stacks—The Forth Shall Be First. *SIGARCH Computer Architecture News* (1994).

Henry G. Baker. 1994b. Minimizing Reference Count Updating with Deferred Anchored Pointers for Functional Data Structures. *SIGPLAN Notices* (1994).

Henry G. Baker. 1995. 'Use-Once' Variables and Linear Objects — Storage Management, Reflection, and Multi-Threading. *SIGPLAN Notices* (1995).

Marek Baranowski, Shaobo He, and Zvonimir Rakamarić. 2018. Verifying Rust Programs with SMACK. In *Automated Technology for Verification and Analysis.*

Sergio Benitez. 2016. Short Paper: Rusty Types for Solid Safety. In *Workshop on Programming Languages and Analysis for Security.*

Sandrine Blazy and Xavier Leroy. 2009. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning* 43, 3 (2009).

David G. Clarke, John M. Potter, and James Noble. 1998. Ownership Types for Flexible Alias Protection. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA).*

Karl Crary, David Walker, and Greg Morrisett. 1999. Typed Memory Management in a Calculus of Capabilities. In *ACM Symposium on Principles of Programming Languages (POPL), San Antonio, Texas.*

Raimil Cruz, Tamara Rezk, Bernard Serpette, and Éric Tanter. 2017. Type Abstraction for Relaxed Noninterference. In *European Conference on Object-Oriented Programming (ECOOP).*

Matthias Felleisen. 1991. On the expressive power of programming languages. *Science of Computer Programming* (1991).

Matthias Felleisen and Robert Hieb. 1992. The Revised Report on the Syntactic Theories of Sequential Control and State. *Theoretical Computer Science* (1992).

Matthew Fluet, Greg Morrisett, and Amal Ahmed. 2006. Linear Regions Are All You Need. In *European Symposium on Programming (ESOP).*

Jean-Yves Girard. 1987. Linear Logic. *Theoretical Computer Science* (1987).

Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. 2002. Region-Based Memory Management in Cyclone. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany.*

Unsafe Code Guidelines Working Group. 2019. Unsafe Code Guidelines. https://github.com/rust-rfcs/unsafe-code-guidelines. Accessed: 2019-02-22.

Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. 2010. The Essence of JavaScript. In *European Conference on Object-Oriented Programming (ECOOP)*.

Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems* (2001).

Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2019. Stacked Borrows: An Aliasing Model for Rust. *Proc. ACM Program. Lang.* 4, POPL, Article 41 (Dec. 2019), 32 pages. https://doi.org/10.1145/3371109

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: Securing the Foundations of the Rust Programming Language. In *ACM Symposium on Principles of Programming Languages (POPL), Los Angeles, California*.

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2017. Iris from the Ground Up: A Modular Foundation for Higher-Order Concurrent Separation Logic. In *Journal of Functional Programming*.

Yves Lafont. 1988. The Linear Abstract Machine. *Theoretical Computer Science* (1988).

Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society, Washington, DC, USA. http://dl.acm.org/citation.cfm?id=977395.977673

Nicholas D. Matsakis. 2016a. Non-lexical lifetimes: introduction. http://smallcultfollowing.com/babysteps/blog/2016/04/27/non-lexical-lifetimes-introduction/. Accessed: 2019-02-28.

Nicholas D. Matsakis. 2016b. Observational equivalence and unsafe code. http://smallcultfollowing.com/babysteps/blog/2016/10/02/observational-equivalence-and-unsafe-code/. Accessed: 2019-02-20.

Nicholas D. Matsakis. 2017. Nested method calls via two-phase borrowing. http://smallcultfollowing.com/babysteps/blog/2017/03/01/nested-method-calls-via-two-phase-borrowing/. Accessed: 2019-02-18.

Nicholas D. Matsakis. 2018. An alias-based formulation of the borrow checker. http://smallcultfollowing.com/babysteps/blog/2018/04/27/an-alias-based-formulation-of-the-borrow-checker/.

Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. System Sci.* (1978).

Naftaly Minsky. 1996. Towards Alias-Free Pointers. In *European Conference on Object-Oriented Programming (ECOOP)*.

Greg Morrisett, Amal Ahmed, and Matthew Fluet. 2007. L3: A Linear Language with Locations. *Fundamenta Informaticae* (2007).

Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*.

Guillaume Munch-Maccagnoni. 2018. Resource Polymorphism. *CoRR* abs/1803.02796 (2018). arXiv:1803.02796 http://arxiv.org/abs/1803.02796

James Noble, Jan Vitek, and John Potter. 1998. Flexible Alias Protection. In *European Conference on Object-Oriented Programming (ECOOP)*.

François Pottier and Jonathan Protzenko. 2013. Programming with Permissions in Mezzo. In *International Conference on Functional Programming (ICFP), Boston, Massachusetts*.

Eric Reed. 2015. *Patina: A formalization of the Rust programming language.* Master's thesis. University of Washington.

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *IEEE Symposium on Logic in Computer Science (LICS), Copenhagen, Denmark*.

Bjarne Stroustrup. 1994. *The Design and Evolution of C++*. Addison-Wesley.

Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. 2016. Dependent Types and Multi-monadic Effects in F*. In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg, Florida*.

Mads Tofte and Jean-Pierre Talpin. 1994. Implementation of the Typed Call-by-Value λ-calculus using a Stack of Regions. In *ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon*.

Mads Tofte and Jean-Pierre Talpin. 1997. Region-Based Memory Management. *Information and Computation* (1997).

John Toman, Stuart Pernsteiner, and Emina Torlak. 2015. CRust: A Bounded Verifier for Rust. In *IEEE/ACM International Conference on Automated Software Engineering*.

Jesse A. Tov and Riccardo Pucella. 2011. Practical Affine Types. In *ACM Symposium on Principles of Programming Languages (POPL), Austin, Texas*.

Aaron Turon, Konrad Borowski, Hidehito Yabuuchi, and Dan Aloni. 2017. Non-Lexical Lifetimes. https://github.com/rust-lang/rfcs/blob/master/text/2094-nll.md. Accessed: 2019-02-28.

Sebastian Ullrich. 2016. *Simple Verification of Rust Programs via Functional Purification.* Master's thesis. Karlsruhe Institute of Technology.

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement Types for Haskell. In *International Conference on Functional Programming (ICFP)* (Gothenburg, Sweden) *(ICFP '14)*. ACM, New York, NY, USA, 269–282. https://doi.org/10.1145/2628136.2628161

Philip Wadler. 1991. Is there a use for linear logic?. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation (PEPM)*.

David Wakeling and Colin Runciman. 1991. Linearity and Laziness. In *ACM Symposium on Functional Programming Languages and Computer Architecture (FPCA)*.

Aaron Weiss, Daniel Patterson, and Amal Ahmed. 2018. Rust Distilled: An Expressive Tower of Languages. *ML Family Workshop* (2018).

Andrew K. Wright and Matthias Felleisen. 1992. A Syntactic Approach to Type Soundness. *Information and Computation* (1992).

Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. In *ACM Symposium on Principles of Programming Languages (POPL), Philadelphia, Pennsylvania.*

A OXIDE SYNTAX

Variables	x	Functions	f	Type Vars.	α	Frame Vars.	φ
Concrete Prov.	r	Abstract Prov.	ϱ	Strings	$\texttt{str}$	Naturals	m, n, k

Path	q	::=	$\epsilon \mid n.q$
Places	π	::=	$x.q$
Place Expressions	p	::=	$x \mid *p \mid p.n$
Place Expression Contexts	$p^{\square}$	::=	$\square \mid *p^{\square} \mid p^{\square}.n$
Provenances	ρ	::=	$\varrho \mid r$
Ownership Qualifiers	ω	::=	$\texttt{shrd} \mid \texttt{uniq}$
Loans	ℓ	::=	${}^{\omega}p$
Kinds	κ	::=	$\star \mid \texttt{PRV} \mid \texttt{FRM}$
Base Types	τ^{B}	::=	$\texttt{bool} \mid \texttt{u32} \mid \texttt{unit}$
Sized Types	τ^{SI}	::=	$\tau^{\text{B}} \mid \alpha \mid \&\rho\ \omega\ \tau^{\text{XI}} \mid [\tau^{\text{SI}}; n] \mid (\tau_1^{\text{SI}}, \ldots, \tau_n^{\text{SI}})$
		$\mid$	$\forall <\overline{\varphi}, \overline{\varrho}, \overline{\alpha}> (\tau_1^{\text{SI}}, \ldots, \tau_n^{\text{SI}}) \xrightarrow{\Phi} \tau_r^{\text{SI}}\ \texttt{where}\ \overline{\varrho_1 : \varrho_2}$
Maybe Unsized Types	τ^{XI}	::=	$\tau^{\text{SI}} \mid [\tau^{\text{SI}}]$
Dead Types	τ^{SD}	::=	$\tau^{\text{SI}^{\dagger}} \mid (\tau_1^{\text{SD}}, \ldots, \tau_n^{\text{SD}})$
Maybe Dead Types	τ^{SX}	::=	$\tau^{\text{SI}} \mid \tau^{\text{SD}} \mid (\tau_1^{\text{SX}}, \ldots, \tau_n^{\text{SX}})$
Types	τ	::=	$\tau^{\text{XI}} \mid \tau^{\text{SX}}$
Constants	c	::=	$() \mid n \mid \texttt{true} \mid \texttt{false}$
Expressions	e	::=	$c \mid p \mid \&r\ \omega\ p \mid \&r\ \omega\ p[e] \mid \&r\ \omega\ p[\hat{e}_1..\hat{e}_2] \mid p := e$
		$\mid$	$\texttt{letprov}\ <r>\ \{\ e\ \} \mid \texttt{let}\ x : \tau^{\text{SI}} = e_1;\ e_2 \mid e_1;\ e_2$
		$\mid$	$\lvert x_1 : \tau_1^{\text{SI}}, \ldots, x_n : \tau_n^{\text{SI}} \rvert \rightarrow \tau_r^{\text{SI}}\ \{\ e\ \} \mid e_f ::<\overline{\Phi}, \overline{\rho}, \overline{\tau^{\text{SI}}}> (\hat{e}_1, \ldots, \hat{e}_n)$
		$\mid$	$\texttt{if}\ e_1\ \{\ e_2\ \}\ \texttt{else}\ \{\ e_3\ \} \mid [\hat{e}_1, \ldots, \hat{e}_n] \mid (\hat{e}_1, \ldots, \hat{e}_n)$
		$\mid$	$p[e] \mid \texttt{for}\ x\ \texttt{in}\ e_1\ \{\ e_2\ \} \mid \texttt{while}\ e_1\ \{\ e_2\ \} \mid \texttt{abort!(str)}$
Sequenceless Expressions	$\hat{e}$	::=	$c \mid p \mid \&r\ \omega\ p \mid \&r\ \omega\ p[\hat{e}] \mid \&r\ \omega\ p[\hat{e}_1..\hat{e}_2] \mid p := \hat{e}$
		$\mid$	$\texttt{letprov}\ <r>\ \{\ \hat{e}\ \} \mid$
		$\mid$	$\lvert x_1 : \tau_1^{\text{SI}}, \ldots, x_n : \tau_n^{\text{SI}} \rvert \rightarrow \tau_r^{\text{SI}}\ \{\ \hat{e}\ \} \mid \hat{e}_f ::<\overline{\Phi}, \overline{\rho}, \overline{\tau^{\text{SI}}}> (\hat{e}_1, \ldots, \hat{e}_n)$
		$\mid$	$\texttt{if}\ \hat{e}_1\ \{\ \hat{e}_2\ \}\ \texttt{else}\ \{\ \hat{e}_3\ \} \mid [\hat{e}_1, \ldots, \hat{e}_n] \mid (\hat{e}_1, \ldots, \hat{e}_n)$
		$\mid$	$p[\hat{e}] \mid \texttt{for}\ x\ \texttt{in}\ \hat{e}_1\ \{\ \hat{e}_2\ \} \mid \texttt{while}\ \hat{e}_1\ \{\ \hat{e}_2\ \} \mid \texttt{abort!(str)}$
Frame Expressions	Φ	::=	$\varphi \mid \mathcal{F}$
Global Environment	Σ	::=	$\bullet \mid \Sigma,\ \varepsilon$
Global Entries	ε	::=	$\texttt{fn}\ f <\overline{\varphi}, \overline{\varrho}, \overline{\alpha}> (x_1 : \tau_1^{\text{SI}}, \ldots, x_n : \tau_n^{\text{SI}}) \rightarrow \tau_r^{\text{SI}}\ \texttt{where}\ \overline{\varrho_1 : \varrho_2}\ \{\ e\ \}$
Type Environment	Δ	::=	$\bullet \mid \Delta,\ \alpha : \star \mid \Delta,\ \varrho : \texttt{PRV} \mid \Delta,\ \varphi : \texttt{FRM} \mid \Delta,\ \varrho :> \varrho'$
Frame Typing	$\mathcal{F}$	::=	$\bullet \mid \mathcal{F},\ x : \tau^{\text{SX}} \mid \mathcal{F},\ r \mapsto \{\ \overline{\ell}\ \}$
Stack Typing	Γ	::=	$\bullet \mid \Gamma \natural \mathcal{F}$

B STATICS

B.1 Well-Formedness Judgments

$\boxed{\vdash \Sigma}$
read: "Σ is well-formed"

$$\textsc{WF-GlobalEnv}\ \frac{\forall \varepsilon \in \Sigma.\ \Sigma \vdash \varepsilon}{\vdash \Sigma}$$

$\boxed{\Sigma \vdash \varepsilon}$
read: "ε is a well-formed function definition in Σ"

$$\textsc{WF-FunctionDefinition}\ \frac{\begin{array}{c}\Delta = \overline{\varphi : \mathsf{FRM}},\ \overline{\varrho : \mathsf{PRV}},\ \overline{\varrho_1 :> \varrho_2},\ \overline{\alpha : \star} \qquad \{\ \overline{\varrho_1}\ \} \subseteq \{\ \overline{\varrho}\ \} \qquad \{\ \overline{\varrho_2}\ \} \subseteq \{\ \overline{\varrho}\ \} \\ \Sigma;\ \Delta;\ \bullet \natural x_1 : \tau_1^{\text{SI}},\ \ldots,\ x_n : \tau_n^{\text{SI}} \vdash \boxed{e} : \tau_f^{\text{SI}} \Rightarrow \Gamma' \qquad \Delta;\ \bullet \vdash \tau_f^{\text{SI}} \lesssim \tau_r^{\text{SI}} \Rightarrow \bullet\end{array}}{\Sigma \vdash \mathtt{fn}\ f\texttt{<}\overline{\varphi},\ \overline{\varrho},\ \overline{\alpha}\texttt{>}(x_1 : \tau_1^{\text{SI}},\ \ldots,\ x_n : \tau_n^{\text{SI}})\ \rightarrow\ \tau_r^{\text{SI}}\ \mathtt{where}\ \overline{\varrho_1 : \varrho_2}\ \{\ e\ \}}$$

$\boxed{\vdash \Delta}$
read: "Δ is well-formed"

$$\textsc{WF-TVarEmpty}\ \frac{}{\vdash \bullet} \qquad \textsc{WF-TVarExtendEnv}\ \frac{}{\vdash \Delta,\ \varphi : \mathsf{FRM}} \qquad \textsc{WF-TVarExtendProv}\ \frac{}{\vdash \Delta,\ \varrho : \mathsf{PRV}} \qquad \textsc{WF-TVarExtendType}\ \frac{}{\vdash \Delta,\ \alpha : \star}$$

$$\textsc{WF-TvarExtendOutlives}\ \frac{\varrho_1 : \mathsf{PRV} \in \Delta \qquad \varrho_2 : \mathsf{PRV} \in \Delta}{\vdash \Delta,\ \varrho_1 :> \varrho_2}$$

$\boxed{\Sigma;\ \Delta \vdash \Gamma}$
read: "Γ is well-formed under Σ and Δ"

$$\textsc{WF-EmptyStackTyping}\ \frac{}{\Sigma;\ \Delta \vdash \bullet} \qquad \textsc{WF-StackTyping}\ \frac{\begin{array}{c}\Sigma;\ \Delta \vdash \Gamma \qquad \mathtt{places}(\mathcal{F}) \subseteq \mathrm{dom}(\Gamma \natural \mathcal{F}) \\ \mathrm{dom}(\mathcal{F}) \# \mathrm{dom}(\Gamma) \qquad \forall x : \tau \in \mathcal{F}.\ \Sigma;\ \Delta;\ \Gamma \natural \mathcal{F} \vdash \tau \\ \forall r \mapsto \{\ \overline{\ell}\ \} \in \mathcal{F}.\ \forall^{\omega} p \in \{\ \overline{\ell}\ \}.\ \exists \tau^{\text{XI}}.\ \Delta;\ \Gamma \natural \mathcal{F} \vdash_{\omega} p : \tau^{\text{XI}}\end{array}}{\Sigma;\ \Delta \vdash \Gamma \natural \mathcal{F}}$$

$\boxed{\vdash \Sigma;\ \Delta;\ \Gamma}$
read: "Σ, Δ, and Γ are well-formed."

$$\textsc{WF-Environments}\ \frac{\vdash \Sigma \qquad \vdash \Delta \qquad \Sigma;\ \Delta \vdash \Gamma}{\vdash \Sigma;\ \Delta;\ \Gamma}$$

$\boxed{\Sigma;\ \Delta;\ \Gamma \vdash \Phi}$
read: "Φ is a well-formed captured environment"

$$\textsc{WF-EnvVar}\ \frac{\Delta(\varphi) = \mathsf{FRM}}{\Sigma;\ \Delta;\ \Gamma \vdash \varphi} \qquad \textsc{WF-Env}\ \frac{\Sigma;\ \Delta \vdash \Gamma \natural \mathcal{F}_c}{\Sigma;\ \Delta;\ \Gamma \vdash \mathcal{F}_c}$$

$\boxed{\Delta;\ \Gamma \vdash \rho}$
read: "ρ is a well-formed provenance"

$$\textsc{WF-LocalProv}\ \frac{r \in \mathrm{dom}(\Gamma)}{\Delta;\ \Gamma \vdash r} \qquad \textsc{WF-AbstractProv}\ \frac{\Delta(\varrho) = \mathsf{PRV}}{\Delta;\ \Gamma \vdash \varrho}$$

$\boxed{\Sigma;\ \Delta;\ \Gamma \vdash \tau}$
read: "τ is a well-formed type under Σ, Δ, and Γ"

$$\textsc{WF-BaseType}\ \frac{}{\Sigma;\ \Delta;\ \Gamma \vdash \tau^{\mathrm{B}}} \qquad \textsc{WF-TVar}\ \frac{\Delta(\alpha) = \star}{\Sigma;\ \Delta;\ \Gamma \vdash \alpha}$$

$$\textsc{WF-Ref}\ \frac{(\Gamma(r) = \emptyset \vee \exists\, {}^{\omega} p \in \Gamma(r).\ \exists \tau_p^{\mathrm{XI}}.\ \Delta;\ \Gamma \vdash_{\omega} p : \tau_p^{\mathrm{XI}}.\ \tau^{\mathrm{XI}}\ \mathtt{occurs\ in}\ \tau_p^{\mathrm{XI}}) \quad \Sigma;\ \Delta;\ \Gamma \vdash \tau^{\mathrm{XI}}}{\Sigma;\ \Delta;\ \Gamma \vdash \&r\ \omega\ \tau^{\mathrm{XI}}}$$

$$\textsc{WF-AbstractRef}\ \frac{\Delta(\varrho) = \mathsf{PRV} \qquad \Sigma;\ \Delta;\ \Gamma \vdash \tau}{\Sigma;\ \Delta;\ \Gamma \vdash \&\varrho\ \omega\ \tau} \qquad \textsc{WF-Tuple}\ \frac{\forall i \in \{\,1 \ldots n\,\}.\ \Sigma;\ \Delta;\ \Gamma \vdash \tau_i^{\mathrm{SX}}}{\Sigma;\ \Delta;\ \Gamma \vdash (\tau_1^{\mathrm{SX}}, \ldots, \tau_n^{\mathrm{SX}})}$$

$$\textsc{WF-Function}\ \frac{\begin{array}{c}\Sigma;\ \Delta;\ \Gamma \vdash \Phi \qquad \Sigma;\ \Delta,\ \overline{\varphi : \mathsf{FRM}},\ \overline{\varrho : \mathsf{PRV}},\ \overline{\varrho_1 :> \varrho_2},\ \overline{\alpha : \star};\ \Gamma \vdash \tau_r^{\mathrm{SI}} \\ \forall i \in \{\,1 \ldots n\,\}.\ \Sigma;\ \Delta,\ \overline{\varphi : \mathsf{FRM}},\ \overline{\varrho : \mathsf{PRV}},\ \overline{\varrho_1 :> \varrho_2},\ \overline{\alpha : \star};\ \Gamma \vdash \tau_i^{\mathrm{SI}}\end{array}}{\Sigma;\ \Delta;\ \Gamma \vdash \forall <\overline{\varphi},\ \overline{\varrho},\ \overline{\alpha}>(\tau_1^{\mathrm{SI}}, \ldots, \tau_n^{\mathrm{SI}}) \xrightarrow{\Phi} \tau_r^{\mathrm{SI}}\ \mathtt{where}\ \overline{\varrho_1 : \varrho_2}}$$

$$\textsc{WF-Uninit}\ \frac{}{\Sigma;\ \Delta;\ \Gamma \vdash \tau^{\mathrm{SI}\dagger}} \qquad \textsc{WF-Array}\ \frac{\Sigma;\ \Delta;\ \Gamma \vdash \tau^{\mathrm{SI}}}{\Sigma;\ \Delta;\ \Gamma \vdash [\tau^{\mathrm{SI}};\ n]}$$

$$\textsc{WF-Slice}\ \frac{\Sigma;\ \Delta;\ \Gamma \vdash \tau^{\mathrm{SI}}}{\Sigma;\ \Delta;\ \Gamma \vdash [\tau^{\mathrm{SI}}]}$$

B.2 Subtyping & Provenance Subtyping

$\boxed{\Delta;\ \Gamma \vdash \tau_1 \lesssim \tau_2 \Rightarrow \Gamma'}$
read: "τ_1 is a subtype of τ_2 under Δ and Γ, producing Γ'"

S-Refl
$$\frac{}{\Delta;\ \Gamma \vdash \tau_1 \lesssim \tau_1 \Rightarrow \Gamma}$$

S-Trans
$$\frac{\Delta;\ \Gamma \vdash \tau_1 \lesssim \tau_2 \Rightarrow \Gamma' \qquad \Delta;\ \Gamma' \vdash \tau_2 \lesssim \tau_3 \Rightarrow \Gamma''}{\Delta;\ \Gamma \vdash \tau_1 \lesssim \tau_3 \Rightarrow \Gamma''}$$

S-Array
$$\frac{\Delta;\ \Gamma \vdash \tau_1 \lesssim \tau_2 \Rightarrow \Gamma'}{\Delta;\ \Gamma \vdash [\tau_1;\ n] \lesssim [\tau_2;\ n] \Rightarrow \Gamma'}$$

S-Slice
$$\frac{\Delta;\ \Gamma \vdash \tau_1 \lesssim \tau_2 \Rightarrow \Gamma'}{\Delta;\ \Gamma \vdash [\tau_1] \lesssim [\tau_2] \Rightarrow \Gamma'}$$

S-SharedRef
$$\frac{\Delta;\ \Gamma \vdash \rho_1 :> \rho_2 \Rightarrow \Gamma' \qquad \Delta;\ \Gamma' \vdash \tau_1 \lesssim \tau_2 \Rightarrow \Gamma''}{\Delta;\ \Gamma \vdash \&\rho_1\ \mathsf{shrd}\ \tau_1 \lesssim \&\rho_2\ \mathsf{shrd}\ \tau_2 \Rightarrow \Gamma''}$$

S-UniqueRef
$$\frac{\Delta;\ \Gamma \vdash \rho_1 :> \rho_2 \Rightarrow \Gamma' \qquad \Delta;\ \Gamma' \vdash \tau_1 \lesssim \tau_2 \Rightarrow \Gamma'' \qquad \Delta;\ \Gamma' \vdash \tau_2 \lesssim \tau_1 \Rightarrow \Gamma''}{\Delta;\ \Gamma \vdash \&\rho_1\ \mathsf{uniq}\ \tau_1 \lesssim \&\rho_2\ \mathsf{uniq}\ \tau_2 \Rightarrow \Gamma''}$$

S-Tuple
$$\frac{\forall i \in \{\,1 \ldots n\,\}.\ \Delta;\ \Gamma_{n-1} \vdash \tau_i \lesssim \tau_i' \Rightarrow \Gamma_i}{\Delta;\ \Gamma \vdash (\tau_1 \ldots \tau_n) \lesssim (\tau_1' \ldots \tau_n') \Rightarrow \Gamma_n}$$

S-Uninit
$$\frac{\Delta;\ \Gamma \vdash \tau_1^{\mathrm{SI}} \lesssim \tau_2^{\mathrm{SI}} \Rightarrow \Gamma'}{\Delta;\ \Gamma \vdash \tau_1^{\mathrm{SI}} \lesssim \tau_2^{\mathrm{SI}^\dagger} \Rightarrow \Gamma}$$

$\boxed{\Delta;\ \Gamma \vdash \rho_1 :> \rho_2 \Rightarrow \Gamma'}$
read: "ρ_1 outlives ρ_2 under Δ and Γ, producing Γ'"

OL-Refl
$$\frac{}{\Delta;\ \Gamma \vdash \rho :> \rho \Rightarrow \Gamma}$$

OL-AbstractProvenances
$$\frac{\varrho_1 : \mathsf{PRV} \in \Delta \qquad \varrho_2 : \mathsf{PRV} \in \Delta \qquad \varrho_1 :> \varrho_2 \in \Delta}{\Delta;\ \Gamma \vdash \varrho_1 :> \varrho_2 \Rightarrow \Gamma}$$

OL-Trans
$$\frac{\Delta;\ \Gamma \vdash \varrho_1 :> \varrho_2 \Rightarrow \Gamma' \qquad \Delta;\ \Gamma' \vdash \varrho_2 :> \varrho_3 \Rightarrow \Gamma''}{\Delta;\ \Gamma \vdash \varrho_1 :> \varrho_3 \Rightarrow \Gamma''}$$

OL-LocalProvenances
$$\frac{\forall \pi : \&r_1\ \omega\ \tau \in \Gamma.\ \nexists r'.\ {}^{\omega} * \pi\ \in \Gamma(r') \qquad r_1\ \mathtt{occurs\ before}\ r_2\ \mathtt{in}\ \Gamma}{\Delta;\ \Gamma \vdash r_1 :> r_2 \Rightarrow \Gamma[r_2 \mapsto \{\,\Gamma(r_1) \cup \Gamma(r_2)\,\}]}$$

OL-LocalProvAbsProv
$$\frac{\Gamma_{1,0}(r) = \{\ \overline{{}^{\omega}p}^{\,n}\ \} \neq \emptyset \qquad \overline{\forall \pi.\ p \neq \pi} \qquad \varrho : \mathsf{PRV} \in \Delta \qquad \forall i \in \{\,1 \ldots n\,\}.\ \forall j \in \{\,1 \ldots m_i\,\}.\ \Delta;\ \Gamma_{i,j-1} \vdash \rho_{i,j} :> \varrho \Rightarrow \Gamma_{i,j} \qquad \Delta;\ \Gamma_0 \vdash_{\mathsf{shrd}} p_i : \_,\ \overline{\rho_i}^{m_i}}{\Delta;\ \Gamma_{1,0} \vdash r :> \varrho \Rightarrow \Gamma_{n,m_n}}$$

OL-AbsProvLocalProv
$$\frac{\varrho : \mathsf{PRV} \in \Delta \qquad r \in \mathrm{dom}(\Gamma)}{\Delta;\ \Gamma \vdash \varrho :> r \Rightarrow \Gamma}$$

$\boxed{\Delta;\ \Gamma \vdash \overline{\rho_1 :> \rho_2} \Rightarrow \Gamma'}$

OL-Bounds
$$\frac{\forall i \in \{\,1 \ldots n\,\}.\ \Delta;\ \Gamma_{i-1} \vdash \rho_i :> \rho_i' \Rightarrow \Gamma_i}{\Delta;\ \Gamma_0 \vdash \overline{\rho :> \rho'} \Rightarrow \Gamma_n}$$

B.3 Ownership Safety

$\boxed{\Delta;\ \Gamma \vdash^{\overline{\pi}}_{\omega} p \Rightarrow \{\ \overline{\omega p}\ \}}$ where $\Delta;\ \Gamma \vdash_{\omega} p \Rightarrow \{\ \overline{\omega p}\ \}$ means $\Delta;\ \Gamma \vdash^{\bullet}_{\omega} p \Rightarrow \{\ \overline{\omega p}\ \}$.
read: "p is ω-safe under Δ and Γ, with reborrow exclusion list $\overline{\pi}$, and may point to any of the loans in $\overline{\omega p}$"

O-SafePlace
$$\frac{\begin{array}{c}\forall r' \mapsto \{\ \overline{\ell}\ \} \in \Gamma.\ (\forall\, {}^{\omega'} p^{\square}[\pi'] \in \{\ \overline{\ell}\ \}.(\omega = \mathsf{uniq} \vee \omega' = \mathsf{uniq}) \implies \pi' \# \pi) \\ \vee\ (\exists \pi' : \&r' \omega' \tau' \in \Gamma \wedge (\forall \pi' : \&r' \omega' \tau' \in \Gamma.\ \pi' \in \{\ \overline{\pi_e}\ \}))\end{array}}{\Delta;\ \Gamma \vdash^{\overline{\pi_e}}_{\omega} \pi \Rightarrow \{\ {}^{\omega}\pi\ \}}$$

O-Deref
$$\frac{\begin{array}{c}\Gamma(\pi) = \&r\ \omega_\pi\ \tau_\pi \qquad \Gamma(r) = \{\ \overline{\omega' p_i}\ \} \qquad \overline{p_i = p_i^{\square}[\pi_i]} \qquad \omega \lesssim \omega_\pi \\ \forall i \in \{1 \ldots n\}.\ \Delta;\ \Gamma \vdash^{\overline{\pi_e}, \overline{\pi_i}, \pi}_{\omega} p^{\square}[p_i] \Rightarrow \{\ \overline{\omega p_i'}\ \} \\ \forall r' \mapsto \{\ \overline{\ell}\ \} \in \Gamma.\ (\forall\, {}^{\omega'} p \in \{\ \overline{\ell}\ \}.(\omega = \mathsf{uniq} \vee \omega' = \mathsf{uniq}) \implies p \# p^{\square}[*\pi]) \\ \vee\ (\exists \pi' : \&r' \omega' \tau' \in \Gamma \wedge (\forall \pi' : \&r' \omega' \tau' \in \Gamma.\ \pi' \in \{\ \overline{\pi_e}, \overline{\pi_i}, \pi\ \}))\end{array}}{\Delta;\ \Gamma \vdash^{\overline{\pi_e}}_{\omega} p^{\square}[*\pi] \Rightarrow \{\ \overline{\omega p_1'},\ \ldots\ \overline{\omega p_n'},\ {}^{\omega}p^{\square}[*\pi]\ \}}$$

O-DerefAbs
$$\frac{\begin{array}{c}\Gamma(\pi) = \&\varrho\ \omega_\pi\ \tau_\pi \qquad \Delta;\ \Gamma \vdash_{\omega} p^{\square}[*\pi] : \tau \qquad \omega \lesssim \omega_\pi \\ \forall r' \mapsto \{\ \overline{\ell}\ \} \in \Gamma.\ (\forall\, {}^{\omega'} p \in \{\ \overline{\ell}\ \}.(\omega = \mathsf{uniq} \vee \omega' = \mathsf{uniq}) \implies p \# p^{\square}[*\pi]) \\ \vee\ (\exists \pi' : \&r' \omega' \tau' \in \Gamma \wedge (\forall \pi' : \&r' \omega' \tau' \in \Gamma.\ \pi' \in \{\ \overline{\pi_e}, \pi\ \}))\end{array}}{\Delta;\ \Gamma \vdash^{\overline{\pi_e}}_{\omega} p^{\square}[*\pi] \Rightarrow \{\ {}^{\omega}p^{\square}[*\pi]\ \}}$$

B.4 Typing

$\boxed{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e} : \tau \Rightarrow \Gamma'}$ where $\vdash \Sigma;\ \Delta;\ \Gamma$ and $\Sigma;\ \Delta;\ \Gamma' \vdash \tau$
read: "e has type τ under Σ, Δ, and Γ, producing output context Γ"

T-Move
$$\frac{\begin{array}{c}\Delta;\ \Gamma \vdash_{\mathsf{uniq}} \pi \Rightarrow \{\ {}^{\mathsf{uniq}}\pi\ \} \\ \Gamma(\pi) = \tau^{\mathrm{SI}} \qquad \mathsf{noncopyable}_{\Sigma}\ \tau^{\mathrm{SI}}\end{array}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\pi} : \tau^{\mathrm{SI}} \Rightarrow \Gamma[\pi \mapsto \tau^{\mathrm{SI}^{\dagger}}]}$$

T-Copy
$$\frac{\begin{array}{c}\Delta;\ \Gamma \vdash_{\mathsf{shrd}} p \Rightarrow \{\ \overline{\ell}\ \} \\ \Delta;\ \Gamma \vdash_{\mathsf{shrd}} p : \tau^{\mathrm{SI}} \qquad \mathsf{copyable}_{\Sigma}\ \tau^{\mathrm{SI}}\end{array}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{p} : \tau^{\mathrm{SI}} \Rightarrow \Gamma}$$

T-Borrow
$$\frac{\begin{array}{c}\Gamma(r) = \emptyset \qquad \Delta;\ \Gamma \vdash_{\omega} p \Rightarrow \{\ \overline{\ell}\ \} \\ \Delta;\ \Gamma \vdash_{\omega} p : \tau^{\mathrm{XI}}\end{array}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\&r\ \omega\ p} : \&r\ \omega\ \tau^{\mathrm{XI}} \Rightarrow \Gamma[r \mapsto \{\ \overline{\ell}\ \}]}$$

T-BorrowIndex
$$\frac{\begin{array}{c}\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e} : \mathsf{u32} \Rightarrow \Gamma' \qquad \Gamma'(r) = \emptyset \\ \Delta;\ \Gamma' \vdash_{\omega} p \Rightarrow \{\ \overline{\ell}\ \} \qquad \Delta;\ \Gamma' \vdash_{\omega} p : \tau^{\mathrm{XI}} \\ \tau^{\mathrm{XI}} = [\tau^{\mathrm{SI}};\ n] \vee \tau^{\mathrm{XI}} = [\tau^{\mathrm{SI}}]\end{array}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\&r\ \omega\ p[e]} : \&r\ \omega\ \tau^{\mathrm{SI}} \Rightarrow \Gamma'[r \mapsto \{\ \overline{\ell}\ \}]}$$

T-BorrowSlice
$$\frac{\begin{array}{c}\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\hat{e}_1} : \mathsf{u32} \Rightarrow \Gamma_1 \qquad \Sigma;\ \Delta;\ \Gamma_1 \vdash \boxed{\hat{e}_2} : \mathsf{u32} \Rightarrow \Gamma_2 \qquad \Gamma_2(r) = \emptyset \\ \Delta;\ \Gamma_2 \vdash_{\omega} p \Rightarrow \{\ \overline{\ell}\ \} \qquad \Delta;\ \Gamma_2 \vdash_{\omega} p : [\tau^{\mathrm{SI}}]\end{array}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\&r\ \omega\ p[\hat{e}_1..\hat{e}_2]} : \&r\ \omega\ [\tau^{\mathrm{SI}}] \Rightarrow \Gamma_2[r \mapsto \{\ \overline{\ell}\ \}]}$$

T-IndexCopy
$$\frac{\begin{array}{c}\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e} : \mathsf{u32} \Rightarrow \Gamma' \qquad \Delta;\ \Gamma' \vdash_{\mathsf{shrd}} p \Rightarrow \{\ \overline{\ell}\ \} \\ \mathsf{copyable}_{\Sigma}\ \tau^{\mathrm{SI}} \qquad \Delta;\ \Gamma' \vdash_{\mathsf{shrd}} p : \tau^{\mathrm{XI}} \qquad \tau^{\mathrm{XI}} = [\tau^{\mathrm{SI}};\ n] \vee \tau^{\mathrm{XI}} = [\tau^{\mathrm{SI}}]\end{array}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{p[e]} : \tau^{\mathrm{SI}} \Rightarrow \Gamma'}$$

T-Seq

$$\frac{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e_1} : \tau_1^{\text{SI}} \Rightarrow \Gamma_1 \qquad \Sigma;\ \Delta;\ \text{gc-loans}(\Gamma_1) \vdash \boxed{e_2} : \tau_2^{\text{SI}} \Rightarrow \Gamma_2}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e_1;\ e_2} : \tau_2^{\text{SI}} \Rightarrow \Gamma_2}$$

T-Branch

$$\frac{\begin{array}{c}\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e_1} : \texttt{bool} \Rightarrow \Gamma_1 \qquad \Sigma;\ \Delta;\ \Gamma_1 \vdash \boxed{e_2} : \tau_2^{\text{SI}} \Rightarrow \Gamma_2 \\ \Sigma;\ \Delta;\ \Gamma_1 \vdash \boxed{e_3} : \tau_3^{\text{SI}} \Rightarrow \Gamma_3 \qquad \tau^{\text{SI}} = \tau_2^{\text{SI}} \vee \tau^{\text{SI}} = \tau_3^{\text{SI}} \\ \Delta;\ \Gamma_2 \vdash \tau_2^{\text{SI}} \lesssim \tau^{\text{SI}} \Rightarrow \Gamma_2' \qquad \Delta;\ \Gamma_3 \vdash \tau_3^{\text{SI}} \lesssim \tau^{\text{SI}} \Rightarrow \Gamma_3' \qquad \Gamma_2' \Cup \Gamma_3' = \Gamma'\end{array}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\texttt{if}\ e_1\ \{\ e_2\ \}\ \texttt{else}\ \{\ e_3\ \}} : \tau^{\text{SI}} \Rightarrow \Gamma'}$$

T-LetProv

$$\frac{\Sigma;\ \Delta;\ \Gamma,\ r \mapsto \{\} \vdash \boxed{e} : \tau^{\text{SI}} \Rightarrow \Gamma',\ r \mapsto \{\bar{\ell}\}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\texttt{letprov <}r\texttt{> }\{\ e\ \}} : \tau^{\text{SI}} \Rightarrow \Gamma'}$$

T-Let

$$\frac{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e_1} : \tau_1^{\text{SI}} \Rightarrow \Gamma_1 \qquad \Delta;\ \Gamma_1 \vdash \tau_1^{\text{SI}} \lesssim \tau_a^{\text{SI}} \Rightarrow \Gamma_1' \qquad \Sigma;\ \Delta;\ \text{gc-loans}(\Gamma_1',\ x\ :\ \tau_a^{\text{SI}}) \vdash \boxed{e_2} : \tau_2^{\text{SI}} \Rightarrow \Gamma_2,\ x\ :\ \tau^{\text{SD}}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\texttt{let}\ x : \tau_a^{\text{SI}} = e_1;\ e_2} : \tau_2^{\text{SI}} \Rightarrow \Gamma_2}$$

T-Assign

$$\frac{\begin{array}{c}\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e} : \tau^{\text{SI}} \Rightarrow \Gamma_1 \qquad \Gamma_1(\pi) = \tau^{\text{SX}} \\ (\tau^{\text{SX}} = \tau^{\text{SD}} \vee \Delta;\ \Gamma_1 \vdash_{\texttt{uniq}} \pi \Rightarrow \{\ ^{\texttt{uniq}}\pi\ \}) \\ \Delta;\ \Gamma_1 \vdash \tau^{\text{SI}} \lesssim \tau^{\text{SX}} \Rightarrow \Gamma'\end{array}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\pi \coloneqq e} : \texttt{unit} \Rightarrow \Gamma'[\pi \mapsto \tau^{\text{SI}}] \Subset \pi}$$

T-AssignDeref

$$\frac{\begin{array}{c}\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e} : \tau_n^{\text{SI}} \Rightarrow \Gamma_1 \qquad \Delta;\ \Gamma_1 \vdash_{\texttt{uniq}} p : \tau_o^{\text{SI}} \\ \Delta;\ \Gamma_1 \vdash_{\texttt{uniq}} p \Rightarrow \{\ \bar{\ell}\ \} \qquad \Delta;\ \Gamma_1 \vdash \tau_n^{\text{SI}} \lesssim \tau_o^{\text{SI}} \Rightarrow \Gamma'\end{array}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{p \coloneqq e} : \texttt{unit} \Rightarrow \Gamma' \Subset p}$$

T-While

$$\frac{\begin{array}{c}\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e_1} : \texttt{bool} \Rightarrow \Gamma_1 \qquad \Sigma;\ \Delta;\ \Gamma_1 \vdash \boxed{e_2} : \texttt{unit} \Rightarrow \Gamma_2 \\ \Sigma;\ \Delta;\ \Gamma_2 \vdash \boxed{e_1} : \texttt{bool} \Rightarrow \Gamma_2 \qquad \Sigma;\ \Delta;\ \Gamma_2 \vdash \boxed{e_2} : \texttt{unit} \Rightarrow \Gamma_2\end{array}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\texttt{while}\ e_1\ \{\ e_2\ \}} : \texttt{unit} \Rightarrow \Gamma_2}$$

T-ForArray

$$\frac{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e_1} : [\tau^{\text{SI}};\ n] \Rightarrow \Gamma_1 \qquad \Sigma;\ \Delta;\ \Gamma_1,\ x\ :\ \tau^{\text{SI}} \vdash \boxed{e_2} : \texttt{unit} \Rightarrow \Gamma_1,\ x\ :\ \tau^{\text{SD}}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\texttt{for}\ x\ \texttt{in}\ e_1\ \{\ e_2\ \}} : \texttt{unit} \Rightarrow \Gamma_1}$$

T-ForSlice

$$\frac{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e_1} : \&\rho\ \omega\ [\tau^{\text{SI}}] \Rightarrow \Gamma_1 \qquad \Sigma;\ \Delta;\ \Gamma_1,\ x\ :\ \&\rho\ \omega\ \tau^{\text{SI}} \vdash \boxed{e_2} : \texttt{unit} \Rightarrow \Gamma_1,\ x\ :\ \tau_1^{\text{SX}}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\texttt{for}\ x\ \texttt{in}\ e_1\ \{\ e_2\ \}} : \texttt{unit} \Rightarrow \Gamma_2}$$

T-Function

$$\frac{\Sigma(f) = \texttt{fn}\ f\texttt{<}\overline{\varphi},\ \overline{\varrho},\ \overline{\alpha}\texttt{>}(x_1 : \tau_1^{\text{SI}},\ \ldots,\ x_n : \tau_n^{\text{SI}})\ \rightarrow\ \tau_r^{\text{SI}}\ \texttt{where}\ \overline{\varrho_1 : \varrho_2}\ \{\ e\ \}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{f} : \forall\texttt{<}\overline{\varphi},\ \overline{\varrho},\ \overline{\alpha}\texttt{>}(\tau_1^{\text{SI}},\ \ldots,\ \tau_n^{\text{SI}})\ \rightarrow\ \tau_r^{\text{SI}}\ \texttt{where}\ \overline{\varrho_1 : \varrho_2} \Rightarrow \Gamma}$$

T-Closure

$$\frac{\begin{array}{c}\text{free-vars}(e) \setminus \overline{x} = \overline{x_f} \qquad \text{free-nc-vars}_\Gamma(e) = \overline{x_{nc}} \qquad \overline{r} = \overline{\text{free-provs}(\Gamma(x_f))},\ \text{free-provs}(e) \\ \mathcal{F}_c = \overline{r \mapsto \Gamma(r)},\ \overline{x_f\ :\ \Gamma(x_f)} \qquad \Sigma;\ \Delta;\ \Gamma[\overline{x_{nc} \mapsto \Gamma(x_{nc})^\dagger}] \natural \mathcal{F}_c,\ x_1\ :\ \tau_1^{\text{SI}},\ \ldots,\ x_n\ :\ \tau_n^{\text{SI}} \vdash \boxed{e} : \tau_r^{\text{SI}} \Rightarrow \Gamma' \natural \mathcal{F}\end{array}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{|x_1 : \tau_1^{\text{SI}},\ \ldots,\ x_n : \tau_n^{\text{SI}}|\ \rightarrow\ \tau_r^{\text{SI}}\ \{\ e\ \}} : (\tau_1^{\text{SI}},\ \ldots,\ \tau_n^{\text{SI}}) \xrightarrow{\mathcal{F}_c} \tau_r^{\text{SI}} \Rightarrow \Gamma'}$$

T-App

$$\frac{\begin{array}{c}\overline{\Sigma;\ \Delta;\ \Gamma \vdash \Phi} \qquad \overline{\Delta;\ \Gamma \vdash \rho} \qquad \overline{\Sigma;\ \Delta;\ \Gamma \vdash \tau^{\text{SI}}} \\ \Sigma;\ \Delta;\ \Gamma \vdash \boxed{\hat{e}_f} : \forall\texttt{<}\overline{\varphi},\ \overline{\varrho},\ \overline{\alpha}\texttt{>}(\tau_1^{\text{SI}},\ \ldots,\ \tau_n^{\text{SI}}) \xrightarrow{\Phi_c} \tau_f^{\text{SI}}\ \texttt{where}\ \overline{\varrho_1 : \varrho_2} \Rightarrow \Gamma_0 \\ \forall i \in \{\ 1\ \ldots\ n\ \}.\ \Sigma;\ \Delta;\ \Gamma_{i-1} \vdash \boxed{\hat{e}_i} : \tau_i^{\text{SI}}\ \overline{[\Phi/_\varphi]}\ \overline{[\rho/_\varrho]}\ \overline{[\tau^{\text{SI}}/_\alpha]} \Rightarrow \Gamma_i \qquad \Delta;\ \Gamma_n \vdash \overline{\varrho_2\ [\rho/_\varrho]} :> \overline{\varrho_1\ [\rho/_\varrho]} \Rightarrow \Gamma_b\end{array}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\hat{e}_f ::\texttt{<}\overline{\Phi},\ \overline{\rho},\ \overline{\tau^{\text{SI}}}\texttt{>}(\hat{e}_1,\ \ldots,\ \hat{e}_n)} : \tau_f^{\text{SI}}\ \overline{[\Phi/_\varphi]}\ \overline{[\rho/_\varrho]}\ \overline{[\tau^{\text{SI}}/_\alpha]} \Rightarrow \Gamma_b}$$

T-Abort
$$\frac{}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\texttt{abort!(str)}} : \tau^{\text{SX}} \Rightarrow \Gamma'}$$

T-Unit
$$\frac{}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{()} : \texttt{unit} \Rightarrow \Gamma}$$

T-u32
$$\frac{}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{n} : \texttt{u32} \Rightarrow \Gamma}$$

T-True
$$\frac{}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\texttt{true}} : \texttt{bool} \Rightarrow \Gamma}$$

T-False
$$\frac{}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\texttt{false}} : \texttt{bool} \Rightarrow \Gamma}$$

T-Tuple
$$\frac{\forall i \in \{\ 1\ \ldots\ n\ \}.\ \Sigma;\ \Delta;\ \Gamma_{i-1} \vdash \boxed{\hat{e}_i} : \tau_i^{\text{SI}} \Rightarrow \Gamma_i}{\Sigma;\ \Delta;\ \Gamma_0 \vdash \boxed{(\hat{e}_1,\ \ldots,\ \hat{e}_n)} : (\tau_1^{\text{SI}},\ \ldots,\ \tau_n^{\text{SI}}) \Rightarrow \Gamma_n}$$

T-Array
$$\frac{\forall i \in \{\ 1\ \ldots\ n\ \}.\ \Sigma;\ \Delta;\ \Gamma_{i-1} \vdash \boxed{\hat{e}_i} : \tau^{\text{SI}} \Rightarrow \Gamma_i}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{[\hat{e}_1,\ \ldots,\ \hat{e}_n]} : [\tau^{\text{SI}};\ n] \Rightarrow \Gamma_n}$$

T-Slice
$$\frac{\forall i \in \{\ 1\ \ldots\ n\ \}.\ \Sigma;\ \Delta;\ \Gamma_{i-1} \vdash \boxed{\hat{e}_i} : \tau^{\text{SI}} \Rightarrow \Gamma_i}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{[\hat{e}_1,\ \ldots,\ \hat{e}_n]} : [\tau^{\text{SI}}] \Rightarrow \Gamma_n}$$

T-Drop
$$\frac{\Gamma(\pi) = \tau_\pi^{\text{SI}} \qquad \Sigma;\ \Delta;\ \Gamma[\pi \mapsto \tau_\pi^{\text{SI}^\dagger}] \vdash \boxed{e} : \tau^{\text{SX}} \Rightarrow \Gamma_f}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e} : \tau^{\text{SX}} \Rightarrow \Gamma_f}$$

B.5 Additional Judgments

$\boxed{\omega \lesssim \omega'}$
read: "ω is less than ω' in the qualifier ordering"

QO-Refl
$$\frac{}{\omega \lesssim \omega}$$

QO-ShrdUniq
$$\frac{}{\texttt{shrd} \lesssim \texttt{uniq}}$$

$\boxed{\Sigma;\ \Delta \vdash \Gamma \lesssim \Gamma'}$
read: "Γ is related to Γ' under Σ and Δ"

R-Env
$$\frac{\begin{array}{c}\vdash \Sigma;\ \Delta;\ \Gamma \qquad \vdash \Sigma;\ \Delta;\ \Gamma' \qquad \text{dom}(\Gamma) = \text{dom}(\Gamma') \\ \forall x : \tau \in \Gamma'.\ \forall r \text{ that occurs in } \tau.\ \Gamma(r) = \Gamma'(r) \\ \forall r \in \text{dom}(\Gamma).\ \Gamma(r) = \Gamma'(r) \vee \Gamma'(r) = \emptyset \\ \forall \pi \in \text{dom}(\Gamma).\ \Gamma'(\pi) = \Gamma(\pi) \vee \Gamma'(\pi) = \Gamma(\pi)^\dagger\end{array}}{\Sigma;\ \Delta \vdash \Gamma \lesssim \Gamma'}$$

$\boxed{\Delta;\ \Gamma \vdash_\omega p : \tau,\ \{\ \overline{\rho}\ \}}$
read: "p in an ω context has type τ under Δ and Γ, passing through provenances in $\overline{\rho}$"

TC-Var
$$\frac{\Gamma(x) = \tau^{\text{SI}}}{\Delta;\ \Gamma \vdash_\omega x : \tau^{\text{SI}},\ \emptyset}$$

TC-Proj
$$\frac{\Delta;\ \Gamma \vdash_\omega p : (\tau_1^{\text{SI}},\ \ldots,\ \tau_i^{\text{SI}},\ \ldots,\ \tau_n^{\text{SI}}),\ \{\ \overline{\rho_p}\ \}}{\Delta;\ \Gamma \vdash_\omega p.i : \tau_i^{\text{SI}},\ \{\ \overline{\rho_p}\ \}}$$

TC-Deref
$$\frac{\Delta;\ \Gamma \vdash_\omega p : \&\rho\ \omega'\ \tau^{\text{XI}},\ \{\ \overline{\rho_p}\ \} \qquad \omega \lesssim \omega' \qquad \Delta;\ \Gamma \vdash \overline{\rho :> \rho_p} \Rightarrow \Gamma_f}{\Delta;\ \Gamma \vdash_\omega *p : \tau^{\text{XI}},\ \{\ \overline{\rho_p},\ \rho\ \}}$$

$\boxed{\Delta;\ \Gamma \vdash_\omega p : \tau}$
read: "p in an ω context has type τ under Δ and Γ"

$$\Delta;\ \Gamma \vdash_\omega p : \tau = \Delta;\ \Gamma \vdash_\omega p : \tau,\ \_$$

C METAFUNCTIONS

free-nc-vars$_\sigma(e)$ = all the variables x free in e which are bound to values in σ that are `non-copyable`.
free-nc-vars$_\Gamma(e)$ = all the variables x free in e which are bound to types in Γ that are `non-copyable`.
$\pi_1 \# \pi_2 = \pi_1$ is not a prefix of π_2 and π_2 is not a prefix of π_1 and $\pi_1 \neq \pi_2$.

$\boxed{\Gamma_1 \Cup \Gamma_2 = \Gamma}$

$$
\begin{array}{lcl}
(\Gamma_1, x : \tau) \Cup (\Gamma_2, x : \tau) & = & (\Gamma_1 \Cup \Gamma_2), x : \tau \\
(\Gamma_1, r : \{\, \overline{\ell} \,\}) \Cup (\Gamma_2, r : \{\, \overline{\ell'} \,\}) & = & (\Gamma_1 \Cup \Gamma_2), r \mapsto \{\, \overline{\ell}, \overline{\ell'} \,\} \\
(\Gamma_1 \natural \bullet) \Cup (\Gamma_2 \natural \bullet) & = & \Gamma_1 \Cup \Gamma_2 \natural \bullet \\
\bullet \Cup \bullet & = & \bullet
\end{array}
$$

$\boxed{\mathtt{places}(\Gamma) = \{\, \overline{\pi} \,\}}$

$$
\begin{array}{l}
\mathtt{places}\ (\bullet) \ = \ \emptyset \\
\mathtt{places}\ (\Gamma,\ r \mapsto \{\, \overline{{}^{\omega} p} \,\}) \ = \ \{\, \pi \mid {}^{\omega_i} p_i \in \{\, \overline{{}^{\omega} p} \,\} \wedge (p_i = \pi \vee p_i = p^{\square}[{*}\pi]) \,\} \cup \mathtt{places}\ \Gamma \\
\mathtt{places}\ (\Gamma,\ x : \tau) = \mathtt{places}\ (\Gamma \natural \bullet) = \mathtt{places}\ (\Gamma)
\end{array}
$$

$\boxed{v.q \rightsquigarrow C \boxplus v}$

$$\text{DV-End}\quad \frac{}{v.\epsilon \rightsquigarrow \square \boxplus v}$$

$$\text{DV-Projection}\quad \frac{v_i.q \rightsquigarrow C \boxplus v}{(v_0, \ldots, v_i, \ldots, v_n).i.q \rightsquigarrow (v_0, \ldots, C, \ldots, v_n) \boxplus v}$$

$\boxed{\sigma[\pi \mapsto v]}$

$$
\begin{array}{l}
\sigma[x.q \mapsto v] = \sigma[x \mapsto C[v]] \\
\quad \textbf{where}\ \ \sigma(x).q \rightsquigarrow C \boxplus \_
\end{array}
$$

$\boxed{\sigma(\pi) = v}$

$$
\begin{array}{l}
\sigma(x.q) = v \\
\quad \textbf{where}\ \ \sigma(x).q \rightsquigarrow \_ \boxplus v
\end{array}
$$

$\boxed{\tau.q \rightsquigarrow \tau_\square \boxplus \tau}$

$$\text{D-End}\quad \frac{}{\tau.\epsilon \rightsquigarrow \square \boxplus \tau}$$

$$\text{D-Projection}\quad \frac{\tau_i.q \rightsquigarrow \tau_\square \boxplus \tau}{(\tau_0, \ldots, \tau_i, \ldots, \tau_n).i.q \rightsquigarrow (\tau_0, \ldots, \tau_\square, \ldots, \tau_n) \boxplus \tau}$$

$\boxed{\Gamma[\pi \mapsto \tau] = \Gamma'}$

$$
\begin{array}{l}
\Gamma[x.q \mapsto \tau] = \Gamma[x \mapsto \tau_\square[\tau]] \\
\quad \textbf{where}\ \ \Gamma(x).q \rightsquigarrow \tau_\square \boxplus \_
\end{array}
$$

$\boxed{\Gamma(\pi) = \tau}$

$$\Gamma(x.q) = \tau \quad \textbf{where}\ \ \Gamma(x).q \rightsquigarrow \_ \boxplus \tau$$

$\boxed{\texttt{noncopyable}_\Sigma\ \tau}$

$$\begin{aligned}
\texttt{noncopyable}_\Sigma\ \tau^{\text{B}} &= \bot \\
\texttt{noncopyable}_\Sigma\ \alpha &= \top \\
\texttt{noncopyable}_\Sigma\ \&_\ \texttt{uniq}\ _ &= \top \\
\texttt{noncopyable}_\Sigma\ \&_\ \texttt{shrd}\ _ &= \bot \\
\texttt{noncopyable}_\Sigma\ \forall <_>(_) \xrightarrow{} _ &= \bot \\
\texttt{noncopyable}_\Sigma\ [\tau;\ _] &= \texttt{noncopyable}_\Sigma\ \tau \\
\texttt{noncopyable}_\Sigma\ [\tau] &= \texttt{noncopyable}_\Sigma\ \tau \\
\texttt{noncopyable}_\Sigma\ (\tau, \ldots) &= \texttt{noncopyable}_\Sigma\ \tau \vee \ldots
\end{aligned}$$

$\boxed{\texttt{copyable}_\Sigma\ \tau}$

$$\texttt{copyable}_\Sigma\ \tau = \neg\, \texttt{noncopyable}_\Sigma\ \tau$$

$\boxed{r_1\ \texttt{occurs before}\ r_2\ \texttt{in}\ \Gamma}$

OC-OccursBase
$$\frac{r_1 \in \text{dom}(\Gamma)}{r_1\ \texttt{occurs before}\ r_2\ \texttt{in}\ \Gamma,\ r_2 \mapsto \{\, \overline{\ell}\, \}}$$

OC-OccursExtendFrame
$$\frac{r_1\ \texttt{occurs before}\ r_2\ \texttt{in}\ \Gamma}{r_1\ \texttt{occurs before}\ r_2\ \texttt{in}\ \Gamma, \mathcal{F}'}$$

OC-OccursNewFrame
$$\frac{r_1\ \texttt{occurs before}\ r_2\ \texttt{in}\ \Gamma}{r_1\ \texttt{occurs before}\ r_2\ \texttt{in}\ \Gamma \natural \mathcal{F}}$$

$\boxed{\text{gc-loans}(\Gamma)}$

$\text{gc-loans}(\Gamma) = \Gamma[\overline{r \mapsto \emptyset}]$
where $\overline{r} = \{\, r \in \text{dom}(\Gamma) \mid \forall \tau \in \text{cod}(\Gamma),\ r \text{ does not occur in } \tau \,\}$

$\boxed{\Gamma \Subset p = \Gamma'}$

$\Gamma \Subset p = \Gamma'$ where $\text{dom}(\Gamma) = \text{dom}(\Gamma')$ and
$\forall r.\ \Gamma'(r) = \{\ {}^{\omega}p' \in \Gamma(r) \mid p' \neq p^{\square}[*p]\ \}$ and
$\forall \pi.\ \Gamma(\pi) = \Gamma'(\pi)$

D DYNAMICS

Referent	$\mathcal{R}$	::=	$x \mid \mathcal{R}.n \mid \mathcal{R}[n] \mid \mathcal{R}[n_1..n_2]$
Referent Context	$\mathcal{R}^{\square}$	::=	$\square \mid \mathcal{R}^{\square}.n \mid \mathcal{R}^{\square}[n] \mid \mathcal{R}^{\square}[n_1..n_2]$
Expressions	e	::=	$\ldots \mid [\![v_1, \ldots, v_n]\!] \mid \texttt{dead} \mid \texttt{framed}\, e \mid \texttt{shift}\, e \mid \texttt{ptr}\, \mathcal{R}$
		$\mid$	$\langle \varsigma,\ \lvert x_1 : \tau_1^{\text{SI}}, \ldots, x_n : \tau_n^{\text{SI}} \rvert \rightarrow \tau_r^{\text{SI}}\ \{\ e\ \}\ \rangle$
Values	v	::=	$c \mid (v_1, \ldots, v_n) \mid [v_1, \ldots, v_n] \mid [\![v_1, \ldots, v_n]\!] \mid f \mid \texttt{dead} \mid \texttt{ptr}\, \mathcal{R}$
		$\mid$	$\langle \varsigma,\ \lvert x_1 : \tau_1^{\text{SI}}, \ldots, x_n : \tau_n^{\text{SI}} \rvert \rightarrow \tau_r^{\text{SI}}\ \{\ e\ \}\ \rangle$
Eval. Contexts	C	::=	$\square$
		$\mid$	$\&\rho\ \omega\ p[C] \mid \&\rho\ \omega\ p[C..\hat{e}] \mid \&\rho\ \omega\ p[v..C]$
		$\mid$	$\texttt{let}\, x : \tau^{\text{SI}} = C;\ e \mid \texttt{letprov} <r> \{\ C\ \}$
		$\mid$	$p \coloneqq C \mid C;\ e \mid \texttt{framed}\, C$
		$\mid$	$\texttt{shift}\, C \mid \texttt{shiftprov}\, C$
		$\mid$	$C ::< \overline{\Phi}, \overline{\rho}, \overline{\tau^{\text{SI}}} > (\hat{e}_1, \ldots, \hat{e}_n)$
		$\mid$	$v ::< \overline{\Phi}, \overline{\rho}, \overline{\tau^{\text{SI}}} > (v_1, \ldots, v_m, C, \hat{e}_1, \ldots, \hat{e}_n)$
		$\mid$	$p[C] \mid \texttt{if}\ C\ \{\ e_1\ \}\ \texttt{else}\ \{\ e_2\ \}$
		$\mid$	$\texttt{for}\ x\ \texttt{in}\ C\ \{\ e\ \}$
		$\mid$	$(v_1, \ldots, v_m, C, \hat{e}_1, \ldots, \hat{e}_n)$
		$\mid$	$[v_1, \ldots, v_m, C, \hat{e}_1, \ldots, \hat{e}_n]$
Value Contexts	$\mathcal{V}$	::=	$\square \mid (v_1, \ldots, \mathcal{V}, \ldots, v_n) \mid [v_1, \ldots, \mathcal{V}_1, \ldots, \mathcal{V}_m, \ldots, v_n]$
Stacks	σ	::=	$\bullet \mid \sigma \natural \varsigma$
Stack Frame	ς	::=	$\bullet \mid \varsigma,\ x \mapsto v$

$\boxed{\Sigma; \Gamma \vdash \mathcal{R} : \tau^{\text{XI}}}$

$$\textsc{WF-RefId}\ \frac{\Gamma(x) = \tau^{\text{SI}}}{\Sigma; \Gamma \vdash x : \tau^{\text{SI}}} \qquad \textsc{WF-RefProjection}\ \frac{\Sigma; \Gamma \vdash \mathcal{R} : (\tau_0^{\text{SI}}, \ldots, \tau_i^{\text{SI}}, \ldots, \tau_n^{\text{SI}})}{\Sigma; \Gamma \vdash \mathcal{R}.i : \tau_i^{\text{SI}}}$$

$$\textsc{WF-RefIndexArray}\ \frac{\Sigma; \Gamma \vdash \mathcal{R} : [\tau^{\text{SI}}; n] \qquad 0 \leq i < n}{\Sigma; \Gamma \vdash \mathcal{R}[i] : \tau^{\text{SI}}}$$

$$\textsc{WF-RefIndexSlice}\ \frac{\Sigma; \Gamma \vdash \mathcal{R} : [\tau^{\text{SI}}]}{\Sigma; \Gamma \vdash \mathcal{R}[i] : \tau^{\text{SI}}} \qquad \textsc{WF-RefSliceArray}\ \frac{\Sigma; \Gamma \vdash \mathcal{R} : [\tau^{\text{SI}}; n] \qquad 0 \leq i \leq j < n}{\Sigma; \Gamma \vdash \mathcal{R}[i..j] : [\tau^{\text{SI}}]}$$

$$\textsc{WF-RefSliceSlice}\ \frac{\Sigma; \Gamma \vdash \mathcal{R} : [\tau^{\text{SI}}] \qquad i \leq j}{\Sigma; \Gamma \vdash \mathcal{R}[i..j] : [\tau^{\text{SI}}]}$$

$\boxed{\sigma \vdash \mathcal{R} \Downarrow \mathcal{V} \times v}$

$$\textsc{ER-Id}\ \frac{\sigma(x) = v}{\sigma \vdash x \Downarrow \square \times v} \qquad \textsc{ER-Projection}\ \frac{\sigma \vdash \mathcal{R} \Downarrow \mathcal{V} \times (v_0, \ldots, v_i, \ldots, v_n)}{\sigma \vdash \mathcal{R}.i \Downarrow \mathcal{V}[(v_0, \ldots, \square, \ldots, v_n)] \times v_i}$$

$$\textsc{ER-IndexArray}\ \frac{\sigma \vdash \mathcal{R} \Downarrow \mathcal{V} \times [v_0, \ldots, v_i, \ldots, v_n]}{\sigma \vdash \mathcal{R}[i] \Downarrow \mathcal{V}[[v_0, \ldots, \square, \ldots, v_n]] \times v_i}$$

$$\textsc{ER-IndexSlice}\ \frac{\sigma \vdash \mathcal{R} \Downarrow \mathcal{V} \times [\![v_0, \ldots, v_k, \ldots, v_n]\!]}{\sigma \vdash \mathcal{R}[k] \Downarrow \mathcal{V}[v_0]\ \ldots\ [\square]\ \ldots\ [v_n] \times v_{i+k}} \qquad \textsc{ER-SliceArray}\ \frac{\sigma \vdash \mathcal{R} \Downarrow \mathcal{V} \times [v_0, \ldots, v_i, \ldots, v_j, \ldots, v_n]}{\sigma \vdash \mathcal{R}[i..j] \Downarrow \mathcal{V}[[v_0, \ldots, \square^{j-i+1}, \ldots, v_n]] \times [\![v_i, \ldots, v_j]\!]}$$

$$\textsc{ER-SliceSlice}\ \frac{\sigma \vdash \mathcal{R} \Downarrow \mathcal{V} \times [\![v_0, \ldots, v_i, \ldots, v_j, \ldots, v_n]\!]}{\sigma \vdash \mathcal{R}[i..j] \Downarrow \mathcal{V}[v_0]\ \ldots\ [\square]\ \ldots\ [\square]\ \ldots\ [v_n] \times [\![v_i, \ldots, v_j]\!]}$$

$\boxed{\sigma \vdash p \Downarrow \mathcal{R} \mapsto v}$
read: "p computes to $\mathcal{R}$, which maps to v in σ."

Let $\sigma \vdash p^{\square}[x] \Downarrow \mathcal{R} \mapsto v = \sigma \vdash p^{\square} \times x \Downarrow \mathcal{R} \mapsto v$.

$\boxed{\sigma \vdash p \Downarrow \mathcal{V}}$
read: "p computes to a value in σ with the context $\mathcal{V}$"

Let $\sigma \vdash p^{\square}[x] \Downarrow \mathcal{V} = \sigma \vdash p^{\square} \times x \Downarrow \mathcal{V} \times v$.

$\boxed{\sigma \vdash p^{\square} \times \mathcal{R} \Downarrow \mathcal{R}' \mapsto v}$
read: "$\mathcal{R}$ in a context $p^{\square}$ computes to $\mathcal{R}'$ which maps to v in σ."

P-Referent
$$\frac{\sigma \vdash \mathcal{R} \Downarrow \_ \times v}{\sigma \vdash \square \times \mathcal{R} \Downarrow \mathcal{R} \mapsto v}$$

P-Proj
$$\frac{\sigma \vdash p^{\square} \times \mathcal{R}_1 \Downarrow \mathcal{R}_2 \mapsto (v_0, \ldots, v_i, \ldots, v_n)}{\sigma \vdash p^{\square}[\square.i] \times \mathcal{R}_1 \Downarrow \mathcal{R}_2.i \mapsto v_i}$$

P-DerefPtr
$$\frac{\sigma \vdash \square \times \mathcal{R}_1 \Downarrow \_ \mapsto \mathtt{ptr}\ \pi \qquad \sigma \vdash p^{\square} \times \pi \Downarrow \mathcal{R}_2 \mapsto v}{\sigma \vdash p^{\square}[*\square] \times \mathcal{R}_1 \Downarrow \mathcal{R}_2 \mapsto v}$$

P-DerefIndexPtrArray
$$\frac{\sigma \vdash \square \times \mathcal{R}_1 \Downarrow \_ \mapsto \mathtt{ptr}\ \mathcal{R}_2[i] \qquad \sigma \vdash p^{\square} \times \mathcal{R}_2 \Downarrow \mathcal{R}_3 \mapsto [v_0, \ldots, v_i, \ldots, v_n]}{\sigma \vdash p^{\square}[*\square] \times \mathcal{R}_1 \Downarrow \mathcal{R}_3[i] \mapsto v_i}$$

P-DerefIndexPtrSlice
$$\frac{\sigma \vdash \square \times \mathcal{R}_1 \Downarrow \_ \mapsto \mathtt{ptr}\ \mathcal{R}_2[i] \qquad \sigma \vdash p^{\square} \times \mathcal{R}_2 \Downarrow \mathcal{R}_3 \mapsto [\![v_0, \ldots, v_i, \ldots, v_n]\!]}{\sigma \vdash p^{\square}[*\square] \times \mathcal{R}_1 \Downarrow \mathcal{R}_3[i] \mapsto v_i}$$

P-DerefSlicePtrArray
$$\frac{\sigma \vdash \square \times \mathcal{R}_1 \Downarrow \_ \mapsto \mathtt{ptr}\ \mathcal{R}_2[i..j] \qquad \sigma \vdash p^{\square} \times \mathcal{R}_2 \Downarrow \mathcal{R}_3 \mapsto [v_0, \ldots, v_i, \ldots, v_j, \ldots, v_n]}{\sigma \vdash p^{\square}[*\square] \times \mathcal{R}_1 \Downarrow \mathcal{R}_3[i..j] \mapsto [\![v_i, \ldots, v_j]\!]}$$

P-DerefSlicePtrSlice
$$\frac{\sigma \vdash \square \times \mathcal{R}_1 \Downarrow \_ \mapsto \mathtt{ptr}\ \mathcal{R}_2[i..j] \qquad \sigma \vdash p^{\square} \times \mathcal{R}_2 \Downarrow \mathcal{R}_3 \mapsto [\![v_0, \ldots, v_i, \ldots, v_j, \ldots, v_n]\!]}{\sigma \vdash p^{\square}[*\square] \times \mathcal{R}_1 \Downarrow \mathcal{R}_3[i..j] \mapsto [\![v_i, \ldots, v_j]\!]}$$

$\boxed{\sigma \vdash p^{\square} \times \mathcal{R} \Downarrow \mathcal{V} \times v}$
read: "$\mathcal{R}$ in a context $p^{\square}$ computes to a value in σ with the context $\mathcal{V}$"

PC-Referent
$$\frac{\sigma \vdash \mathcal{R} \Downarrow \mathcal{V} \times v}{\sigma \vdash \square \times \mathcal{R} \Downarrow \mathcal{V} \times v}$$

PC-Proj
$$\frac{\sigma \vdash p^{\square} \times \mathcal{R} \Downarrow \mathcal{V} \times (v_0, \ldots, v_i, \ldots, v_n)}{\sigma \vdash p^{\square}[\square.i] \times \mathcal{V} \Downarrow \mathcal{V}[(v_0, \ldots, \square, \ldots, v_n)] \times v_i}$$

PC-DerefPtr
$$\frac{\sigma \vdash \square \times \mathcal{R}_1 \Downarrow \_ \times \mathtt{ptr}\ \pi \qquad \sigma \vdash p^{\square} \times \pi \Downarrow \mathcal{V} \times v}{\sigma \vdash p^{\square}[*\square] \times \mathcal{R}_1 \Downarrow \mathcal{V} \times v}$$

PC-DerefIndexPtrArray
$$\frac{\sigma \vdash \square \times \mathcal{R}_1 \Downarrow \_ \times \mathtt{ptr}\ \mathcal{R}_2[i] \qquad \sigma \vdash p^{\square} \times \mathcal{R}_2 \Downarrow \mathcal{V} \times [v_0, \ldots, v_i, \ldots, v_n]}{\sigma \vdash p^{\square}[*\square] \times \mathcal{R}_1 \Downarrow \mathcal{V}[[v_0, \ldots, \square, \ldots, v_n]] \times v_i}$$

PC-DerefIndexPtrSlice
$$\frac{\sigma \vdash \square \times \mathcal{R}_1 \Downarrow \_ \times \mathtt{ptr}\ \mathcal{R}_2[i] \qquad \sigma \vdash p^{\square} \times \mathcal{R}_2 \Downarrow \mathcal{V} \times [\![v_0, \ldots, v_i, \ldots, v_n]\!]}{\sigma \vdash p^{\square}[*\square] \times \mathcal{R}_1 \Downarrow \mathcal{V}[v_0] \ \ldots\ [\square]\ \ldots\ [v_n] \times v_i}$$

PC-DerefSlicePtrArray
$$\frac{\sigma \vdash \square \times \mathcal{R}_1 \Downarrow \_ \times \mathtt{ptr}\ \mathcal{R}_2[i..j] \qquad \sigma \vdash p^{\square} \times \mathcal{R}_2 \Downarrow \mathcal{V} \times [v_0, \ldots, v_i, \ldots, v_j, \ldots, v_n]}{\sigma \vdash p^{\square}[(*\square)] \times \mathcal{R}_1 \Downarrow \mathcal{V}[[v_0, \ldots, \square^{j-i+1}, \ldots, v_n]] \times [\![v_i, \ldots, v_j]\!]}$$

PC-DerefSlicePtrSlice
$$\frac{\sigma \vdash \square \times \mathcal{R}_1 \Downarrow \_ \times \mathtt{ptr}\ \mathcal{R}_2[i..j] \qquad \sigma \vdash p^{\square} \times \mathcal{R}_2 \Downarrow \mathcal{V} \times [\![v_0, \ldots, v_i, \ldots, v_j, \ldots, v_n]\!]}{\sigma \vdash p^{\square}[(*\square)] \times \mathcal{R}_1 \Downarrow \mathcal{V}[v_0] \ \ldots\ [\square]\ \ldots\ [\square]\ \ldots\ [v_n] \times [\![v_i, \ldots, v_j]\!]}$$

$\boxed{\Sigma \vdash \sigma : \Gamma}$
read: "σ satisfies Γ under global context Σ"

WF-StackEmpty
$$\frac{}{\Sigma \vdash \bullet : \bullet}$$

WF-StackFrame
$$\frac{\Sigma \vdash \sigma : \Gamma \qquad \mathrm{dom}(\varsigma) = \mathrm{dom}(\mathcal{F})|_x \qquad \forall x \in \mathrm{dom}(\varsigma).\ \Sigma;\ \bullet;\ \Gamma \natural \mathcal{F} \vdash \boxed{(\sigma \natural \varsigma)(x)} : (\Gamma \natural \mathcal{F})(x) \Rightarrow \Gamma \natural \mathcal{F}}{\Sigma \vdash \sigma \natural \varsigma : \Gamma \natural \mathcal{F}}$$

$\boxed{\Sigma;\ \Gamma \vdash \varsigma : \mathcal{F}_c}$
read: "ς satisfies $\mathcal{F}_c$ under Σ and Γ"

WF-Frame
$$\frac{\mathrm{dom}(\varsigma) = \mathrm{dom}(\mathcal{F}_c)|_x \qquad \forall x \in \mathrm{dom}(\varsigma).\ \Sigma;\ \bullet;\ \Gamma \natural \mathcal{F}_c \vdash \boxed{\varsigma(x)} : \mathcal{F}_c(x) \Rightarrow \Gamma \natural \mathcal{F}_c}{\Sigma;\ \Gamma \vdash \varsigma : \mathcal{F}_c}$$

$\boxed{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e} : \tau \Rightarrow \Gamma'}$ where $\vdash \Sigma;\ \Delta;\ \Gamma$ and $\Sigma;\ \Delta;\ \Gamma' \vdash \tau$

...

T-Shift
$$\frac{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e} : \tau^{\mathrm{SI}} \Rightarrow \Gamma',\ x\ :\ \tau^{\mathrm{SD}}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\texttt{shift}\ e} : \tau^{\mathrm{SI}} \Rightarrow \Gamma'}$$

T-Framed
$$\frac{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e} : \tau^{\mathrm{SI}} \Rightarrow \Gamma' \natural \mathcal{F}'}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\texttt{framed}\ e} : \tau^{\mathrm{SI}} \Rightarrow \Gamma'}$$

T-Pointer
$$\frac{\Sigma;\ \Gamma \vdash \mathcal{R}^{\square}[\pi] : \tau^{\mathrm{XI}} \qquad {}^{\omega}\pi\ \in \Gamma(r)}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\texttt{ptr}\ \mathcal{R}^{\square}[\pi]} : \&r\ \omega\ \tau^{\mathrm{XI}} \Rightarrow \Gamma}$$

T-ClosureValue
$$\frac{\text{free-vars}(e) \setminus \overline{x} = \overline{x_f} = \mathrm{dom}(\mathcal{F}_c)|_x \qquad \overline{r} = \overline{\text{free-provs}(\Gamma(x_f))},\ \text{free-provs}(e) = \mathrm{dom}(\mathcal{F}_c)|_r \qquad \Sigma;\ \Gamma \vdash \varsigma_c : \mathcal{F}_c \qquad \Sigma;\ \Delta;\ \Gamma \natural \mathcal{F}_c,\ x_1\ :\ \tau_1^{\mathrm{SI}},\ \ldots,\ x_n\ :\ \tau_n^{\mathrm{SI}} \vdash \boxed{e} : \tau_r^{\mathrm{SI}} \Rightarrow \Gamma' \natural \mathcal{F}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\langle \varsigma_c,\ |x_1 : \tau_1^{\mathrm{SI}},\ \ldots,\ x_n : \tau_n^{\mathrm{SI}}|\ \rightarrow\ \tau_r^{\mathrm{SI}}\ \{\ e\ \}\ \rangle} : (\tau_1^{\mathrm{SI}},\ \ldots,\ \tau_n^{\mathrm{SI}}) \xrightarrow{\mathcal{F}_c} \tau_r^{\mathrm{SI}} \Rightarrow \Gamma}$$

T-Dead
$$\frac{}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{v} : \tau^{\mathrm{SI}^{\dagger}} \Rightarrow \Gamma}$$

$\boxed{\Sigma \vdash (\sigma;\ \boxed{e}) \rightarrow (\sigma';\ \boxed{e'})}$
read: "σ and e step to σ' and e' under Σ"

E-Move
$$\frac{\sigma \vdash \pi \Downarrow \_ \mapsto v}{\Sigma \vdash (\sigma;\ \boxed{\pi}) \rightarrow (\sigma[\pi \mapsto \mathsf{dead}];\ \boxed{v})}$$

E-Copy
$$\frac{\sigma \vdash p \Downarrow \_ \mapsto v}{\Sigma \vdash (\sigma;\ \boxed{p}) \rightarrow (\sigma;\ \boxed{v})}$$

E-Borrow
$$\frac{\sigma \vdash p \Downarrow \mathcal{R} \mapsto \_}{\Sigma \vdash (\sigma;\ \boxed{\&r\ \omega\ p}) \rightarrow (\sigma;\ \boxed{\texttt{ptr}\ \mathcal{R}})}$$

E-BorrowIndex
$$\frac{\sigma \vdash p \Downarrow \mathcal{R} \mapsto [v_0,\ \ldots,\ v_n] \qquad 0 \le n_i \le n}{\Sigma \vdash (\sigma;\ \boxed{\&r\ \omega\ p[n_i]}) \rightarrow (\sigma;\ \boxed{\texttt{ptr}\ \mathcal{R}[n_i]})}$$

E-BorrowSlice
$$\frac{\sigma \vdash p \Downarrow \mathcal{R} \mapsto [v_0,\ \ldots,\ v_n] \qquad 0 \le n_1 \le n_2 \le n}{\Sigma \vdash (\sigma;\ \boxed{\&r\ \omega\ p[n_1..n_2]}) \rightarrow (\sigma;\ \boxed{\texttt{ptr}\ \mathcal{R}[n_1..n_2]})}$$

E-BorrowIndexOOB
$$\frac{\sigma \vdash p \Downarrow \_ \mapsto [v_0,\ \ldots,\ v_n] \qquad n_i < 0 \vee n_i > n}{\Sigma \vdash (\sigma;\ \boxed{\&r\ \omega\ *p[n_i]}) \rightarrow (\sigma;\ \boxed{\texttt{abort!}(\text{"attempted to index out of bounds"})})}$$

E-BorrowSliceOOB
$$\frac{\sigma \vdash p \Downarrow \_ \mapsto [v_0, \ldots, v_n] \qquad n_1 < 0 \vee n_1 > n \vee n_2 < 0 \vee n_2 > n \vee n_1 > n_2}{\Sigma \vdash (\sigma; \boxed{\&r\ \omega\ p[n_1..n_2]}) \to (\sigma; \boxed{\texttt{abort!}(\text{“attempted to slice out of bounds”})})}$$

E-IndexCopy
$$\frac{\sigma \vdash p \Downarrow \_ \mapsto [v_0, \ldots, v_{n_i}, \ldots, v_n]}{\Sigma \vdash (\sigma; \boxed{p[n_i]}) \to (\sigma; \boxed{v_{n_i}})}$$

E-IndexCopyOOB
$$\frac{\sigma \vdash p \Downarrow \_ \mapsto [v_0, \ldots, v_n] \qquad n_i < 0 \vee n_i > n}{\Sigma \vdash (\sigma; \boxed{p[n_i]}) \to (\sigma; \boxed{\texttt{abort!}(\text{“attempted to index out of bounds”})})}$$

E-Framed
$$\frac{}{\Sigma \vdash (\sigma \natural \varsigma; \boxed{\texttt{framed}\ v}) \to (\sigma; \boxed{v})}$$

E-Shift
$$\frac{}{\Sigma \vdash (\sigma, x \mapsto v'; \boxed{\texttt{shift}\ v}) \to (\sigma; \boxed{v})}$$

E-IfTrue
$$\frac{}{\Sigma \vdash (\sigma; \boxed{\texttt{if true } \{ e_1 \} \texttt{ else } \{ e_2 \}}) \to (\sigma; \boxed{e_1})}$$

E-IfFalse
$$\frac{}{\Sigma \vdash (\sigma; \boxed{\texttt{if false } \{ e_1 \} \texttt{ else } \{ e_2 \}}) \to (\sigma; \boxed{e_2})}$$

E-LetProv
$$\frac{}{\Sigma \vdash (\sigma; \boxed{\texttt{letprov <}r\texttt{> } \{ v \}}) \to (\sigma; \boxed{v})}$$

E-Let
$$\frac{}{\Sigma \vdash (\sigma; \boxed{\texttt{let}\ x : \tau_a^{\text{SI}} = v; e}) \to (\sigma, x \mapsto v; \boxed{\texttt{shift}\ e})}$$

E-Seq
$$\frac{}{\Sigma \vdash (\sigma; \boxed{v; e}) \to (\sigma; \boxed{e})}$$

E-Assign
$$\frac{\sigma \vdash p \Downarrow \mathcal{V} \qquad p = p^{\square}[x]}{\Sigma \vdash (\sigma; \boxed{p := v}) \to (\sigma[x \mapsto \mathcal{V}[v]]; \boxed{()})}$$

E-While
$$\frac{}{\Sigma \vdash (\sigma; \boxed{\texttt{while}\ e_1 \{ e_2 \}}) \to (\sigma; \boxed{\texttt{if}\ e_1 \{ e_2; \texttt{while}\ e_1 \{ e_2 \} \} \texttt{ else } \{ () \}})}$$

E-ForArray
$$\frac{}{\Sigma \vdash (\sigma; \boxed{\texttt{for}\ x\ \texttt{in}\ [v_0, \ldots, v_n] \{ e \}}) \to (\sigma, x \mapsto v_0; \boxed{\texttt{shift}\ e; \texttt{for}\ x\ \texttt{in}\ [v_1, \ldots, v_n] \{ e \}})}$$

E-ForSlice
$$\frac{\sigma \vdash \mathcal{R} \Downarrow \_ \mapsto [v_1, \ldots, v_i, \ldots, v_j, \ldots, v_n] \qquad i < j \qquad i' = i + 1}{\Sigma \vdash (\sigma; \boxed{\texttt{for}\ x\ \texttt{in ptr}\ \mathcal{R}[i..j] \{ e \}}) \to (\sigma, x \mapsto \texttt{ptr}\ \mathcal{R}[i]; \boxed{\texttt{shift}\ e; \texttt{for}\ x\ \texttt{in ptr}\ \mathcal{R}[i'..j] \{ e \}})}$$

E-ForEmptyArray
$$\frac{}{\Sigma \vdash (\sigma; \boxed{\texttt{for}\ x\ \texttt{in}\ [] \{ e \}}) \to (\sigma; \boxed{()})}$$

E-ForEmptySlice
$$\frac{}{\Sigma \vdash (\sigma; \boxed{\texttt{for}\ x\ \texttt{in ptr}\ \pi[n..n] \{ e \}}) \to (\sigma; \boxed{()})}$$

E-Closure
$$\frac{\text{free-vars}(e) = \overline{x_f} \qquad \text{free-nc-vars}_\sigma(e) = \overline{x_{nc}} \qquad \varsigma_c = \sigma \mid_{\overline{x_f}}}{\Sigma \vdash (\sigma; \boxed{|x_1 : \tau_1^{\text{S}}, \ldots, x_n : \tau_n^{\text{S}}| \ \to\ \tau_r^{\text{S}} \{ e \}}) \to (\sigma[\overline{x_{nc} \mapsto \text{dead}}]; \boxed{\langle \varsigma_c, \ |x_1 : \tau_1^{\text{S}}, \ldots, x_n : \tau_n^{\text{S}}| \ \to\ \tau_r^{\text{S}} \{ e \} \rangle})}$$

E-AppClosure
$$\frac{v_f = \langle \varsigma_c, \ |x_1 : \tau_1^{\text{S}}, \ldots, x_n : \tau_n^{\text{S}}| \ \to\ \tau_r^{\text{S}} \{ e \} \rangle}{\Sigma \vdash (\sigma; \boxed{v_f(v_1, \ldots, v_n)}) \to (\sigma \natural \varsigma_c, x_1 \mapsto v_1, \ldots, x_n \mapsto v_n; \boxed{\texttt{framed}\ e})}$$

E-AppFunction
$$\frac{\Sigma(f) = \texttt{fn}\ f\texttt{<}\overline{\varphi}, \overline{\varrho}, \overline{\alpha}\texttt{>}(x_1 : \tau_1^{\text{S}}, \ldots, x_n : \tau_n^{\text{S}}) \ \to\ \tau_r^{\text{S}}\ \texttt{where}\ \overline{\varrho : \varrho'} \{ e \}}{\Sigma \vdash (\sigma; \boxed{f\texttt{::<}\overline{\Phi}, \overline{r'}, \overline{\tau^{\text{S}}}\texttt{>}(v_1, \ldots, v_n)}) \to (\sigma \natural x_1 \mapsto v_1, \ldots, x_n \mapsto v_n; \boxed{\texttt{framed}\ e[\overline{\Phi}/\overline{\varphi}][\overline{r'}/\overline{\varrho}][\overline{\tau^{\text{S}}}/\overline{\alpha}]})}$$

E-EvalCtx
$$\frac{\Sigma \vdash (\sigma; \boxed{e}) \to (\sigma'; \boxed{e'})}{\Sigma \vdash (\sigma; \boxed{C[e]}) \to (\sigma'; \boxed{C[e']})}$$

E-EvalCtxAbort
$$\frac{}{\Sigma \vdash (\sigma; \boxed{C[\texttt{abort!}(\texttt{str})]}) \to (\sigma; \boxed{\texttt{abort!}(\texttt{str})})}$$

E METATHEORY

E.1 Supporting Lemmas

Lemma E.1 (Canonical Forms). *If* $\Sigma; \Delta; \Gamma \vdash \boxed{v} : \tau \Rightarrow \Gamma$ *then*

(1) *if* $\tau = \texttt{bool}$, *then* $v = \texttt{true}$ *or* $v = \texttt{false}$.
(2) *if* $\tau = \texttt{u32}$, *then* $v = n$.
(3) *if* $\tau = \texttt{unit}$, *then* $v = ()$.
(4) *if* $\tau = \&\rho\ \omega\ \tau^{SI}$, *then* v *is of the form* $\texttt{ptr}\ \mathcal{R}$.
(5) *if* $\tau = \&\rho\ \omega\ [\tau^{SI}]$, *then* v *is of the form* $\texttt{ptr}\ \mathcal{R}[i..j]$.
(6) *if* $\tau = \forall <\overline{\varphi}, \overline{\varrho}, \overline{\alpha}> (\tau_1^{SI}, \ldots, \tau_n^{SI}) \rightarrow \tau_r^{SI}\ \texttt{where}\ \overline{\varrho_1 : \varrho_2}$, *then* v *is of the form* f.
(7) *if* $\tau = (\tau_1^{SI}, \ldots, \tau_n^{SI}) \xrightarrow{\mathcal{F}} \tau_r^{SI}$, *then* v *is of the form* $\langle \sigma, \ |x_1 : \tau_1^{SI}, \ldots, x_n : \tau_n^{SI}| \rightarrow \tau_r^{SI}\ \{\ e\ \}\ \rangle$.
(8) *if* $\tau = [\tau'; n]$, *then* v *is of the form* $[v_1, \ldots, v_n]$.
(9) *if* $\tau = [\tau']$, *then* v *is of the form* $[\![v_1, \ldots, v_n]\!]$.
(10) *if* $\tau = (\tau_1, \ldots, \tau_n)$, *then* v *is of the form* $(v_1, \ldots, v_n)$.

Proof. By inspection of the grammar of values and typing rules. □

Lemma E.2 (Well-Formed References Evaluate to Well-Typed Values). *If* $\Sigma; \Gamma \vdash \mathcal{R} : \tau^{XI}$ *and* $\Sigma \vdash \sigma : \Gamma$, *then* $\sigma \vdash \mathcal{R} \Downarrow \mathcal{V} \times v$.

Proof. We proceed by induction on $\Sigma; \Gamma \vdash \mathcal{R} : \tau^{\text{XI}}$. There are six cases: WF-RefId, WF-RefProj, WF-RefIndexArray, WF-RefIndexSlice, WF-RefSliceArray, and WF-RefSliceSlice. Each of these cases has a corresponding evaluation rule:

$$\frac{\Gamma(x) = \tau^{\text{SI}}}{\Sigma; \Gamma \vdash x : \tau^{\text{SI}}}\ \text{WF-RefId} \qquad \frac{\sigma(x) = v}{\sigma \vdash x \Downarrow \square \times v}\ \text{ER-Id}$$

For the base case, we consider the frame of Γ which contains x. By inversion of WF-StackFrame for the portion of the derivation $\Sigma \vdash \sigma : \Gamma$ pertaining to that frame, we have $\forall x \in \text{dom}(\varsigma).\ \Sigma; \bullet; \Gamma \natural \mathcal{F} \vdash \boxed{(\sigma \natural \varsigma)(x)} : (\Gamma \natural \mathcal{F})(x) \Rightarrow \Gamma \natural \mathcal{F}$. Focusing on our particular x, we have both that $\sigma(x) = v$ and that $\Sigma; \bullet; \Gamma \natural \mathcal{F} \vdash \boxed{v} : (\Gamma \natural \mathcal{F})(x) \Rightarrow \Gamma \natural \mathcal{F}$, finishing the case. The remaining cases follow:

$$\frac{\Sigma; \Gamma \vdash \mathcal{R} : (\tau_0^{\text{SI}}, \ldots, \tau_i^{\text{SI}}, \ldots, \tau_n^{\text{SI}})}{\Sigma; \Gamma \vdash \mathcal{R}.i : \tau_i^{\text{SI}}}\ \text{WF-RefProjection} \qquad \frac{\sigma \vdash \mathcal{R} \Downarrow \mathcal{V} \times (v_0, \ldots, v_i, \ldots, v_n)}{\sigma \vdash \mathcal{R}.i \Downarrow \mathcal{V}[(v_0, \ldots, \square, \ldots, v_n)] \times v_i}\ \text{ER-Projection}$$

$$\frac{\Sigma; \Gamma \vdash \mathcal{R} : [\tau^{\text{SI}}; n] \qquad 0 \leq i < n}{\Sigma; \Gamma \vdash \mathcal{R}[i] : \tau^{\text{SI}}}\ \text{WF-RefIndexArray} \qquad \frac{\sigma \vdash \mathcal{R} \Downarrow \mathcal{V} \times [v_0, \ldots, v_i, \ldots, v_n]}{\sigma \vdash \mathcal{R}[i] \Downarrow \mathcal{V}[[v_0, \ldots, \square, \ldots, v_n]] \times v_i}\ \text{ER-IndexArray}$$

$$\frac{\Sigma; \Gamma \vdash \mathcal{R} : [\tau^{\text{SI}}]}{\Sigma; \Gamma \vdash \mathcal{R}[i] : \tau^{\text{SI}}}\ \text{WF-RefIndexSlice} \qquad \frac{\sigma \vdash \mathcal{R} \Downarrow \mathcal{V} \times [\![v_0, \ldots, v_k, \ldots, v_n]\!]}{\sigma \vdash \mathcal{R}[k] \Downarrow \mathcal{V}[v_0] \ \ldots\ [\square] \ \ldots\ [v_n] \times v_{i+k}}\ \text{ER-IndexSlice}$$

$$\frac{\Sigma; \Gamma \vdash \mathcal{R} : [\tau^{\text{SI}}; n] \qquad 0 \leq i \leq j < n}{\Sigma; \Gamma \vdash \mathcal{R}[i..j] : [\tau^{\text{SI}}]}\ \text{WF-RefSliceArray} \qquad \frac{\sigma \vdash \mathcal{R} \Downarrow \mathcal{V} \times [v_0, \ldots, v_i, \ldots, v_j, \ldots, v_n]}{\sigma \vdash \mathcal{R}[i..j] \Downarrow \mathcal{V}[[v_0, \ldots, \square^{j-i+1}, \ldots, v_n]] \times [\![v_i, \ldots, v_j]\!]}\ \text{ER-SliceArray}$$

$$\textsc{WF-RefSliceSlice}\ \frac{\Sigma;\ \Gamma \vdash \mathcal{R} : [\tau^{\text{SI}}] \qquad i \le j}{\Sigma;\ \Gamma \vdash \mathcal{R}[i..j] : [\tau^{\text{SI}}]}$$

$$\textsc{ER-SliceSlice}\ \frac{\sigma \vdash \mathcal{R} \Downarrow \mathcal{V} \times [\![v_0, \ldots, v_i, \ldots, v_j, \ldots, v_n]\!]}{\sigma \vdash \mathcal{R}[i..j] \Downarrow \mathcal{V}[v_0]\ \ldots\ [\square]\ \ldots\ [\square]\ \ldots\ [v_n] \times [\![v_i, \ldots, v_j]\!]}$$

The proof for each case is identical: apply the induction hypothesis and then Lemma E.1 and then the evaluation rule on the right. For the well-typed portion, apply inversion on the typing rule for the appropriate value. □

Lemma E.3 (Place Expressions Reduce). *If* $\Delta;\ \Gamma \vdash_\omega p : \tau^{\text{XI}}$ *and* $\Sigma \vdash \sigma : \Gamma$, *then* $\sigma \vdash p \Downarrow \mathcal{R} \mapsto v$ *and* $\Sigma;\ \Delta;\ \Gamma \vdash \boxed{v} : \tau^{\text{XI}} \Rightarrow \Gamma$.

Proof. We proceed by induction on $\Delta;\ \Gamma \vdash_\omega p : \tau^{\text{XI}}$. There are three cases: TC-Var, TC-Proj, and TC-Deref.

$$\textsc{TC-Var}\ \frac{\Gamma(x) = \tau^{\text{SI}}}{\Delta;\ \Gamma \vdash_\omega x : \tau^{\text{SI}},\ \emptyset} \qquad \textsc{P-Referent}\ \frac{\sigma \vdash \mathcal{R} \Downarrow \_ \times v}{\sigma \vdash \square \times \mathcal{R} \Downarrow \mathcal{R} \mapsto v}$$

For TC-Var, we consider the piece of the derivation for $\Sigma \vdash \sigma : \Gamma$ (from our premise) for the frame containing x. By inversion on WF-StackFrame, we have $\forall x \in \text{dom}(\varsigma).\ \Sigma;\ \bullet;\ \Gamma \natural \mathcal{F} \vdash \boxed{(\sigma \natural \varsigma)(x)} : (\Gamma \natural \mathcal{F})(x) \Rightarrow \Gamma \natural \mathcal{F}$. This immediately gives us that $\sigma(x) = v$ and that $\Sigma;\ \Delta;\ \Gamma \vdash \boxed{v} : \tau^{\text{XI}} \Rightarrow \Gamma$. To construct our premise for P-Referent, we apply ER-Id to $\sigma(x) = v$.

$$\textsc{TC-Proj}\ \frac{\Delta;\ \Gamma \vdash_\omega p : (\tau_1^{\text{SI}}, \ldots, \tau_i^{\text{SI}}, \ldots, \tau_n^{\text{SI}}),\ \{\ \overline{\rho_p}\ \}}{\Delta;\ \Gamma \vdash_\omega p.i : \tau_i^{\text{SI}},\ \{\ \overline{\rho_p}\ \}} \qquad \textsc{P-Proj}\ \frac{\sigma \vdash p^{\square} \times \mathcal{R}_1 \Downarrow \mathcal{R}_2 \mapsto (v_0, \ldots, v_i, \ldots, v_n)}{\sigma \vdash p^{\square}[\square.i] \times \mathcal{R}_1 \Downarrow \mathcal{R}_2.i \mapsto v_i}$$

For TC-Proj, we apply our induction hypothesis to $\Delta;\ \Gamma \vdash_\omega p : (\tau_1^{\text{SI}}, \ldots, \tau_i^{\text{SI}}, \ldots, \tau_n^{\text{SI}}),\ \{\ \overline{\rho_p}\ \}$ from the premise of TC-Proj and get $\sigma \vdash p \Downarrow \mathcal{R} \mapsto v$ and $\Sigma;\ \Delta;\ \Gamma \vdash \boxed{v} : (\tau_1^{\text{SI}}, \ldots, \tau_i^{\text{SI}}, \ldots, \tau_n^{\text{SI}}) \Rightarrow \Gamma$. Then, by Lemma E.1, we know that v must be of the form $(v_1, \ldots, v_i, \ldots, v_n)$. We can use this and the definition of $\sigma \vdash p \Downarrow \mathcal{R} \mapsto v$ to get $\sigma \vdash p^{\square} \times x \Downarrow \mathcal{R} \mapsto (v_1, \ldots, v_i, \ldots, v_n)$ (where $p^{\square}[x] = p$). This is precisely the premise of P-Proj and thus we can use that. We also have by inversion of T-Tuple for $\Sigma;\ \Delta;\ \Gamma \vdash \boxed{v} : (\tau_1^{\text{SI}}, \ldots, \tau_i^{\text{SI}}, \ldots, \tau_n^{\text{SI}}) \Rightarrow \Gamma$ that $\Sigma;\ \Delta;\ \Gamma \vdash \boxed{v_i} : \tau_i^{\text{SI}} \Rightarrow \Gamma$.

$$\textsc{TC-Deref}\ \frac{\Delta;\ \Gamma \vdash_\omega p : \&\rho\ \omega'\ \tau^{\text{XI}},\ \{\ \overline{\rho_p}\ \} \qquad \omega \lesssim \omega' \qquad \Delta;\ \Gamma \vdash \overline{\rho :> \rho_p} \Rightarrow \Gamma_f}{\Delta;\ \Gamma \vdash_\omega *p : \tau^{\text{XI}},\ \{\ \overline{\rho_p},\ \rho\ \}}$$

For TC-Deref, we apply our induction hypothesis to $\Delta;\ \Gamma \vdash_\omega p : \&\rho\ \omega'\ \tau^{\text{XI}},\ \{\ \overline{\rho_p}\ \}$ to get $\sigma \vdash p \Downarrow \mathcal{R} \mapsto v$ and $\Sigma;\ \Delta;\ \Gamma \vdash \boxed{v} : \&\rho\ \omega'\ \tau^{\text{XI}} \Rightarrow \Gamma$. Then, by Lemma E.1, we know that v must of the form $\texttt{ptr}\ \mathcal{R}$. We now have five subcases to consider depending on whether $\mathcal{R}$ is of π, $\mathcal{R}_3[i]$, or $\mathcal{R}[i..j]$, and for the latter two, whether τ^{XI} is $[\tau^{\text{SI}};\ n]$ or $[\tau^{\text{SI}}]$.

$$\textsc{P-DerefPtr}\ \frac{\sigma \vdash \square \times \mathcal{R}_1 \Downarrow \_ \mapsto \texttt{ptr}\ \pi \qquad \sigma \vdash p^{\square} \times \pi \Downarrow \mathcal{R}_2 \mapsto v}{\sigma \vdash p^{\square}[*\square] \times \mathcal{R}_1 \Downarrow \mathcal{R}_2 \mapsto v}$$

$$\textsc{P-DerefIndexPtrArray}\ \frac{\sigma \vdash \square \times \mathcal{R}_1 \Downarrow \_ \mapsto \texttt{ptr}\ \mathcal{R}_2[i] \qquad \sigma \vdash p^{\square} \times \mathcal{R}_2 \Downarrow \mathcal{R}_3 \mapsto [v_0, \ldots, v_i, \ldots, v_n]}{\sigma \vdash p^{\square}[*\square] \times \mathcal{R}_1 \Downarrow \mathcal{R}_3[i] \mapsto v_i}$$

$$\textsc{P-DerefIndexPtrSlice}\ \frac{\sigma \vdash \square \times \mathcal{R}_1 \Downarrow \_ \mapsto \texttt{ptr}\ \mathcal{R}_2[i] \qquad \sigma \vdash p^{\square} \times \mathcal{R}_2 \Downarrow \mathcal{R}_3 \mapsto [\![v_0, \ldots, v_i, \ldots, v_n]\!]}{\sigma \vdash p^{\square}[*\square] \times \mathcal{R}_1 \Downarrow \mathcal{R}_3[i] \mapsto v_i}$$

$$\textsc{P-DerefSlicePtrArray}\ \frac{\sigma \vdash \square \times \mathcal{R}_1 \Downarrow \_ \mapsto \texttt{ptr}\ \mathcal{R}_2[i..j] \qquad \sigma \vdash p^{\square} \times \mathcal{R}_2 \Downarrow \mathcal{R}_3 \mapsto [v_0, \ldots, v_i, \ldots, v_j, \ldots, v_n]}{\sigma \vdash p^{\square}[*\square] \times \mathcal{R}_1 \Downarrow \mathcal{R}_3[i..j] \mapsto [\![v_i, \ldots, v_j]\!]}$$

$$\textsc{P-DerefSlicePtrSlice}\ \frac{\sigma \vdash \square \times \mathcal{R}_1 \Downarrow \_ \mapsto \texttt{ptr}\ \mathcal{R}_2[i..j] \qquad \sigma \vdash p^{\square} \times \mathcal{R}_2 \Downarrow \mathcal{R}_3 \mapsto [\![v_0, \ldots, v_i, \ldots, v_j, \ldots, v_n]\!]}{\sigma \vdash p^{\square}[*\square] \times \mathcal{R}_1 \Downarrow \mathcal{R}_3[i..j] \mapsto [\![v_i, \ldots, v_j]\!]}$$

In all these cases, we know structurally that $p^{\square} = \square$ since TC-Deref has no context outside of the dereference. So, for each of them, we need to be able to show $\square \vdash \mathcal{R} \Downarrow \mathcal{R}' \mapsto v'$. Inversion on T-Pointer gives us $\Sigma; \Gamma \vdash \mathcal{R} : \tau^{\text{XI}}$. We can then apply Lemma E.2 to get $\Sigma \vdash \Gamma \Downarrow \mathcal{V} \times v$. Then, we can apply P-Referent to this to produce the derivation we need to apply the appropriate rule. For P-DerefIndexPtrArray and P-DerefSlicePtrArray, we apply Lemma E.1 to get that the value is an array. For P-DerefIndexPtrSlice and P-DerefSlicePtrSlice, we apply Lemma E.1 to get that the value is a slice value. □

Lemma E.4 (Reducible Place Expressions can also Compute a Context). *If* $\sigma \vdash p \Downarrow \mathcal{R} \mapsto v$ *and* $\mathcal{R} \neq \mathcal{R}[i..j]$, *then* $\sigma \vdash p \Downarrow \mathcal{V}$.

Proof. The proof proceeds by induction on $\sigma \vdash p \Downarrow \mathcal{R} \mapsto v$ by cases. Since the two judgments share an identical inductive structure, we essentially pair the corresponding rules as follows:

$$\textsc{P-Referent}\ \frac{\sigma \vdash \mathcal{R} \Downarrow \_ \times v}{\sigma \vdash \square \times \mathcal{R} \Downarrow \mathcal{R} \mapsto v} \qquad \textsc{PC-Referent}\ \frac{\sigma \vdash \mathcal{R} \Downarrow \mathcal{V} \times v}{\sigma \vdash \square \times \mathcal{R} \Downarrow \mathcal{V} \times v}$$

$$\textsc{P-Proj}\ \frac{\sigma \vdash p^{\square} \times \mathcal{R}_1 \Downarrow \mathcal{R}_2 \mapsto (v_0, \ldots, v_i, \ldots, v_n)}{\sigma \vdash p^{\square}[\square.i] \times \mathcal{R}_1 \Downarrow \mathcal{R}_2.i \mapsto v_i} \qquad \textsc{PC-Proj}\ \frac{\sigma \vdash p^{\square} \times \mathcal{R} \Downarrow \mathcal{V} \times (v_0, \ldots, v_i, \ldots, v_n)}{\sigma \vdash p^{\square}[\square.i] \times \mathcal{V} \Downarrow \mathcal{V}[(v_0, \ldots, \square, \ldots, v_n)] \times v_i}$$

$$\textsc{P-DerefPtr}\ \frac{\sigma \vdash \square \times \mathcal{R}_1 \Downarrow \_ \mapsto \texttt{ptr}\ \pi \qquad \sigma \vdash p^{\square} \times \pi \Downarrow \mathcal{R}_2 \mapsto v}{\sigma \vdash p^{\square}[*\square] \times \mathcal{R}_1 \Downarrow \mathcal{R}_2 \mapsto v} \qquad \textsc{PC-DerefPtr}\ \frac{\sigma \vdash \square \times \mathcal{R}_1 \Downarrow \_ \times \texttt{ptr}\ \pi \qquad \sigma \vdash p^{\square} \times \pi \Downarrow \mathcal{V} \times v}{\sigma \vdash p^{\square}[*\square] \times \mathcal{R}_1 \Downarrow \mathcal{V} \times v}$$

$$\textsc{P-DerefIndexPtrArray}\ \frac{\sigma \vdash \square \times \mathcal{R}_1 \Downarrow \_ \mapsto \texttt{ptr}\ \mathcal{R}_2[i] \qquad \sigma \vdash p^{\square} \times \mathcal{R}_2 \Downarrow \mathcal{R}_3 \mapsto [v_0, \ldots, v_i, \ldots, v_n]}{\sigma \vdash p^{\square}[*\square] \times \mathcal{R}_1 \Downarrow \mathcal{R}_3[i] \mapsto v_i} \qquad \textsc{PC-DerefIndexPtrArray}\ \frac{\sigma \vdash \square \times \mathcal{R}_1 \Downarrow \_ \times \texttt{ptr}\ \mathcal{R}_2[i] \qquad \sigma \vdash p^{\square} \times \mathcal{R}_2 \Downarrow \mathcal{V} \times [v_0, \ldots, v_i, \ldots, v_n]}{\sigma \vdash p^{\square}[*\square] \times \mathcal{R}_1 \Downarrow \mathcal{V}[[v_0, \ldots, \square, \ldots, v_n]] \times v_i}$$

$$\textsc{P-DerefIndexPtrSlice}\ \frac{\sigma \vdash \square \times \mathcal{R}_1 \Downarrow \_ \mapsto \texttt{ptr}\ \mathcal{R}_2[i] \qquad \sigma \vdash p^{\square} \times \mathcal{R}_2 \Downarrow \mathcal{R}_3 \mapsto [\![v_0, \ldots, v_i, \ldots, v_n]\!]}{\sigma \vdash p^{\square}[*\square] \times \mathcal{R}_1 \Downarrow \mathcal{R}_3[i] \mapsto v_i} \qquad \textsc{PC-DerefIndexPtrSlice}\ \frac{\sigma \vdash \square \times \mathcal{R}_1 \Downarrow \_ \times \texttt{ptr}\ \mathcal{R}_2[i] \qquad \sigma \vdash p^{\square} \times \mathcal{R}_2 \Downarrow \mathcal{V} \times [\![v_0, \ldots, v_i, \ldots, v_n]\!]}{\sigma \vdash p^{\square}[*\square] \times \mathcal{R}_1 \Downarrow \mathcal{V}[v_0] \ \ldots\ [\square] \ \ldots\ [v_n] \times v_i}$$

$$\textsc{P-DerefSlicePtrArray}\ \frac{\sigma \vdash \square \times \mathcal{R}_1 \Downarrow \_ \mapsto \texttt{ptr}\ \mathcal{R}_2[i..j] \qquad \sigma \vdash p^{\square} \times \mathcal{R}_2 \Downarrow \mathcal{R}_3 \mapsto [v_0, \ldots, v_i, \ldots, v_j, \ldots, v_n]}{\sigma \vdash p^{\square}[*\square] \times \mathcal{R}_1 \Downarrow \mathcal{R}_3[i..j] \mapsto [\![v_i, \ldots, v_j]\!]}$$

$$\textsc{PC-DerefSlicePtrArray}\ \frac{\sigma \vdash \square \times \mathcal{R}_1 \Downarrow \_ \times \texttt{ptr}\ \mathcal{R}_2[i..j] \qquad \sigma \vdash p^{\square} \times \mathcal{R}_2 \Downarrow \mathcal{V} \times [v_0, \ldots, v_i, \ldots, v_j, \ldots, v_n]}{\sigma \vdash p^{\square}[(*\square)] \times \mathcal{R}_1 \Downarrow \mathcal{V}[[v_0, \ldots, \square^{j-i+1}, \ldots, v_n]] \times [\![v_i, \ldots, v_j]\!]}$$

$$\textsc{P-DerefSlicePtrSlice}\ \frac{\sigma \vdash \square \times \mathcal{R}_1 \Downarrow \_ \mapsto \texttt{ptr}\ \mathcal{R}_2[i..j] \qquad \sigma \vdash p^{\square} \times \mathcal{R}_2 \Downarrow \mathcal{R}_3 \mapsto [\![v_0, \ldots, v_i, \ldots, v_j, \ldots, v_n]\!]}{\sigma \vdash p^{\square}[*\square] \times \mathcal{R}_1 \Downarrow \mathcal{R}_3[i..j] \mapsto [\![v_i, \ldots, v_j]\!]}$$

$$\textsc{PC-DerefSlicePtrSlice}\ \frac{\sigma \vdash \square \times \mathcal{R}_1 \Downarrow \_ \times \texttt{ptr}\ \mathcal{R}_2[i..j] \qquad \sigma \vdash p^{\square} \times \mathcal{R}_2 \Downarrow \mathcal{V} \times [\![v_0, \ldots, v_i, \ldots, v_j, \ldots, v_n]\!]}{\sigma \vdash p^{\square}[(*\square)] \times \mathcal{R}_1 \Downarrow \mathcal{V}[v_0] \ \ldots\ [\square] \ \ldots\ [\square] \ \ldots\ [v_n] \times [\![v_i, \ldots, v_j]\!]}$$

□

Lemma E.5 (Reduced Place Expressions Produce Valid Referents). *If* $\Sigma \vdash \sigma : \Gamma$ *and* $\sigma \vdash p \Downarrow \mathcal{R}^{\square}[\pi] \mapsto v$, *then* $\Sigma; \Gamma \vdash \mathcal{R}^{\square}[\pi] : \tau^{XI}$.

Proof. We start by rewriting $\sigma \vdash p \Downarrow \mathcal{R}^{\square}[\pi] \mapsto v$ with its definition to get $\sigma \vdash p^{\square} \times x \Downarrow \mathcal{R}^{\square}[\pi] \mapsto v$ where $p = p^{\square}[x]$. We then proceed by induction by cases (note this means our induction hypothesis is really about the rewritten form).

$$\textsc{P-Referent}\ \frac{\sigma \vdash \mathcal{R} \Downarrow \_ \times v}{\sigma \vdash \square \times \mathcal{R} \Downarrow \mathcal{R} \mapsto v} \qquad \textsc{WF-RefId}\ \frac{\Gamma(x) = \tau^{\text{SI}}}{\Sigma; \Gamma \vdash x : \tau^{\text{SI}}}$$

P-Referent only applies if the context is □ which is only the case if our original place expression was x. We can rewrite with this knowledge to see that we really have $\sigma \vdash x \Downarrow \_ \times v$ in our premise. Inversion on ER-Id gives us $\sigma(v) =$ Then, we consider the frame of Γ which contains x. By inversion of WF-StackFrame for the portion of the derivation $\Sigma \vdash \sigma : \Gamma$ pertaining to that frame, we have $\forall x \in \text{dom}(\varsigma).\ \Sigma; \bullet; \Gamma \natural \mathcal{F} \vdash \boxed{(\sigma \natural \varsigma)(x)} : (\Gamma \natural \mathcal{F})(x) \Rightarrow \Gamma \natural \mathcal{F}$. Focusing on our particular x, we have both that $\Gamma(x) = v$. We can then apply WF-RefId.

$$\textsc{P-Proj}\ \frac{\sigma \vdash p^{\square} \times \mathcal{R}_1 \Downarrow \mathcal{R}_2 \mapsto (v_0, \ldots, v_i, \ldots, v_n)}{\sigma \vdash p^{\square}[\square.i] \times \mathcal{R}_1 \Downarrow \mathcal{R}_2.i \mapsto v_i} \qquad \textsc{WF-RefProjection}\ \frac{\Sigma; \Gamma \vdash \mathcal{R} : (\tau_0^{\text{SI}}, \ldots, \tau_i^{\text{SI}}, \ldots, \tau_n^{\text{SI}})}{\Sigma; \Gamma \vdash \mathcal{R}.i : \tau_i^{\text{SI}}}$$

Applying the induction hypothesis to $\sigma \vdash p^{\square} \times \mathcal{R}_1 \Downarrow \mathcal{R}_2 \mapsto (v_0, \ldots, v_i, \ldots, v_n)$ gives us $\Sigma; \Gamma \vdash \mathcal{R}_2 : (\tau_0^{\text{SI}}, \ldots, \tau_i^{\text{SI}}, \ldots, \tau_n^{\text{SI}})$. We can then apply WF-RefProjection.

$$\textsc{P-DerefPtr}\ \frac{\sigma \vdash \square \times \mathcal{R}_1 \Downarrow \_ \mapsto \texttt{ptr}\ \pi \qquad \sigma \vdash p^{\square} \times \pi \Downarrow \mathcal{R}_2 \mapsto v}{\sigma \vdash p^{\square}[*\square] \times \mathcal{R}_1 \Downarrow \mathcal{R}_2 \mapsto v}$$

Applying the induction hypothesis to $\sigma \vdash p^{\square} \times \pi \Downarrow \mathcal{R}_2 \mapsto v$ gives us $\Sigma; \Gamma \vdash \mathcal{R}_2 : \tau^{\text{XI}}$.

$$\textsc{P-DerefIndexPtrArray}\ \frac{\sigma \vdash \square \times \mathcal{R}_1 \Downarrow \_ \mapsto \texttt{ptr}\ \mathcal{R}_2[i] \qquad \sigma \vdash p^{\square} \times \mathcal{R}_2 \Downarrow \mathcal{R}_3 \mapsto [v_0, \ldots, v_i, \ldots, v_n]}{\sigma \vdash p^{\square}[*\square] \times \mathcal{R}_1 \Downarrow \mathcal{R}_3[i] \mapsto v_i}$$

$$\textsc{WF-RefIndexArray}\ \frac{\Sigma; \Gamma \vdash \mathcal{R} : [\tau^{\text{SI}}; n] \qquad 0 \le i < n}{\Sigma; \Gamma \vdash \mathcal{R}[i] : \tau^{\text{SI}}}$$

Applying the induction hypothesis to $\sigma \vdash p^{\square} \times \mathcal{R}_2 \Downarrow \mathcal{R}_3 \mapsto [v_0, \ldots, v_i, \ldots, v_n]$ gives us $\Sigma; \Gamma \vdash \mathcal{R}_3 : [\tau^{\text{SI}}; n]$. Then, we can apply WF-RefIndexArray to get $\Sigma; \Gamma \vdash \mathcal{R}_3[i] : \tau^{\text{SI}}$.

$$\textsc{P-DerefIndexPtrSlice}\ \frac{\sigma \vdash \square \times \mathcal{R}_1 \Downarrow \_ \mapsto \texttt{ptr}\ \mathcal{R}_2[i] \qquad \sigma \vdash p^{\square} \times \mathcal{R}_2 \Downarrow \mathcal{R}_3 \mapsto [\![v_0, \ldots, v_i, \ldots, v_n]\!]}{\sigma \vdash p^{\square}[*\square] \times \mathcal{R}_1 \Downarrow \mathcal{R}_3[i] \mapsto v_i}$$

$$\textsc{WF-RefIndexSlice}\ \frac{\Sigma; \Gamma \vdash \mathcal{R} : [\tau^{\text{SI}}]}{\Sigma; \Gamma \vdash \mathcal{R}[i] : \tau^{\text{SI}}}$$

Applying the induction hypothesis to $\sigma \vdash p^{\square} \times \mathcal{R}_2 \Downarrow \mathcal{R}_3 \mapsto [\![v_0, \ldots, v_i, \ldots, v_n]\!]$ gives us $\Sigma; \Gamma \vdash \mathcal{R}_3 : [\tau^{\text{SI}}]$. Then, we can apply WF-RefIndexSlice to get $\Sigma; \Gamma \vdash \mathcal{R}_3[i] : \tau^{\text{SI}}$.

$$\textsc{P-DerefSlicePtrArray}\ \frac{\sigma \vdash \square \times \mathcal{R}_1 \Downarrow \_ \mapsto \texttt{ptr}\ \mathcal{R}_2[i..j] \qquad \sigma \vdash p^{\square} \times \mathcal{R}_2 \Downarrow \mathcal{R}_3 \mapsto [v_0, \ldots, v_i, \ldots, v_j, \ldots, v_n]}{\sigma \vdash p^{\square}[*\square] \times \mathcal{R}_1 \Downarrow \mathcal{R}_3[i..j] \mapsto [\![v_i, \ldots, v_j]\!]}$$

$$\textsc{WF-RefSliceArray}\ \frac{\Sigma; \Gamma \vdash \mathcal{R} : [\tau^{\text{SI}}; n] \qquad 0 \le i \le j < n}{\Sigma; \Gamma \vdash \mathcal{R}[i..j] : [\tau^{\text{SI}}]}$$

Applying the induction hypothesis to $\sigma \vdash p^{\square} \times \mathcal{R}_2 \Downarrow \mathcal{R}_3 \mapsto [v_0, \ldots, v_i, \ldots, v_j, \ldots, v_n]$ gives us $\Sigma; \Gamma \vdash \mathcal{R}_3 : [\tau^{\text{SI}}; n]$. Then, we can apply WF-RefSliceArray to get $\Sigma; \Gamma \vdash \mathcal{R}_3[i..j] : \tau^{\text{SI}}$.

$$\textsc{P-DerefSlicePtrSlice}\ \frac{\sigma \vdash \square \times \mathcal{R}_1 \Downarrow \_ \mapsto \texttt{ptr}\ \mathcal{R}_2[i..j] \qquad \sigma \vdash p^{\square} \times \mathcal{R}_2 \Downarrow \mathcal{R}_3 \mapsto [\![v_0, \ldots, v_i, \ldots, v_j, \ldots, v_n]\!]}{\sigma \vdash p^{\square}[*\square] \times \mathcal{R}_1 \Downarrow \mathcal{R}_3[i..j] \mapsto [\![v_i, \ldots, v_j]\!]}$$

$$\textsc{WF-RefSliceSlice}\ \frac{\Sigma; \Gamma \vdash \mathcal{R} : [\tau^{\text{SI}}] \qquad i \le j}{\Sigma; \Gamma \vdash \mathcal{R}[i..j] : [\tau^{\text{SI}}]}$$

Applying the induction hypothesis to $\sigma \vdash p^{\square} \times \mathcal{R}_2 \Downarrow \mathcal{R}_3 \mapsto [\![v_0, \ldots, v_i, \ldots, v_j, \ldots, v_n]\!]$ gives us $\Sigma; \Gamma \vdash \mathcal{R}_3 : [\tau^{\text{SI}}]$. Then, we can apply WF-RefSliceSlice to get $\Sigma; \Gamma \vdash \mathcal{R}_3[i..j] : \tau^{\text{SI}}$. □

Lemma E.6 (Reduced Place Expressions Have Roots in Loan Sets). *If* $\Sigma \vdash \sigma : \Gamma$, $\sigma \vdash p \Downarrow \mathcal{R}^{\square}[\pi] \mapsto v$, *and* $\bullet; \Gamma \vdash_{\omega} p \Rightarrow \{\, \overline{\ell} \,\}$, *then* $\mathcal{R} = \mathcal{R}^{\square}[\pi]$ *and* ${}^{\omega}\pi \in \{\, \overline{\ell} \,\}$.

Proof. We proceed by induction on $\bullet; \Gamma \vdash_{\omega} p \Rightarrow \{\, \overline{\ell} \,\}$. There are ordinarily three cases: O-SafePlace, O-Deref, and O-DerefAbs. However, O-DerefAbs requires the type variable context to contain entries, and thus can be immediately discharged by contradiction. This leaves us with only O-SafePlace and O-Deref.

$$\textsc{O-SafePlace}\ \frac{\begin{array}{c}\forall r' \mapsto \{\, \overline{\ell} \,\} \in \Gamma.\ (\forall\, {}^{\omega'} p^{\square}[\pi'] \in \{\, \overline{\ell} \,\}.(\omega = \texttt{uniq} \vee \omega' = \texttt{uniq}) \implies \pi' \# \pi) \\ \vee\ (\exists \pi' : \&r'\ \omega'\ \tau' \in \Gamma \wedge (\forall \pi' : \&r'\ \omega'\ \tau' \in \Gamma.\ \pi' \in \{\, \overline{\pi_e} \,\}))\end{array}}{\Delta; \Gamma \vdash_{\omega}^{\overline{\pi_e}} \pi \Rightarrow \{\, {}^{\omega}\pi \,\}}$$

O-SafePlace tells us that our p is in fact a place π meaning that it does not contain any dereferences. As such, we know that $\sigma \vdash p \Downarrow \mathcal{R}^{\square}[\pi] \mapsto v$ must have been derived using a combination of P-Referent and P-Proj corresponding to the structure of π. The resulting referent in such a case is precisely π (meaning $\mathcal{R}^{\square} = \square$), which we know is in the output immediately from the definition of O-SafePlace.

$$\textsc{O-Deref}\ \frac{\begin{array}{c}\Gamma(\pi) = \&r\ \omega_\pi\ \tau_\pi \qquad \Gamma(r) = \{\ \overline{\omega' p_i}\ \} \qquad \overline{p_i = p_i^{\square}[\pi_i]} \qquad \omega \lesssim \omega_\pi \\ \forall i \in \{1 \ldots n\}.\ \Delta;\ \Gamma \vdash^{\overline{\pi_e}, \overline{\pi_i}, \pi}_{\omega} p^{\square}[p_i] \Rightarrow \{\ \overline{{}^{\omega} p_i'}\ \} \\ \forall r' \mapsto \{\ \overline{\ell}\ \} \in \Gamma.\ (\forall\, {}^{\omega'} p \in \{\ \overline{\ell}\ \}.(\omega = \mathtt{uniq} \vee \omega' = \mathtt{uniq}) \implies p \# p^{\square}[*\pi]) \\ \vee\ (\exists \pi' : \&r'\ \omega'\ \tau' \in \Gamma \wedge (\forall \pi' : \&r'\ \omega'\ \tau' \in \Gamma.\ \pi' \in \{\ \overline{\pi_e}, \overline{\pi_i}, \pi\ \}))\end{array}}{\Delta;\ \Gamma \vdash^{\overline{\pi_e}}_{\omega} p^{\square}[*\pi] \Rightarrow \{\ \overline{{}^{\omega} p_1'}, \ \ldots\ \overline{{}^{\omega} p_n'}, \ {}^{\omega} p^{\square}[*\pi]\ \}}$$

In the premise of O-Deref, we have a number of ownership safety derivations corresponding to each of the loans for the pointer being dereferenced. Since we know we have a dereference, we know that we must have derived $\sigma \vdash p \Downarrow \mathcal{R}^{\square}[\pi] \mapsto v$ using one of the five dereference rules at the appropriate point (P-DerefPtr, P-DerefIndexPtrArray, P-DerefIndexPtrSlice, P-DerefSlicePtrArray, and P-DerefSlicePtrSlice). Each of which share a common premise (at least when sufficiently generalized): $\sigma \vdash p^{\square} \times \mathcal{R}_2 \Downarrow \mathcal{R}_3 \mapsto v$. Here, $\mathcal{R}_2$ corresponds to the referent of the pointer we are dereferencing. As such, we know that one of the derivations of ownership safety corresponds to that particular referent. So, we can apply our induction hypothesis and get that ${}^{\omega}\pi \in \{\ \overline{{}^{\omega} p_i'}\ \}$ for the appropriate ownership safety derivation numbered i. The final output is the union of all of these sets, and thus we can generalize to ${}^{\omega}\pi \in \{\ \overline{{}^{\omega} p_1'}, \ \ldots, \ \overline{{}^{\omega} p_n'}, \ \ldots, \ {}^{\omega} p^{\square} * \pi\ \}$. □

Lemma E.7 (Subtyping Preserves Value Typing). *If* $\Sigma;\ \Delta;\ \Gamma \vdash \boxed{v} : \tau \Rightarrow \Gamma$ *and* $\Delta;\ \Gamma \vdash \tau_2 \lesssim \tau_1 \Rightarrow \Gamma'$ *then* $\Sigma;\ \Delta;\ \Gamma' \vdash \boxed{v} : \tau \Rightarrow \Gamma'$.

Proof. We proceed by induction on the subtyping judgement. The only cases that don't follow immediately by induction and application of premises are S-UniqueRef and S-SharedRef, and in both cases the only interesting part of the proof is the outlives constraint.

Proceeding by induction on the outlives constraint, the only interesting case is OL-LocalProvenances.

$$\textsc{OL-LocalProvenances}\ \frac{\begin{array}{c}\forall \pi : \&r_1\ \omega\ \tau \in \Gamma.\ \nexists r'.\ {}^{\omega} * \pi \in \Gamma(r') \\ r_1\ \mathtt{occurs\ before}\ r_2\ \mathtt{in}\ \Gamma\end{array}}{\Delta;\ \Gamma \vdash r_1 :> r_2 \Rightarrow \Gamma[r_2 \mapsto \{\ \Gamma(r_1) \cup \Gamma(r_2)\ \}]}$$

So we want to show that $\Sigma;\ \Delta;\ \Gamma[r_2 \mapsto \Gamma(1) \cup \Gamma(2)] \vdash \boxed{v} : \tau \Rightarrow \Gamma[r_2 \mapsto \Gamma(1) \cup \Gamma(2)]$. We proceed by induction on the value typing.

$$\textsc{T-Pointer}\ \frac{\Sigma;\ \Gamma \vdash \mathcal{R}^{\square}[\pi] : \tau^{\mathrm{XI}} \qquad {}^{\omega}\pi \in \Gamma(r)}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\mathtt{ptr}\ \mathcal{R}^{\square}[\pi]} : \&r\ \omega\ \tau^{\mathrm{XI}} \Rightarrow \Gamma}$$

The T-Pointer case is immediate, because by inspection of the referent well formedness, there is no reliance on loan sets, and the loan is preserved since the loan sets only grow.

$$\textsc{T-ClosureValue}\ \frac{\text{free-vars}(e) \setminus \overline{x} = \overline{x_f} = \text{dom}(\mathcal{F}_c)|_x \qquad \overline{r} = \overline{\text{free-provs}(\Gamma(x_f))},\ \text{free-provs}(e) = \text{dom}(\mathcal{F}_c)|_r \qquad \Sigma;\ \Gamma \vdash \varsigma_c : \mathcal{F}_c \qquad \Sigma;\ \Delta;\ \Gamma \natural \mathcal{F}_c,\ x_1 : \tau_1^{\text{SI}},\ \ldots,\ x_n : \tau_n^{\text{SI}} \vdash \boxed{e} : \tau_r^{\text{SI}} \Rightarrow \Gamma' \natural \mathcal{F}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\langle \varsigma_c,\ |x_1 : \tau_1^{\text{SI}},\ \ldots,\ x_n : \tau_n^{\text{SI}}|\ \rightarrow\ \tau_r^{\text{SI}}\ \{\ e\ \}\ \rangle} : (\tau_1^{\text{SI}},\ \ldots,\ \tau_n^{\text{SI}}) \xrightarrow{\mathcal{F}_c} \tau_r^{\text{SI}} \Rightarrow \Gamma}$$

We proceed by induction on the body of the closure. The only interesting cases are those that use ownership safety, which are T-Move, T-Copy, T-Borrow, T-BorrowIndex, T-BorrowSlice, T-IndexCopy, and T-AssignDeref. The only interesting part is the ownership safety judgement itself, but since we're only unioning together two loan sets, the disjointness condition is mostly immediate, since those places would have been considered anyway. The only potential problem is that r_1 is excluded in a reborrow chain and not r_2, but then the only way for all of the pointers with r_1 to be added to the exclusion list is if all of them were reborrowed, but this is disallowed by the reborrow restriction on r_1 in OL-LocalProvenances. □

Lemma E.8 (Subtyping Preserves Well Formed Stacks). *If* $\Sigma \vdash \sigma : \Gamma$ *and* $\bullet;\ \Gamma \vdash \tau_2 \lesssim \tau_1 \Rightarrow \Gamma'$ *then* $\Sigma \vdash \sigma : \Gamma'$.

Proof. We proceed by induction on the stack typing derivation.

$$\textsc{WF-StackFrame}\ \frac{\Sigma \vdash \sigma : \Gamma \qquad \text{dom}(\varsigma) = \text{dom}(\mathcal{F})|_x \qquad \forall x \in \text{dom}(\varsigma).\ \Sigma;\ \bullet;\ \Gamma \natural \mathcal{F} \vdash \boxed{(\sigma \natural \varsigma)(x)} : (\Gamma \natural \mathcal{F})(x) \Rightarrow \Gamma \natural \mathcal{F}}{\Sigma \vdash \sigma \natural \varsigma : \Gamma \natural \mathcal{F}} \qquad \textsc{WF-StackEmpty}\ \frac{}{\Sigma \vdash \bullet : \bullet}$$

The WF-StackEmpty case is immediate. In the WF-StackFrame case, we get the well formedness in the premise from our induction hypothesis. What's left to show is that for all of the values v in the stack frame, they remain well typed in Γ'. This follows from applying Lemma E.7. □

Lemma E.9 (Values Change Environments in Limited Ways). *If* $\Sigma;\ \Delta;\ \Gamma \vdash \boxed{v} : \tau \Rightarrow \Gamma'$ *, then* $\Sigma;\ \Delta \vdash \Gamma \lesssim \Gamma'$.

Proof. We proceed by induction on the structure of the typing derivation. Since we assume that the expression being typed is a value, we need only consider the cases that can be used to type a value.

For many cases, the output environments are precisely the input environments, and thus this holds immediately. These cases are T-Unit, T-u32, T-True, T-False, T-Pointer, T-Function, T-ClosureValue, and T-Dead.

For T-Tuple and T-Array, knowing that we have a value means that all of the subterms are themselves values, and thus we can apply our induction hypothesis to them in sequence (relying on the transitivity of $\lesssim$ for stack typings).

This leaves us with one remaining case: T-Drop.

$$\textsc{T-Drop}\ \frac{\Gamma(\pi) = \tau_\pi^{\text{SI}} \qquad \Sigma;\ \Delta;\ \Gamma[\pi \mapsto \tau_\pi^{\text{SI}^\dagger}] \vdash \boxed{e} : \tau^{\text{SX}} \Rightarrow \Gamma_f}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e} : \tau^{\text{SX}} \Rightarrow \Gamma_f}$$

For T-Drop, we apply our induction hypothesis to $\Sigma;\ \Delta;\ \Gamma[\pi \mapsto \tau_\pi^{\mathrm{SI}^\dagger}] \vdash \boxed{e} : \tau^{\mathrm{SX}} \Rightarrow \Gamma_f$ which tells us that $\Sigma;\ \Delta \vdash \Gamma[\pi \mapsto \tau_\pi^{\mathrm{SI}^\dagger}] \lesssim \Gamma_f$. Then, by R-Env, we have that $\Sigma;\ \Delta \vdash \Gamma \lesssim \Gamma[\pi \mapsto \tau_\pi^{\mathrm{SI}^\dagger}]$. Then, by transitivity, we have $\Sigma;\ \Delta \vdash \Gamma \lesssim \Gamma_f$. □

Lemma E.10 (Preservation of Types under Substitution).

(1) *If* $\Sigma;\ \Delta,\ \alpha : \star;\ \Gamma \vdash \boxed{e} : \tau \Rightarrow \Gamma'$ *and* $\Sigma;\ \Delta;\ \Gamma \vdash \tau'$, *then* $\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e[^{\tau'}/_{\alpha}]} : \tau[^{\tau'}/_{\alpha}] \Rightarrow \Gamma'[^{\tau'}/_{\alpha}]$

(2) *If* $\Sigma;\ \Delta,\ \varrho : \mathtt{PRV};\ \Gamma \vdash \boxed{e} : \tau \Rightarrow \Gamma'$ *and* $\Delta;\ \Gamma \vdash \rho$, *then* $\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e[^{\rho}/_{\varrho}]} : \tau[^{\rho}/_{\varrho}] \Rightarrow \Gamma'[^{\rho}/_{\varrho}]$

(3) *If* $\Sigma;\ \Delta, \varphi{:}\mathtt{FRM};\ \Gamma \vdash \boxed{e} : \tau \Rightarrow \Gamma'$ *and* $\Sigma;\ \Delta;\ \Gamma \vdash \Phi$, *then* $\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e[^{\Phi}/_{\varphi}]} : \tau[^{\Phi}/_{\varphi}] \Rightarrow \Gamma'[^{\Phi}/_{\varphi}]$

Proof. By induction on the typing derivation. □

Lemma E.11 (Type Computation is Preserved in Related Environments). *If* $\Sigma;\ \Delta \vdash \Gamma \lesssim \Gamma'$ *and* $\Delta;\ \Gamma \vdash_\omega p^{\square}[\pi] : \tau,\ \{\,\overline{\rho}\,\}$ *and* $\Gamma(\pi) = \Gamma'(\pi)$, *then* $\Delta;\ \Gamma' \vdash_\omega p^{\square}[\pi] : \tau,\ \{\,\overline{\rho}\,\}$.

Proof. We proceed by induction on the type computation derivation. TC-Var follows immediately by the same type hypothesis, and TC-Proj follows from applying the induction hypothesis. All that is left is TC-Deref.

$$\textsc{TC-Deref}\ \frac{\Delta;\ \Gamma \vdash_\omega p : \&\rho\ \omega'\ \tau^{\mathrm{XI}},\ \{\,\overline{\rho_p}\,\} \qquad \omega \lesssim \omega' \qquad \Delta;\ \Gamma \vdash \overline{\rho :> \rho_p} \Rightarrow \Gamma_f}{\Delta;\ \Gamma \vdash_\omega *p : \tau^{\mathrm{XI}},\ \{\,\overline{\rho_p},\ \rho\,\}}$$

First, we can apply the induction hypothesis to get the type computation for p. Then, all that's left is to show the outlives constraint, but this is immediate because Δ is unchanged and both Γ and Γ' have the exact same domains.

□

Lemma E.12 (Ownership Safety Preserved in Related Environments). *If* $\Delta;\ \Gamma \vdash_\omega^{\overline{\pi_e}} p \Rightarrow \{\,\overline{\ell}\,\}$ *and* $\Sigma; \Delta \vdash \Gamma \lesssim \Gamma'$ *and* $\Delta;\ \Gamma' \vdash_\omega p : \tau^{\mathrm{XI}}$ *and* $p = p^{\square}[\pi_p]$ *and* $\Gamma(\pi_p) = \Gamma'(\pi_p)$, *then* $\Delta; \Gamma' \vdash_\omega^{\overline{\pi_e}} p \Rightarrow \{\overline{\ell}\}$.

Proof. We proceed by induction on the ω-safety derivation, for which there are three cases to consider.

$$\textsc{O-SafePlace}\ \frac{\begin{array}{l}\forall r' \mapsto \{\,\overline{\ell}\,\} \in \Gamma.\ (\forall\, {}^{\omega'} p^{\square}[\pi'] \in \{\,\overline{\ell}\,\}.(\omega = \mathtt{uniq} \vee \omega' = \mathtt{uniq}) \implies \pi' \# \pi) \\ \qquad \vee\ (\exists \pi' : \&r'\ \omega'\ \tau' \in \Gamma \ \wedge\ (\forall \pi' : \&r'\ \omega'\ \tau' \in \Gamma.\ \pi' \in \{\,\overline{\pi_e}\,\}))\end{array}}{\Delta;\ \Gamma \vdash_\omega^{\overline{\pi_e}} \pi \Rightarrow \{\ {}^{\omega}\pi\ \}}$$

We'd like to show that O-SafePlace can be applied with context Γ'. First, note that for any r', if the right side of the or is true for Γ with $\overline{\pi}$ then it will be true for Γ' with $\overline{\pi}$. That is, if all of the pointers with provenance r' in Γ are in the exclusion list $\overline{\pi}$, then all of the pointers with provenance r' in Γ' are also in the exclusion list $\overline{\pi}$. Therefore, the only cases we need to consider are where r' occurs in pointers in Γ and Γ' that do not occur in $\overline{\pi}$.

Since the only allowed change to loan sets is emptying, and an emptied loan set has the left side of the disjunction as vacuously true, and if the loan set is the same we have the condition from the ownership safety in the premise, we are done.

$$\textsc{O-Deref}\quad \frac{\Gamma(\pi) = \&r\ \omega_\pi\ \tau_\pi \qquad \Gamma(r) = \{\ \overline{\omega' p_i}\ \} \qquad \overline{p_i = p_i^\square[\pi_i]} \qquad \omega \lesssim \omega_\pi \\ \forall i \in \{1 \ldots n\}.\ \Delta;\ \Gamma \vdash^{\overline{\pi_e}, \overline{\pi_i}, \pi}_{\omega} p^\square[p_i] \Rightarrow \{\ \overline{{}^{\omega} p_i'}\ \} \\ \forall r' \mapsto \{\ \overline{\ell}\ \} \in \Gamma.\ (\forall\, {}^{\omega'} p \in \{\ \overline{\ell}\ \}.(\omega = \mathtt{uniq} \vee \omega' = \mathtt{uniq}) \implies p \# p^\square[*\pi]) \\ \vee\ (\exists \pi' : \&r' \omega' \tau' \in \Gamma \wedge (\forall \pi' : \&r' \omega' \tau' \in \Gamma.\ \pi' \in \{\ \overline{\pi_e}, \overline{\pi_i}, \pi\ \}))}{\Delta;\ \Gamma \vdash^{\overline{\pi_e}}_{\omega} p^\square[*\pi] \Rightarrow \{\ \overline{{}^{\omega} p_1'}, \ \ldots\ \overline{{}^{\omega} p_n'}, \ {}^{\omega} p^\square[*\pi]\ \}}$$

Firstly, we have that $\Gamma(\pi_i) = \Gamma'(\pi_i)$, because $\Gamma'(\pi_i)$ must be an initialized type by the type computation premise, and the only changes in types between Γ and Γ' allowed by the environment relation is dropping some types to uninitialized.

Second, note that $\Gamma'(r) = \Gamma(r)$ since $\Gamma'(\pi)$ being a reference with provenance r means we can't empty the loan set. So we proceed by applying the induction hypothesis for all n loans, noting that the type computation requirement follows from the well formedness of Γ'.

Finally, we have to show the statement about no conflicting loans, but here the argument is identical to that in the O-SafePlace case. If the loan set is empty then we're done, otherwise we just use the ownership safety premise.

$$\textsc{O-DerefAbs}\quad \frac{\Gamma(\pi) = \&\varrho\ \omega_\pi\ \tau_\pi \qquad \Delta;\ \Gamma \vdash_\omega p^\square[*\pi] : \tau \qquad \omega \lesssim \omega_\pi \\ \forall r' \mapsto \{\ \overline{\ell}\ \} \in \Gamma.\ (\forall\, {}^{\omega'} p \in \{\ \overline{\ell}\ \}.(\omega = \mathtt{uniq} \vee \omega' = \mathtt{uniq}) \implies p \# p^\square[*\pi]) \\ \vee\ (\exists \pi' : \&r' \omega' \tau' \in \Gamma \wedge (\forall \pi' : \&r' \omega' \tau' \in \Gamma.\ \pi' \in \{\ \overline{\pi_e}, \pi\ \}))}{\Delta;\ \Gamma \vdash^{\overline{\pi_e}}_{\omega} p^\square[*\pi] \Rightarrow \{\ {}^{\omega} p^\square[*\pi]\ \}}$$

This case proceeds similarly to the O-Deref case, but with an added application of Lemma E.11 to get the type computation, and no application of any induction hypothesis. □

Lemma E.13 (Types Are Well Formed in Related Environments). *If* $\Sigma;\ \Delta \vdash \Gamma \lesssim \Gamma'$ *and* $\Sigma;\ \Delta;\ \Gamma \vdash \tau^{XI}$ *and* $\forall r$ *that occur in* τ^{XI}, $\Gamma(r) = \Gamma'(r)$, *then* $\Sigma;\ \Delta;\ \Gamma' \vdash \tau^{XI}$.

Proof. We proceed by induction on the type well formedness derivation. The only case that doesn't follow directly from induction and the fact that Δ is unchanged between the two related environments is WF-Ref.

$$\textsc{WF-Ref}\quad \frac{(\Gamma(r) = \emptyset \vee \exists\, {}^{\omega} p \in \Gamma(r).\ \exists \tau_p^{\mathrm{XI}}.\ \Delta;\ \Gamma \vdash_\omega p : \tau_p^{\mathrm{XI}}.\ \tau^{\mathrm{XI}}\ \mathtt{occurs\ in}\ \tau_p^{\mathrm{XI}}) \\ \Sigma;\ \Delta;\ \Gamma \vdash \tau^{\mathrm{XI}}}{\Sigma;\ \Delta;\ \Gamma \vdash \&r\ \omega\ \tau^{\mathrm{XI}}}$$

Firstly we apply our induction hypothesis to get that $\Sigma;\ \Delta;\ \Gamma' \vdash \tau_p^{\mathrm{XI}}$. What's left to show is the loan set condition on r. If $\Gamma'(r) = \emptyset$, then we're done. Otherwise, we just need that the type computation still holds, which we get from Lemma E.11. We know the places in these place expressions all have the same type in Γ and Γ' because between these two contexts the only changes allowed that could cause problems here are dropping one of these places, but then Γ' would not be well formed since there would be an invalid loan.

□

Lemma E.14 (Related Environments remain well formed). *If* $\Sigma;\ \Delta \vdash \Gamma \lesssim \Gamma'$ *and* $\vdash \Sigma;\ \Delta;\ \Gamma \natural \mathcal{F}_c$ *then* $\vdash \Sigma;\ \Delta;\ \Gamma' \natural \mathcal{F}_c$.

Proof. From the well formedness of $\Gamma \natural \mathcal{F}_c$, we know that the places and disjointness conditions both hold. By Lemma E.13, noting that $\Sigma;\ \Delta \vdash \Gamma \natural \mathcal{F}_c \lesssim \Gamma' \natural \mathcal{F}_c$ is immediate, we know that the types remain well formed in the environment. We also have the well formedness of Γ' as a premise of the related environments judgement. All that's left to show is the loan set condition. But for this all we have to show is that each place computes to some type, which follows from Lemma E.11. We know the types of the places in each place expression remain the same because the only allowed changes between Γ and Γ' are that places can be dropped and loan sets emptied, but if one such place was dropped, then Γ' would have not been well formed. □

Lemma E.15 (Related Input Environments Produce Similar Output Environments). *If:*

- $\Sigma;\ \Delta;\ \Gamma_1 \vdash \boxed{e_1} : \tau_1 \Rightarrow \Gamma_2$
- $\Sigma;\ \Delta;\ \Gamma_1 \vdash \boxed{e_2} : \tau_2 \Rightarrow \Gamma_3$
- $\Sigma;\ \Delta \vdash \Gamma_1 \lesssim \Gamma_1'$
- $\Sigma;\ \Delta;\ \Gamma_1' \vdash \boxed{e_1} : \tau_1 \Rightarrow \Gamma_2'$
- $\Sigma;\ \Delta;\ \Gamma_1' \vdash \boxed{e_2} : \tau_2 \Rightarrow \Gamma_3'$
- $\Sigma;\ \Delta \vdash \Gamma_2 \lesssim \Gamma_2'$
- $\Sigma;\ \Delta \vdash \Gamma_3 \lesssim \Gamma_3'$
- $\forall x \in dom(\Gamma_2), \Gamma_2(x) = \Gamma_3(x)$ *and* $\Gamma_2'(x) = \Gamma_3'(x)$
- $\forall r$ *that occur in* e_1 *or* e_2 *or* τ_1 *or* τ_2, $\Gamma_1(r) = \Gamma_1'(r)$

then $\forall r \in dom(\Gamma_1)$, *if* $\Gamma_2'(r) = \emptyset$ *and* $\Gamma_3'(r) \neq \emptyset$, *then* $\Gamma_2(r) = \emptyset$, *and if* $\Gamma_3'(r) = \emptyset$ *and* $\Gamma_2'(r) \neq \emptyset$, *then* $\Gamma_3(r) = \emptyset$.

Proof. The proofs for both statements in the conclusion follow identically, so without loss of generality it suffices to show that if $\Gamma_2'(r) = \emptyset$ and $\Gamma_3'(r) \neq \emptyset$, then $\Gamma_2(r) = \emptyset$. Note there are two cases to consider: that the loan set was empty all along, or that the loan set was at some point non empty, but then got garbage collected.

First, at some point between Γ_1' and Γ_2', r mapped to a non empty set of loans but then was garbage collected. In this case, Γ_2' must not contain any references that contain r, since otherwise it would have been invalid to garbage collect r. But then since Γ_2' and Γ_3' agree on types, it must be the case that it was also garbage collected in Γ_3', which is a contradiction with the fact that $\Gamma_3'(r)$ is non empty, so this case is impossible.

Second, at each step of the derivation between Γ_1' and Γ_2', r mapped to empty. If $\Gamma_1(r)$ also was empty, then this means $\Gamma_2(r)$ is also empty, and we're done. Otherwise, r was garbage collected between Γ_1 and Γ_1'. But then r must be free in e_2 for loans to have been added between Γ_1' and Γ_3', which means the loan set could not have been emptied between Γ_1 and Γ_1', which is a contradiction. □

Lemma E.16 (Outlives Preserves Related Environments). *If* $\Delta;\ \Gamma \vdash \rho_1 :> \rho_2 \Rightarrow \Gamma_o$, *and* $\Sigma;\ \Delta \vdash \Gamma \lesssim \Gamma'$ *and* $\vdash \Sigma;\ \Delta;\ \Gamma_o$ *and* $\Gamma(\rho_1) = \Gamma'(\rho_1)$ *and* $\Gamma(\rho_2) = \Gamma'(\rho_2)$, *then* $\Delta;\ \Gamma' \vdash \rho_1 :> \rho_2 \Rightarrow \Gamma_o'$, *and* $\Sigma;\ \Delta \vdash \Gamma_o \lesssim \Gamma_o'$. *and* $\Gamma_o(\rho_1) = \Gamma_o'(\rho_1)$ *and* $\Gamma_o(\rho_2) = \Gamma_o'(\rho_2)$

Proof. Proceed by induction on the outlives derivation. OL-ReflProv, OL-TransProv, OL-AbsProvLocalProv, and OL-AbstractProvenances are immediate.

OL-LocalProvAbsProv follows from additionally applying Lemma E.11. The condition on the place having the same type follows from the fact that p is a loan and $\Gamma'(r)$ is not emptied, so we could not have dropped the place.

This leaves one case: OL-LocalProvenances

$$\textsc{OL-LocalProvenances}\ \frac{\forall \pi : \&r_1\ \omega\ \tau \in \Gamma.\ \nexists r'.\ {}^{\omega} * \pi\ \in \Gamma(r') \qquad r_1\ \texttt{occurs before}\ r_2\ \texttt{in}\ \Gamma}{\Delta;\ \Gamma \vdash r_1 :> r_2 \Rightarrow \Gamma[r_2 \mapsto \{\ \Gamma(r_1) \cup \Gamma(r_2)\ \}]}$$

Since $\Gamma'(r_1) = \Gamma(r_1)$ and $\Gamma'(r_2) = \Gamma(r_2)$, $\Gamma'(r_1) \cup \Gamma'(r_2) = \Gamma(r_1) \cup \Gamma'(r_2)$. The rest of the conditions are immediate: the equality on r_1 and r_2's loan sets, and well formedness. □

Lemma E.17 (Subtyping Preserves Related Environments). *If* $\Delta;\ \Gamma \vdash \tau_1^{SI} \lesssim \tau_2^{SI} \Rightarrow \Gamma_o$, *and* $\Sigma; \Delta \vdash \Gamma \lesssim \Gamma'$ *and* $\vdash \Sigma$; Δ; Γ_o *and* $\forall r$ *that occur in* τ_1^{SI} *or* τ_2^{SI}, $\Gamma(r) = \Gamma'(r)$, *then* $\Delta;\ \Gamma' \vdash \tau_1^{SI} \lesssim \tau_2^{SI} \Rightarrow \Gamma_o'$, *and* $\Sigma;\ \Delta \vdash \Gamma_o \lesssim \Gamma_o'$, *and* $\forall r$ *that occur in* τ_1^{SI} *or* τ_2^{SI}, $\Gamma_o(r) = \Gamma_o'(r)$.

Proof. Proceed by induction on the Subtyping derivation. The only interesting cases are S-SharedRef and S-UniqueRef, both of which proceed by Lemma E.16 in addition to applying the induction hypothesis. □

Lemma E.18 (Expression Typing Preserved in Related Environments). *Let* e *be a surface expression as defined on page 1. If* $\Sigma;\ \Delta;\ \Gamma \natural \mathcal{F} \vdash \boxed{e} : \tau \Rightarrow \Gamma_o \natural \mathcal{F}_o$ *and* $\Sigma;\ \Delta \vdash \Gamma \natural \mathcal{F} \lesssim \Gamma' \natural \mathcal{F}$ *and* $\overline{\textit{free-vars}(e) = x_f} \subseteq \textit{dom}(\mathcal{F})|_x$ *and* $\forall r \in \textit{free-provs}(e).\ r \in \textit{dom}(\mathcal{F})$, *and* $\forall r$ *that occur a type in* $\overline{\mathcal{F}(x_f)}$, $\Gamma(r) = \Gamma'(r)$ *then* $\Sigma;\ \Delta;\ \Gamma' \natural \mathcal{F} \vdash \boxed{e} : \tau \Rightarrow \Gamma_o' \natural \mathcal{F}_o$ *and* $\Sigma;\ \Delta \vdash \Gamma_o \natural \mathcal{F}_o \lesssim \Gamma_o' \natural \mathcal{F}_o$ *and* $\forall r$ *that occur a type in* $\overline{\mathcal{F}(x_f)}$, $\Gamma_o(r) = \Gamma_o'(r)$.

Proof. Proceed by induction on the typing derivation for e. In the cases of T-Abort, T-Function, T-Unit, T-u32, T-True, and T-False, the results are immediate.

In the cases of T-LetProv, T-While, T-ForArray, T-ForSlice, T-Closure, T-Tuple, and T-Array, they all follow immediately from induction hypotheses.

For each of the following cases, the convention is that the statement in the box is our assumption, and we want to prove the same statement with Γ' replaced for each Γ.

$$\textsc{T-Branch}\ \frac{\begin{array}{c}\Sigma;\ \Delta;\ \Gamma \natural \mathcal{F} \vdash \boxed{e_1} : \texttt{bool} \Rightarrow \Gamma_1 \natural \mathcal{F}_1 \qquad \Sigma;\ \Delta;\ \Gamma_1 \natural \mathcal{F}_1 \vdash \boxed{e_2} : \tau_2^{\text{SI}} \Rightarrow \Gamma_2 \natural \mathcal{F}_2 \\ \Sigma;\ \Delta;\ \Gamma_1 \natural \mathcal{F}_1 \vdash \boxed{e_3} : \tau_3^{\text{SI}} \Rightarrow \Gamma_3 \natural \mathcal{F}_3 \qquad \tau^{\text{SI}} = \tau_2^{\text{SI}} \vee \tau^{\text{SI}} = \tau_3^{\text{SI}} \\ \Delta;\ \Gamma_2 \natural \mathcal{F}_2 \vdash \tau_2^{\text{SI}} \lesssim \tau^{\text{SI}} \Rightarrow \Gamma_{2s} \natural \mathcal{F}_{2s} \qquad \Delta;\ \Gamma_3 \natural \mathcal{F}_3 \vdash \tau_3^{\text{SI}} \lesssim \tau^{\text{SI}} \Rightarrow \Gamma_{3s} \natural \mathcal{F}_{3s} \qquad \Gamma_{2s} \natural \mathcal{F}_{2s} \Cup \Gamma_{3s} \natural \mathcal{F}_{3s} = \Gamma_o \natural \mathcal{F}_o\end{array}}{\Sigma;\ \Delta;\ \Gamma \natural \mathcal{F} \vdash \boxed{\texttt{if}\ e_1\ \{\ e_2\ \}\ \texttt{else}\ \{\ e_3\ \}} : \tau^{\text{SI}} \Rightarrow \Gamma_o \natural \mathcal{F}_o}$$

By our induction hypothesis we get that $\Sigma;\ \Delta;\ \Gamma' \natural \mathcal{F} \vdash \boxed{e_1} : \texttt{bool} \Rightarrow \Gamma_1' \natural \mathcal{F}_1$, and $\Sigma;\ \Delta;\ \Gamma_1' \natural \mathcal{F}_1 \vdash \boxed{e_2} : \texttt{bool} \Rightarrow \Gamma_2' \natural \mathcal{F}_2$, and $\Sigma;\ \Delta;\ \Gamma_1' \natural \mathcal{F}_1 \vdash \boxed{e_3} : \texttt{bool} \Rightarrow \Gamma_3' \natural \mathcal{F}_3$ with $\Sigma;\ \Delta \vdash \Gamma_2 \natural \mathcal{F}_2 \lesssim \Gamma_2' \natural \mathcal{F}_2$ and $\Sigma;\ \Delta \vdash \Gamma_3 \natural \mathcal{F}_3 \lesssim \Gamma_3' \natural \mathcal{F}_3$.

Next we want to show that $\Delta;\ \Gamma_2' \natural \mathcal{F}_2 \vdash + \lesssim \tau_2^{\text{SI}} \Rightarrow \tau^{\text{SI}} \Gamma_{2s}' \natural \mathcal{F}_{2s}$, $\Sigma;\ \Delta \vdash \Gamma_{2s} \natural \mathcal{F}_{2s} \lesssim \Gamma_{2s}' \natural \mathcal{F}_{2s}$, $\Delta;\ \Gamma_3' \natural \mathcal{F}_3 \vdash + \lesssim \tau_3^{\text{SI}} \Rightarrow \tau^{\text{SI}} \Gamma_{3s}' \natural \mathcal{F}_{3s}$, and $\Sigma;\ \Delta \vdash \Gamma_{3s} \natural \mathcal{F}_{3s} \lesssim \Gamma_{3s}' \natural \mathcal{F}_{3s}$, which all follow from applying Lemma E.17. To do this lemma application, we just need to show that for all r in τ_1^{SI}, τ_2^{SI} and τ^{SI}, $\Gamma(r) = \Gamma'(r)$ which follows from the premise.

Finally, we need to show that $\Sigma;\ \Delta \vdash \Gamma_o \natural \mathcal{F}_o \lesssim \Gamma_o' \natural \mathcal{F}_o$. The well formedness condition on Γ_o' follows immediately since all types are the same as in Γ_2' and Γ_3' and the loan sets are just unioned, meaning reference types remain valid and we can compute types for all loans.

The equal or empty condition follows from the fact that Γ_2' and Γ_3' both agree on types by Lemma E.15, which means they drop exactly the same entries. For any provenances emptied, either the same provenances are emptied, or the provenance was emptied in the corresponding smaller context Γ_2 or Γ_3. Otherwise the loan sets are untouched.

Finally, both of these are preserved when adding on the same frame, so we're done.

$$\textsc{T-Let}\quad \frac{\Sigma;\ \Delta;\ \Gamma \natural \mathcal{F} \vdash \boxed{e_1} : \tau_1^{\text{SI}} \Rightarrow \Gamma_1 \natural \mathcal{F}_1 \qquad \Delta;\ \Gamma_1 \natural \mathcal{F}_1 \vdash \tau_1^{\text{SI}} \lesssim \tau_a^{\text{SI}} \Rightarrow \Gamma_{1s} \natural \mathcal{F}_{1s} \qquad \Sigma;\ \Delta;\ \text{gc-loans}(\Gamma_{1s} \natural \mathcal{F}_{1s},\ x : \tau_a^{\text{SI}}) \vdash \boxed{e_2} : \tau_2^{\text{SI}} \Rightarrow \Gamma_2 \natural \mathcal{F}_2,\ x : \tau^{\text{SD}}}{\Sigma;\ \Delta;\ \Gamma \natural \mathcal{F} \vdash \boxed{\texttt{let}\ x : \tau_a^{\text{SI}} = e_1;\ e_2} : \tau_2^{\text{SI}} \Rightarrow \Gamma_2 \natural \mathcal{F}_2}$$

Firstly, we apply our induction hypothesis to get that e_1 is well typed with input environment $\Gamma' \natural \mathcal{F}$ and output environment $\Gamma_1' \natural \mathcal{F}_1$ with $\Sigma;\ \Delta \vdash \Gamma_1 \natural \mathcal{F}_1 \lesssim \Gamma_1' \natural \mathcal{F}_1$. Then, we apply Lemma E.17 to get $\Sigma;\ \Delta \vdash \Gamma_{1s} \natural \mathcal{F}_{1s} \lesssim \Gamma_{1s}' \natural \mathcal{F}_{1s}$. In order to apply this lemma we need to know that for any r that occur in τ_1^{SI} or τ_a^{SI}, $\Gamma_1 \natural \mathcal{F}_1(r) = \Gamma_1' \natural \mathcal{F}_1(r)$, which we have as a conclusion from the previous application of the induction hypothesis.

To apply our induction hypothesis on e_2 and finish the case, we need that $\Sigma;\ \Delta \vdash \text{gc-loans}(\Gamma_{1s} \natural \mathcal{F}_{1s}, x : \tau_a^{\text{SI}}) \lesssim \text{gc-loans}(\Gamma_{1s}' \natural \mathcal{F}_{1s}, x : \tau_a^{\text{SI}})$. But this is immediate by definition since gcloans can only empty loan sets for provenances for which there are no types that contain them, which is allowed by S-Env.

Our final obligation to apply the induction hypothesis is that for any r that occurs in a type in $\mathcal{F}_{1s}$ but is not in $\mathcal{F}_{1s}$, we need that $\text{gc-loans}(\Gamma_{1s} \natural \mathcal{F}_{1s})(r) = \text{gc-loans}(\Gamma_{1s}' \natural \mathcal{F}_{1s})(r)$. We already have that $\Gamma_{1s} \natural \mathcal{F}_{1s}(r) = \Gamma_{1s}' \natural \mathcal{F}_{1s}(r)$, so we just need to know that $\exists \pi : \tau \in \Gamma_{1s}$, where r occurs in τ, and $\Gamma_{1s} \natural \mathcal{F}_{1s}(\pi) = \Gamma_{1s} \natural \mathcal{F}_{1s}'(\pi)$. But we said that r is contained in a type in $\mathcal{F}_{1s}$, so the place for that type is one such place, so we cannot empty the loan set.

$$\textsc{T-Seq}\quad \frac{\Sigma;\ \Delta;\ \Gamma \natural \mathcal{F} \vdash \boxed{e_1} : \tau_1^{\text{SI}} \Rightarrow \Gamma \natural \mathcal{F}_1 \qquad \Sigma;\ \Delta;\ \text{gc-loans}(\Gamma_1 \natural \mathcal{F}_1) \vdash \boxed{e_2} : \tau_2^{\text{SI}} \Rightarrow \Gamma_2 \natural \mathcal{F}_2}{\Sigma;\ \Delta;\ \Gamma \natural \mathcal{F} \vdash \boxed{e_1;\ e_2} : \tau_2^{\text{SI}} \Rightarrow \Gamma_2 \natural \mathcal{F}_2}$$

Firstly, we apply our induction hypothesis to get that e_1 is well typed with input environment $\Gamma' \natural \mathcal{F}$ and output environment $\Gamma_1' \natural \mathcal{F}_1$, with $\Sigma;\ \Delta \vdash \Gamma_1 \natural \mathcal{F}_1 \lesssim \Gamma_1' \natural \mathcal{F}_1$. We need to know that $\Sigma;\ \Delta \vdash \text{gc-loans}(\Gamma_1 \natural \mathcal{F}_1) \lesssim \text{gc-loans}(\Gamma_1' \natural \mathcal{F}_1)$ before we can apply our induction hypothesis to finish the proof. But this fact is trivial by the definitions, since gc-loans can only empty provenances that are not in initialized types in the context, which is allowed in S-Env.

Our final obligation to apply the induction hypothesis is that for any r that occurs in a type in $\mathcal{F}_1$ but is not in $\mathcal{F}_1$, we need that $\text{gc-loans}(\Gamma_1 \natural \mathcal{F}_1)(r) = \text{gc-loans}(\Gamma_1' \natural \mathcal{F}_1)(r)$. We already have that $\Gamma_1 \natural \mathcal{F}_1(r) = \Gamma_1' \natural \mathcal{F}_1(r)$, so we just need to know that $\exists \pi : \tau \in \Gamma_1$, where r occurs in τ, and $\Gamma_1 \natural \mathcal{F}_1(\pi) = \Gamma_1' \natural \mathcal{F}_1(\pi)$. But since *oxcprov* occurs in a type in $\mathcal{F}_1$, the place that maps to that type is such a place.

$$\textsc{T-Drop}\quad \frac{\Gamma(\pi) = \tau_\pi^{\text{SI}} \qquad \Sigma;\ \Delta;\ (\Gamma \natural \mathcal{F})[\pi \mapsto \tau_\pi^{\text{SI}^\dagger}] \vdash \boxed{e} : \tau^{\text{SX}} \Rightarrow \Gamma_o \natural \mathcal{F}_o}{\Sigma;\ \Delta;\ \Gamma \natural \mathcal{F} \vdash \boxed{e} : \tau^{\text{SX}} \Rightarrow \Gamma_o \natural \mathcal{F}_o}$$

In order to apply our induction hypothesis and finish the case, we only need to show that $\Sigma;\ \Delta \vdash (\Gamma \natural \mathcal{F})[\pi \mapsto \tau_\pi^{\text{SI}^\dagger}] \lesssim (\Gamma' \natural \mathcal{F})[\pi \mapsto \tau_\pi^{\text{SI}^\dagger}]$, which is immediate by the definition of related contexts. Note that $(\Gamma' \natural \mathcal{F})[\pi \mapsto \tau_\pi^{\text{SI}^\dagger}]$ is well formed because $(\Gamma \natural \mathcal{F})[\pi \mapsto \tau_\pi^{\text{SI}^\dagger}]$ is well formed.

There cannot be any loans to π because in the $(\Gamma' \natural \mathcal{F})[\pi \mapsto \tau_\pi^{\mathrm{SI}^\dagger}]$ because those loans would be there in $(\Gamma \natural \mathcal{F})[\pi \mapsto \tau_\pi^{\mathrm{SI}^\dagger}]$.

T-App

$$\frac{\begin{array}{c}\overline{\Sigma;\ \Delta;\ \Gamma \natural \mathcal{F} \vdash \Phi} \qquad \overline{\Delta;\ \Gamma \natural \mathcal{F} \vdash \rho} \qquad \overline{\Sigma;\ \Delta;\ \Gamma \natural \mathcal{F} \vdash \tau^{\mathrm{SI}}} \\ \Sigma;\ \Delta;\ \Gamma \natural \mathcal{F} \vdash \boxed{\hat{e}_f} : \forall <\overline{\varphi},\ \overline{\varrho},\ \overline{\alpha}>(\tau_1^{\mathrm{SI}},\ \ldots,\ \tau_n^{\mathrm{SI}}) \xrightarrow{\Phi_c} \tau_f^{\mathrm{SI}}\ \mathtt{where}\ \overline{\varrho_1 : \varrho_2} \Rightarrow \Gamma_0 \natural \mathcal{F}_0 \\ \forall i \in \{1 \ldots n\}.\ \Sigma;\ \Delta;\ \Gamma_{i-1} \natural \mathcal{F}_{i-1} \vdash \boxed{\hat{e}_i} : \tau_i^{\mathrm{SI}}\,\overline{[\Phi/\varphi]}\,\overline{[\rho/\varrho]}\,\overline{[\tau^{\mathrm{SI}}/\alpha]} \Rightarrow \Gamma_i \natural \mathcal{F}_i \\ \Delta;\ \Gamma_n \natural \mathcal{F}_n \vdash \overline{\varrho_2\,[\rho/\varrho]} :> \overline{\varrho_1\,[\rho/\varrho]} \Rightarrow \Gamma_b \natural \mathcal{F}_b\end{array}}{\Sigma;\ \Delta;\ \Gamma \natural \mathcal{F} \vdash \boxed{\hat{e}_f ::<\overline{\Phi},\ \overline{\rho},\ \overline{\tau^{\mathrm{SI}}}>(\hat{e}_1,\ \ldots,\ \hat{e}_n)} : \tau_f^{\mathrm{SI}}\,\overline{[\Phi/\varphi]}\,\overline{[\rho/\varrho]}\,\overline{[\tau^{\mathrm{SI}}/\alpha]} \Rightarrow \Gamma_b \natural \mathcal{F}_b}$$

In the case of T-App, we firstly must prove the well formedness properties:

- $\Sigma;\ \Delta;\ \Gamma' \natural \mathcal{F} \vdash \Phi$. Since Δ is unchanged, WF-Env is the only interesting case.

WF-Env

$$\frac{\Sigma;\ \Delta \vdash \Gamma \natural \mathcal{F} \natural \mathcal{F}_c}{\Sigma;\ \Delta;\ \Gamma \natural \mathcal{F} \vdash \mathcal{F}_c}$$

  Let $\Phi = \mathcal{F}_e$. We want to show that $\vdash \Sigma;\ \Delta;\ \Gamma' \natural \mathcal{F} \natural \mathcal{F}_c$ given $\vdash \Sigma;\ \Delta;\ \Gamma \natural \mathcal{F} \natural \mathcal{F}_c$, which is immediate from Lemma E.14.
- $\Delta;\ \Gamma' \natural \mathcal{F} \vdash \rho$, which is immediate from the premises since related loan environments have the same domains and Δ is the same.
- $\Sigma;\ \Delta;\ \Gamma' \natural \mathcal{F} \vdash \tau^{\mathrm{SI}}$, which is immediate from Lemma E.13. We just need that for the provenances that occur in the type, their loan sets are unchanged, but we get that from the premise, because the function argument is either: locally defined, in which case it can only use and produce types accessible in the context; an argument, in which case its arguments are also part of the argument type; or a global function, in which case these types do not contain any non abstract provenances which are replaced with concrete provenances all in $\mathcal{F}$.

For the rest of the application case, we can apply our induction hypothesis on the function and the arguments, additionally applying the substitution lemma, Lemma E.10, where needed. The last part about outlives follows from Lemma E.16, where we have the condition on the loan sets from the conclusion of the application of the induction hypothesis.

In the cases of T-Move, T-Copy, T-Borrow, T-BorrowIndex, T-BorrowSlice, IndexCopy, they all follow from the induction hypothesis and additionally applying Lemma E.12 and Lemma E.11. Note we get the place having the same type requirement from the fact that the place must be in $\mathcal{F}$ since it is a free variable.

The remaining cases of T-Assign and T-AssignDeref proceed similarly. Firstly, we apply the induction hypothesis on the expression, then Lemma E.12 and Lemma E.11, and finally we get well formedness and relatedness on the output environment by applying Lemma E.17. Note we get the place having the same type requirement for type computation from the fact that the place must be in $\mathcal{F}$ since it is a free variable.

□

Lemma E.19 (Referent Well Formedness Preserved in Related Environments). *If* $\Sigma;\ \Delta \vdash \Gamma \lesssim \Gamma'$ *and* $\Sigma;\ \Gamma \vdash \mathcal{R}^{\square}[\pi] : \tau^{XI}$ *and* $\Gamma(\pi) = \Gamma'(\pi)$, *then* $\Sigma;\ \Gamma' \vdash \mathcal{R}^{\square}[\pi] : \tau^{XI}$.

Proof. Proceed by induction on the referent validity derivation. The only case that doesn't follow immediately from premises and the induction hypothesis in WF-RefId, which follows from the equal types premise. □

Lemma E.20 (Value Typing Preserved in Related Environments). *If* $\Sigma;\ \bullet \vdash \Gamma \lesssim \Gamma'$, *then:*

(1) *If* $\Sigma;\ \bullet;\ \Gamma \vdash \boxed{v} : \Gamma(x) \Rightarrow \Gamma$, *then* $\Sigma;\ \bullet;\ \Gamma' \vdash \boxed{v} : \Gamma'(x) \Rightarrow \Gamma'$.

(2) *If* $\Sigma;\ \Gamma \vdash \varsigma : \mathcal{F}_c$, *then* $\Sigma;\ \Gamma' \vdash \varsigma : \mathcal{F}_c$.

Proof. Proceed by simultaneous induction on the typing derivation and the stack frame well formedness.

(1) Since we know the expression is already a value, we restrict ourselves only to those cases that type values: T-Unit, T-u32, T-True, T-False, T-Tuple, T-Array, T-Dead, T-Pointer, and T-ClosureValue.
For T-Unit, T-u32, T-Dead, T-True, and T-False, this holds trivially. For T-Tuple, and T-Array, this holds directly by repeated application of our induction hypothesis. This leaves us with four cases.

$$\text{T-ClosureValue}\quad \frac{\text{free-vars}(e) \setminus \overline{x} = \overline{x_f} = \text{dom}(\mathcal{F}_c)|_x \qquad \overline{r} = \overline{\text{free-provs}(\Gamma(x_f))},\ \text{free-provs}(e) = \text{dom}(\mathcal{F}_c)|_r \qquad \Sigma;\ \Gamma \vdash \varsigma_c : \mathcal{F}_c \qquad \Sigma;\ \Delta;\ \Gamma \natural \mathcal{F}_c,\ x_1 : \tau_1^{\text{SI}},\ \ldots,\ x_n : \tau_n^{\text{SI}} \vdash \boxed{e} : \tau_r^{\text{SI}} \Rightarrow \Gamma' \natural \mathcal{F}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\langle \varsigma_c,\ |x_1 : \tau_1^{\text{SI}},\ \ldots,\ x_n : \tau_n^{\text{SI}}|\ \rightarrow\ \tau_r^{\text{SI}}\ \{\ e\ \}\ \rangle} : (\tau_1^{\text{SI}},\ \ldots,\ \tau_n^{\text{SI}}) \xrightarrow{\mathcal{F}_c} \tau_r^{\text{SI}} \Rightarrow \Gamma}$$

For the T-ClosureValue case, firstly we want to show $\Sigma;\ \Gamma' \vdash \varsigma : \mathcal{F}_c$. This follows immediatedly from (2).
Then to finish the closure case, it suffices to show
$\Sigma;\ \bullet;\ \Gamma' \natural \mathcal{F}_c \vdash \boxed{e} : \tau_r^{\text{SI}} \Rightarrow \Gamma_o' \natural \mathcal{F}$, which follows immediately from Lemma E.18.

$$\text{T-Pointer}\quad \frac{\Sigma;\ \Gamma \vdash \mathcal{R}^{\square}[\pi] : \tau^{\text{XI}} \qquad {}^{\omega}\pi\ \in \Gamma(r)}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\texttt{ptr}\ \mathcal{R}^{\square}[\pi]} : \&r\ \omega\ \tau^{\text{XI}} \Rightarrow \Gamma}$$

If x was dropped, then $\Gamma'(x) = \Gamma(x)^{\dagger}$. Then the proof follows immediately from T-Dead.
If x was not dropped, then $\Gamma(x) = \Gamma'(x)$. All that is left to show is that that the referent remains well formed, and the loan ${}^{\omega}\pi$ is in $\Gamma'(r)$. The first condition follows from Lemma E.19. The second condition is immediate because the only potential changes allowed in the related environment to loan sets is emptying the loan sets of provenances if there's no references with the provenance in their type, and this particular reference is a reference with the provenance, so emptying the loan set is ruled out.

(2) The proof amounts to showing that $\forall x \in \text{dom}(\mathcal{F}_c), \Sigma;\ \bullet;\ \Gamma \natural \mathcal{F}_c \vdash \boxed{\varsigma(x)} : \mathcal{F}_c(x) \Rightarrow \Gamma \natural \mathcal{F}_c$ implies $\Sigma;\ \bullet;\ \Gamma' \natural \mathcal{F}_c \vdash \boxed{\varsigma(x)} : \mathcal{F}_c(x) \Rightarrow \Gamma' \natural \mathcal{F}_c$. This proceeds directly from applying (1), since $\Sigma;\ \bullet \vdash \Gamma \lesssim \Gamma'$ implies $\Sigma;\ \bullet \vdash \Gamma \natural \mathcal{F}_c \lesssim \Gamma' \natural \mathcal{F}_c$.

□

Lemma E.21 (Value Typing Fixed on Output Environments). *If* $\Sigma;\ \Delta;\ \Gamma \vdash \boxed{v} : \tau \Rightarrow \Gamma'$, *then* $\Sigma;\ \Delta;\ \Gamma' \vdash \boxed{v} : \tau \Rightarrow \Gamma'$.

Proof. Immediate by induction on the typing derivation. The only non immediate case is T-Pointer, where we also need to apply Lemma E.19. □

Lemma E.22 (Stack Validity is Invariant Under Loan Context Update). *If* $\Sigma \vdash \sigma : \Gamma$ *and* $\vdash \Sigma; \Delta; \Gamma[r \mapsto \{ \bar{\ell} \}]$ *and* $\Gamma(r) = \emptyset$ *then* $\Sigma \vdash \sigma : \Gamma[r \mapsto \{ \bar{\ell} \}]$.

Proof. We proceed by induction on the stack validity. There are two cases, WF-StackEmpty, and WF-StackFrame. WF-StackEmpty is impossible, since we already know that r is in Γ.

$$\textsc{WF-StackFrame}\ \frac{\Sigma \vdash \sigma : \Gamma \qquad \mathrm{dom}(\varsigma) = \mathrm{dom}(\mathcal{F})|_x \qquad \forall x \in \mathrm{dom}(\varsigma).\ \Sigma; \bullet; \Gamma \natural \mathcal{F} \vdash \boxed{(\sigma \natural \varsigma)(x)} : (\Gamma \natural \mathcal{F})(x) \Rightarrow \Gamma \natural \mathcal{F}}{\Sigma \vdash \sigma \natural \varsigma : \Gamma \natural \mathcal{F}}$$

$$\textsc{WF-StackEmpty}\ \frac{}{\Sigma \vdash \bullet : \bullet}$$

In the case of WF-StackFrame, we mostly just have to show that the values remain well typed in the updated environment. For the remaining Γ', if $r \in \Gamma'$, then we apply the induction hypothesis, otherwise we just apply the derivation from the premise.

To show that the values in the stack are still well typed in $\Gamma[r \mapsto \{\bar{\ell}\}]$, we proceed by induction on the typing derivation with Γ. The only interesting case is T-ClosureValue.

$$\textsc{T-ClosureValue}\ \frac{\text{free-vars}(e) \setminus \overline{x} = \overline{x_f} = \mathrm{dom}(\mathcal{F}_c)|_x \qquad \overline{r} = \overline{\text{free-provs}(\Gamma(x_f))},\ \text{free-provs}(e) = \mathrm{dom}(\mathcal{F}_c)|_r \qquad \Sigma; \Gamma \vdash \varsigma_c : \mathcal{F}_c \qquad \Sigma; \Delta; \Gamma \natural \mathcal{F}_c,\ x_1 : \tau_1^{\mathrm{SI}}, \ldots, x_n : \tau_n^{\mathrm{SI}} \vdash \boxed{e} : \tau_r^{\mathrm{SI}} \Rightarrow \Gamma' \natural \mathcal{F}}{\Sigma; \Delta; \Gamma \vdash \boxed{\langle \varsigma_c,\ |x_1 : \tau_1^{\mathrm{SI}}, \ldots, x_n : \tau_n^{\mathrm{SI}}|\ \rightarrow\ \tau_r^{\mathrm{SI}}\ \{ e \}\ \rangle} : (\tau_1^{\mathrm{SI}}, \ldots, \tau_n^{\mathrm{SI}}) \xrightarrow{\mathcal{F}_c} \tau_r^{\mathrm{SI}} \Rightarrow \Gamma}$$

For this case, we proceed by induction on the expression typing derivation. The interesting cases are all of the cases which use ownership safety: T-Move, T-Copy, T-Borrow, T-BorrowIndex, T-BorrowSlice, T-IndexCopy, and T-AssignDeref. What we want to know is that that ownership safety is preserved given the addition of these loans. In all of these cases cases, this is immediate because these places are by definition disjoint from the ones outside of the closure in the loan set. □

Lemma E.23 (Stack Validity is Preserved in Related Environments). *If* $\Sigma \vdash \sigma : \Gamma$ *and* $\Sigma; \bullet \vdash \Gamma \lesssim \Gamma'$, *then* $\Sigma \vdash \sigma : \Gamma'$.

Proof. We proceed by induction over the well typedness of the store.

$$\textsc{WF-StackFrame}\ \frac{\Sigma \vdash \sigma : \Gamma \qquad \mathrm{dom}(\varsigma) = \mathrm{dom}(\mathcal{F})|_x \qquad \forall x \in \mathrm{dom}(\varsigma).\ \Sigma; \bullet; \Gamma \natural \mathcal{F} \vdash \boxed{(\sigma \natural \varsigma)(x)} : (\Gamma \natural \mathcal{F})(x) \Rightarrow \Gamma \natural \mathcal{F}}{\Sigma \vdash \sigma \natural \varsigma : \Gamma \natural \mathcal{F}}$$

$$\textsc{WF-StackEmpty}\ \frac{}{\Sigma \vdash \bullet : \bullet}$$

The interesting case is when the stack is non empty. Then we have that $\Sigma \vdash \sigma : \Gamma'$ and want to show that $\Sigma \vdash \sigma \natural \varsigma : \Gamma' \natural \mathcal{F}$. The requirement on the domain is immediate since related environments have the same domains. What's left to show is that the values in the store remain well typed under the new environment. This follows from repeated applications of Lemma E.20 □

Lemma E.24 (Stack Validity is Preserved When Popping A Stack Frame). *If* $\Sigma \vdash \sigma \natural \varsigma : \Gamma \natural \mathcal{F}$, *then* $\Sigma \vdash \sigma : \Gamma$.

Proof. Immediate by inversion on WF-StackFrame which gives us $\Sigma \vdash \sigma : \Gamma$. □

Lemma E.25 (Stack Validity is Preserved under Well-Typed Extensions). *If* $\Sigma \vdash \sigma : \Gamma$ *and* $\Sigma;\ \Delta;\ \Gamma \vdash \boxed{v} : \tau^{SI} \Rightarrow \Gamma$, *then* $\Sigma \vdash \sigma,\ x \mapsto v : \Gamma,\ x\ :\ \tau^{SI}$.

Proof. This proof follows directly from the definition of WF-StackFrame.

$$\textsc{WF-StackFrame}\quad \frac{\Sigma \vdash \sigma : \Gamma \qquad \mathrm{dom}(\varsigma) = \mathrm{dom}(\mathcal{F})|_x \qquad \forall x \in \mathrm{dom}(\varsigma).\ \Sigma;\ \bullet;\ \Gamma \natural \mathcal{F} \vdash \boxed{(\sigma \natural \varsigma)(x)} : (\Gamma \natural \mathcal{F})(x) \Rightarrow \Gamma \natural \mathcal{F}}{\Sigma \vdash \sigma \natural \varsigma : \Gamma \natural \mathcal{F}}$$

In particular, inversion of WF-StackFrame on $\Sigma \vdash \sigma : \Gamma$ gives us well-formedness for the remainder of the stack, $\mathrm{dom}(\varsigma) = \mathrm{dom}(\mathcal{F})|_x$ and $\forall x \in \mathrm{dom}(\varsigma).\ \Sigma;\ \bullet;\ \Gamma \natural \mathcal{F} \vdash \boxed{(\sigma \natural \varsigma)(x)} : (\Gamma \natural \mathcal{F})(x) \Rightarrow \Gamma \natural \mathcal{F}$. We can then see that the well-formedness of the remainder of the stack is unaffected, and that the domains when extended with x remain equal. The last obligation is to show that the v is well-typed in the current stack typing, but we already have that from our premise. Thus, we can apply WF-StackFrame with the extended stack to get $\Sigma \vdash \sigma,\ x \mapsto v : \Gamma,\ x\ :\ \tau^{SI}$. □

Lemma E.26 (Values Are Well-Typed At Super-Types). *If* $\Sigma;\ \Delta;\ \Gamma\ \vdash\ \boxed{v}\ :\ \tau^{SI}\ \Rightarrow\ \Gamma_i$ *and* $\Delta;\ \Gamma_i \vdash +\lesssim \tau^{SI} \Rightarrow \tau^{SI\prime}\Gamma'$, *then* $\Sigma;\ \Delta;\ \Gamma' \vdash \boxed{v} : \tau^{SI\prime} \Rightarrow \Gamma'$.

Proof. We proceed by induction on the value typing relation.

In the case of T-Tuple, we need to apply the induction hypothesis for each entry which has a changed type, and Lemma E.7 for each entry which does not.

In the case of T-Array, we just apply the induction hypothesis to each entry.

$$\textsc{T-Pointer}\quad \frac{\Sigma;\ \Gamma \vdash \mathcal{R}^{\square}[\pi] : \tau^{\mathrm{XI}} \qquad {}^{\omega}\pi\ \in \Gamma(r)}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\mathtt{ptr}\ \mathcal{R}^{\square}[\pi]} : \&r\ \omega\ \tau^{\mathrm{XI}} \Rightarrow \Gamma}$$

For the T-Pointer case, we proceed by induction on the subtyping judgement. The only interesting cases are for reference types. From there, we proceed by induction on the outlives relation, for which the only interesting case is OL-LocalProvenances.

$$\textsc{OL-LocalProvenances}\quad \frac{\forall \pi : \&r_1\ \omega\ \tau \in \Gamma.\ \nexists r'.\ {}^{\omega} * \pi\ \in \Gamma(r') \qquad r_1\ \mathtt{occurs\ before}\ r_2\ \mathtt{in}\ \Gamma}{\Delta;\ \Gamma \vdash r_1 :> r_2 \Rightarrow \Gamma[r_2 \mapsto \{\ \Gamma(r_1) \cup \Gamma(r_2)\ \}]}$$

The T-Pointer case is immediate. We know that the referent type is preserved since we do not change any types in the context, and we know the loan is preserved since loan sets only grow.

In all other cases, we know the types cannot change, which means $\Gamma = \Gamma'$, so we are done. □

Lemma E.27 (Stack Validity is Preserved by Assignment). *If* $\Sigma \vdash \sigma : \Gamma$ *and* $\Sigma;\ \bullet;\ \Gamma \vdash \boxed{v} : \tau^{SI} \Rightarrow \Gamma_1$ *and* $\bullet;\ \Gamma_1 \vdash_{\mathtt{uniq}} p : \tau^{SX}$ *and* $\Delta;\ \Gamma_1 \vdash \tau^{SI} \lesssim \tau^{SX} \Rightarrow \Gamma'$, $\sigma \vdash p \Downarrow \mathcal{V}$, *and either* $\tau^{SX} = \tau^{SD}$ *or* $\bullet;\ \Gamma_1 \vdash_{\mathtt{uniq}} p \Rightarrow \{\ \overline{\ell}\ \}$ *and* $p = p^{\square}[x]$, *then:*

(1) $\Sigma \vdash \sigma[x \mapsto \mathcal{V}[v]] : \Gamma'[\pi \mapsto \tau^{SI}] \Vdash p$ *if* $p = \pi$, *and*
(2) $\Sigma \vdash \sigma[x \mapsto \mathcal{V}[v]] : \Gamma' \Vdash p$ *if* $\tau^{SX} = \tau^{SI}_o$.

Proof. First, note that by applying Lemma E.21 we get Σ; $\bullet$; $\Gamma_1 \vdash \boxed{v} : \tau^{\text{SI}} \Rightarrow \Gamma_1$ and by applying Lemma E.23, we get that $\Sigma \vdash \sigma : \Gamma_1$.

Next, note that $\Sigma \vdash \sigma : \Gamma'$ follows from applying Lemma E.8. Then we get Σ; $\bullet$; $\Gamma' \vdash \boxed{\mathcal{V}[v]} : \Gamma'(x) \Rightarrow \Gamma'$ by firstly applying Lemma E.26 to get Σ; $\bullet$; $\Gamma' \vdash \boxed{v} : \tau^{\text{SX}} \Rightarrow \Gamma'$, and then noting by a quick induction that $\sigma \vdash p \Downarrow \mathcal{V}$ and $\Sigma \vdash \sigma : \Gamma'$ means that all of the other values in $\mathcal{V}$ have their corresponding types, so plugging in v for the hole produces a well typed value. Therefore, we get that $\Sigma \vdash \sigma[x \mapsto \mathcal{V}[v]] : \Gamma'$.

Next, there are two cases, depending on whether $p = \pi$ or not. If not, then we just need to show that $\Sigma \vdash \sigma[x \mapsto \mathcal{V}[v]] : \Gamma' \Vdash p$. We know that $\tau^{\text{SX}} = \tau_o^{\text{SI}}$, so we just need to show all values in the store remain well typed, for which the only interesting case is pointers. For pointers, we need to show that their loan is preserved by this operation, but this is immediate: referants cannot contain dereferences, so the loan that the typing derivation was using could not have been a removed loan.

If $p = \pi$, the reasoning proceeds similarly, but we instead wish to prove that $\Sigma \vdash \sigma[x \mapsto \mathcal{V}[v]] : \Gamma'[\pi \mapsto \tau^{\text{SI}}] \Vdash p$. If $\tau^{\text{SX}} = \tau^{\text{SD}}$, then $\Gamma' \Vdash \pi = \Gamma'$ because any such loans that would be removed would be invalid since they would point to an uninitialized type. Otherwise, the reasoning proceeds exactly as above in the p case. □

Lemma E.28 (Garbage-Collecting Loans Preserves Stack Validity). *If* $\Sigma \vdash \sigma : \Gamma$, *then* $\Sigma \vdash \sigma : \textit{gc-loans}(\Gamma)$.

Proof. Consider the definitions of gc-loans(Γ) and R-Env.

$$\textsc{R-Env}\quad \frac{\begin{array}{c}\vdash \Sigma;\ \Delta;\ \Gamma \qquad \vdash \Sigma;\ \Delta;\ \Gamma' \qquad \text{dom}(\Gamma) = \text{dom}(\Gamma') \\ \forall x : \tau \in \Gamma'.\ \forall r \text{ that occurs in } \tau.\ \Gamma(r) = \Gamma'(r) \\ \forall r \in \text{dom}(\Gamma).\ \Gamma(r) = \Gamma'(r) \vee \Gamma'(r) = \emptyset \\ \forall \pi \in \text{dom}(\Gamma).\ \Gamma'(\pi) = \Gamma(\pi) \vee \Gamma'(\pi) = \Gamma(\pi)^{\dagger}\end{array}}{\Sigma;\ \Delta \vdash \Gamma \lesssim \Gamma'}$$

We know that gc-loans(Γ) can set a provenance's loan set to $\emptyset$ only if that provenance does not occur in the type of any bindings in Γ. This corresponds exactly to the definition of R-Env which says that all provenances which do occur in the type of a binding in Γ must have the same loan set, and otherwise they are also permitted to be $\emptyset$. Thus, we have directly that $\Sigma;\ \Delta \vdash \Gamma \lesssim \text{gc-loans}(\Gamma)$.

We can then apply Lemma E.23 to conclude $\Sigma \vdash \sigma : \text{gc-loans}(\Gamma)$. □

Lemma E.29 (Function Definitions Are Self-Contained).
If $\vdash \Sigma$; $\bullet$; Γ *and* $\Sigma(f) = \texttt{fn}\ f\langle\overline{\varphi}, \overline{\varrho}, \overline{\alpha}\rangle(x_1 : \tau_1^{SI}, \ldots, x_n : \tau_n^{SI}) \rightarrow \tau_r^{SI}\ \texttt{where}\ \overline{\varrho_1 : \varrho_2}\ \{\ e\ \}$, *then* Σ; $\overline{\varphi : \texttt{FRM}}$, $\overline{\varrho : \texttt{PRV}}$, $\overline{\varrho_1 :> \varrho_2}$, $\overline{\alpha : \star}$; $\Gamma \natural x_1 : \tau_1^{SI}, \ldots, x_n : \tau_n^{SI} \vdash \boxed{\texttt{framed}\ e} : \tau_f^{SI} \Rightarrow \Gamma$.

Proof. Begin by noting that WF-FunctionDefinition gives us that Σ; $\overline{\varphi : \texttt{FRM}}$, $\overline{\varrho : \texttt{PRV}}$, $\overline{\varrho_1 :> \varrho_2}$, $\overline{\alpha : \star}$; $\bullet \natural x_1 : \tau_1^{\text{SI}}, \ldots, x_n : \tau_n^{\text{SI}} \vdash \boxed{e} : \tau_f^{\text{SI}} \Rightarrow \Gamma'$. We also have by inspection of the typing rules that $\Gamma' = \bullet \natural \mathcal{F}'$ for some frame $\mathcal{F}'$. Then by T-Framed, it suffices to show that Σ; $\overline{\varphi : \texttt{FRM}}$, $\overline{\varrho : \texttt{PRV}}$, $\overline{\varrho_1 :> \varrho_2}$, $\overline{\alpha : \star}$; $\Gamma \natural x_1 : \tau_1^{\text{SI}}, \ldots, x_n : \tau_n^{\text{SI}} \vdash \boxed{e} : \tau_f^{\text{SI}} \Rightarrow \Gamma \natural \mathcal{F}'$. But note that this is immediate. The typing derivation with $\bullet$ and the current frame means that there's absolutely no reliance on context outside $x_1, \ldots x_n$, and these places are necessarily completely disjoint from places in Γ since any provenances in their types must be abstract. □

E.2 Progress

Lemma E.30 (Progress). *If* $\Sigma;\ \bullet;\ \Gamma \vdash \boxed{e} : \tau^{SI} \Rightarrow \Gamma'$ *and* $\Sigma \vdash \sigma : \Gamma$ *, then either* e *is a value,* e *is an* `abort!` $(\ldots)$ *, or* $\exists\, \sigma', e'.\ \Sigma \vdash (\sigma;\ \boxed{e}) \to (\sigma';\ \boxed{e'})$.

Proof. We proceed by induction on the derivation $\Sigma;\ \bullet;\ \Gamma \vdash \boxed{e} : \tau \Rightarrow \Gamma'$.

Case T-Move:

From premise:

$$\textsc{T-Move}\ \frac{\Delta;\ \Gamma \vdash_{\mathsf{uniq}} \pi \Rightarrow \{\ {}^{\mathsf{uniq}}\pi\ \} \qquad \Gamma(\pi) = \tau^{\mathrm{SI}} \qquad \mathsf{noncopyable}_{\Sigma}\ \tau^{\mathrm{SI}}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\pi} : \tau^{\mathrm{SI}} \Rightarrow \Gamma[\pi \mapsto \tau^{\mathrm{SI}^{\dagger}}]}$$

We want to step with:

$$\textsc{E-Move}\ \frac{\sigma \vdash \pi \Downarrow \_ \mapsto v}{\Sigma \vdash (\sigma;\ \boxed{\pi}) \to (\sigma[\pi \mapsto \mathsf{dead}];\ \boxed{v})}$$

Applying Lemma E.3 to $\Delta;\ \Gamma \vdash_{\mathsf{uniq}} \pi \Rightarrow \{\ {}^{\mathsf{uniq}}\pi\ \}$, $\Delta;\ \Gamma \vdash_{\mathsf{uniq}} \pi : \tau^{\mathrm{SI}}$ (from $\Gamma(\pi) = \tau^{\mathrm{SI}}$ by T-Place), and $\Sigma \vdash \sigma : \Gamma$ to conclude that $\sigma \vdash \pi \Downarrow \_ \mapsto v$. Thus, we can step with E-Move.

Case T-Copy:

From premise:

$$\textsc{T-Copy}\ \frac{\Delta;\ \Gamma \vdash_{\mathsf{shrd}} p \Rightarrow \{\ \overline{\ell}\ \} \qquad \Delta;\ \Gamma \vdash_{\mathsf{shrd}} p : \tau^{\mathrm{SI}} \qquad \mathsf{copyable}_{\Sigma}\ \tau^{\mathrm{SI}}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{p} : \tau^{\mathrm{SI}} \Rightarrow \Gamma}$$

We want to step with:

$$\textsc{E-Copy}\ \frac{\sigma \vdash p \Downarrow \_ \mapsto v}{\Sigma \vdash (\sigma;\ \boxed{p}) \to (\sigma;\ \boxed{v})}$$

Applying Lemma E.3 to $\Delta;\ \Gamma \vdash_{\mathsf{shrd}} p \Rightarrow \{\ \overline{\ell}\ \}$, $\Delta;\ \Gamma \vdash_{\mathsf{shrd}} p : \tau^{\mathrm{SI}}$, and $\Sigma \vdash \sigma : \Gamma$ to conclude that $\sigma \vdash p \Downarrow \_ \mapsto v$. Thus, we can step with E-Copy.

Case T-Borrow:

From premise:

$$\textsc{T-Borrow}\ \frac{\Gamma(r) = \emptyset \qquad \Delta;\ \Gamma \vdash_{\omega} p \Rightarrow \{\ \overline{\ell}\ \} \qquad \Delta;\ \Gamma \vdash_{\omega} p : \tau^{\mathrm{XI}}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\&r\ \omega\ p} : \&r\ \omega\ \tau^{\mathrm{XI}} \Rightarrow \Gamma[r \mapsto \{\ \overline{\ell}\ \}]}$$

We want to step with:

$$\textsc{E-Borrow}\ \frac{\sigma \vdash p \Downarrow \mathcal{R} \mapsto \_}{\Sigma \vdash (\sigma;\ \boxed{\&r\ \omega\ p}) \to (\sigma;\ \boxed{\mathtt{ptr}\ \mathcal{R}})}$$

Applying Lemma E.3 to $\Delta;\ \Gamma \vdash_{\omega} p \Rightarrow \{\ \overline{\ell}\ \}$, $\Delta;\ \Gamma \vdash_{\omega} p : \tau^{\mathrm{XI}}$, and $\Sigma \vdash \sigma : \Gamma$ to conclude that $\sigma \vdash p \Downarrow \mathcal{R} \mapsto \_$. Thus, we can step with E-Borrow.

Case T-BorrowIndex:

From premise:

$$\textsc{T-BorrowIndex}\ \frac{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e} : \mathtt{u32} \Rightarrow \Gamma' \qquad \Gamma'(r) = \emptyset \qquad \Delta;\ \Gamma' \vdash_\omega p \Rightarrow \{\,\overline{\ell}\,\} \qquad \Delta;\ \Gamma' \vdash_\omega p : \tau^{\mathrm{XI}} \qquad \tau^{\mathrm{XI}} = [\tau^{\mathrm{SI}};\ n] \vee \tau^{\mathrm{XI}} = [\tau^{\mathrm{SI}}]}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\&r\ \omega\ p[e]} : \&r\ \omega\ \tau^{\mathrm{SI}} \Rightarrow \Gamma'[r \mapsto \{\,\overline{\ell}\,\}]}$$

We proceed based on whether or not e is a value. If it is not, we can decompose our expression into the evaluation context $\&\rho\ \omega\ p[\square]$ and redex e. Then, by applying our induction hypothesis to the typing derivation for e, we know either that e is an `abort!` expression or it e steps to some e'. In the former case, we can step with E-EvalCtxAbort. In the latter case, we can plug e' back into our evaluation context and step with E-EvalCtx.

If e is a value, we would like to step with one of:

$$\textsc{E-BorrowIndex}\ \frac{\sigma \vdash p \Downarrow \mathcal{R} \mapsto [v_0, \ldots, v_n] \qquad 0 \le n_i \le n}{\Sigma \vdash (\sigma;\ \boxed{\&r\ \omega\ p[n_i]}) \rightarrow (\sigma;\ \boxed{\mathtt{ptr}\ \mathcal{R}[n_i]})}$$

$$\textsc{E-BorrowIndexOOB}\ \frac{\sigma \vdash p \Downarrow \_ \mapsto [v_0, \ldots, v_n] \qquad n_i < 0 \vee n_i > n}{\Sigma \vdash (\sigma;\ \boxed{\&r\ \omega\ *p[n_i]}) \rightarrow (\sigma;\ \boxed{\mathtt{abort!}(\text{“attempted to index out of bounds”})})}$$

Since e is a value, we can apply Lemma E.9 to get $\Sigma;\ \bullet \vdash \Gamma \lesssim \Gamma'$. Applying Lemma E.23, then gives us $\Sigma \vdash \sigma : \Gamma'$.

Then, we can apply Lemma E.3 to $\Delta;\ \Gamma' \vdash_\omega p \Rightarrow \{\,\overline{\ell}\,\}$, $\Delta;\ \Gamma' \vdash_\omega p : \tau^{\mathrm{XI}}$, and $\Sigma \vdash \sigma : \Gamma'$ to get $\sigma \vdash p \Downarrow \mathcal{R} \mapsto v$. By Lemma E.1, we know that $v = [v_0, \ldots, v_n]$ since the type tells us the shape of the resultant value.

Since we wish to step with one of E-BorrowIndex and E-BorrowIndexOOB, we should observe that we now have their shared requirement: $\sigma \vdash p \Downarrow \mathcal{R} \mapsto [v_0, \ldots, v_n]$. Their other obligations are a bounds check which together are a tautology (i.e. one of them must hold). Thus, we can step with the appropriate rule based on whether or not the bounds check succeeds.

Case T-BorrowSlice:

From premise:

$$\textsc{T-BorrowSlice}\ \frac{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\hat{e}_1} : \mathtt{u32} \Rightarrow \Gamma_1 \qquad \Sigma;\ \Delta;\ \Gamma_1 \vdash \boxed{\hat{e}_2} : \mathtt{u32} \Rightarrow \Gamma_2 \qquad \Gamma_2(r) = \emptyset \qquad \Delta;\ \Gamma_2 \vdash_\omega p \Rightarrow \{\,\overline{\ell}\,\} \qquad \Delta;\ \Gamma_2 \vdash_\omega p : [\tau^{\mathrm{SI}}]}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\&r\ \omega\ p[\hat{e}_1..\hat{e}_2]} : \&r\ \omega\ [\tau^{\mathrm{SI}}] \Rightarrow \Gamma_2[r \mapsto \{\,\overline{\ell}\,\}]}$$

The proof proceeds along similar lines as for T-BorrowIndex. We proceed based on whether or not e_1 and e_2 are values.

If e_1 is not a value, then we can decompose our whole expression into the evaluation context $\&\rho\ \omega\ p[\square..e_2]$ and redex e_1. Then, by applying our induction hypothesis to e_1, we know either that e_1 steps to some e_1' or is an `abort!` expression. In the former case, this satisfies our requirement since we can plug e' back into our evaluation context. In the latter case, we can step with E-EvalCtxAbort.

If e_1 is a value and e_2 is not a value, then we can decompose our whole expression into the evaluation context $\&\rho\ \omega\ p[v_1..\square]$ and redex e_2. Then, by applying our induction hypothesis to e_2,

we know either that e_2 steps to some e_2' or is an `abort!` expression. In the former case, this satisfies our requirement since we can plug e' back into our evaluation context. In the latter case, we can step with E-EvalCtxAbort.

If e_1 and e_2 are values, we would like to step with one of:

$$\textsc{E-BorrowSlice}\ \frac{\sigma \vdash p \Downarrow \mathcal{R} \mapsto [v_0, \ldots, v_n] \qquad 0 \le n_1 \le n_2 \le n}{\Sigma \vdash (\sigma; \boxed{\&r\ \omega\ p[n_1..n_2]}) \to (\sigma; \boxed{\texttt{ptr}\ \mathcal{R}[n_1..n_2]})}$$

$$\textsc{E-BorrowSliceOOB}\ \frac{\sigma \vdash p \Downarrow \_ \mapsto [v_0, \ldots, v_n] \qquad n_1 < 0 \vee n_1 > n \vee n_2 < 0 \vee n_2 > n \vee n_1 > n_2}{\Sigma \vdash (\sigma; \boxed{\&r\ \omega\ p[n_1..n_2]}) \to (\sigma; \boxed{\texttt{abort!}(\text{“attempted to slice out of bounds”})})}$$

Since e_1 is a value, we can apply Lemma E.9 to get $\Sigma;\ \bullet \vdash \Gamma \lesssim \Gamma_1$. Then, since e_2 is also a value, we can apply Lemma E.9 to get $\Sigma;\ \bullet \vdash \Gamma_1 \lesssim \Gamma_2$. Then, by transitivity, we get $\Sigma;\ \bullet \vdash \Gamma \lesssim \Gamma_2$. Then, applying Lemma E.23 gives us $\Sigma \vdash \sigma : \Gamma_2$.

Then, we can apply Lemma E.3 to $\Delta;\ \Gamma_2 \vdash_\omega p \Rightarrow \{\, \overline{\ell}\, \}$, $\Delta;\ \Gamma_2 \vdash_\omega p : [\tau^{\text{SI}}]$, and $\Sigma \vdash \sigma : \Gamma_2$ to get $\sigma \vdash p \Downarrow \mathcal{R} \mapsto v$. By Lemma E.1, we know that $v = [v_0, \ldots, v_n]$ since the type tells us the shape of the resultant value.

Since we wish to step with one of E-BorrowSlice and E-BorrowSliceOOB, we should observe that we now have their shared requirement: $\sigma \vdash p \Downarrow \mathcal{R} \mapsto [v_0, \ldots, v_n]$. Their other obligations are a bounds check which together are a tautology (i.e. one of them must hold). Thus, we can step with the appropriate rule based on whether or not the bounds check succeeds.

Case T-IndexCopy:

From premise:

$$\textsc{T-IndexCopy}\ \frac{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e} : \texttt{u32} \Rightarrow \Gamma' \qquad \Delta;\ \Gamma' \vdash_{\texttt{shrd}} p \Rightarrow \{\, \overline{\ell}\, \} \qquad \texttt{copyable}_\Sigma\ \tau^{\text{SI}} \qquad \Delta;\ \Gamma' \vdash_{\texttt{shrd}} p : \tau^{\text{XI}} \qquad \tau^{\text{XI}} = [\tau^{\text{SI}}; n] \vee \tau^{\text{XI}} = [\tau^{\text{SI}}]}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{p[e]} : \tau^{\text{SI}} \Rightarrow \Gamma'}$$

We proceed based on whether or not e is a value. If it is not, we can decompose our expression into the evaluation context $p[\square]$ and redex e. Then, by applying our induction hypothesis to e, we know either that e steps to some e' or is an `abort!` expression. In the former case, this satisfies our requirement since we can plug e' back into our evaluation context. In the latter case, we can step with E-EvalCtxAbort.

If e is a value, we would like to step with one of:

$$\textsc{E-IndexCopy}\ \frac{\sigma \vdash p \Downarrow \_ \mapsto [v_0, \ldots, v_{n_i}, \ldots, v_n]}{\Sigma \vdash (\sigma; \boxed{p[n_i]}) \to (\sigma; \boxed{v_{n_i}})} \qquad \textsc{E-IndexCopyOOB}\ \frac{\sigma \vdash p \Downarrow \_ \mapsto [v_0, \ldots, v_n] \qquad n_i < 0 \vee n_i > n}{\Sigma \vdash (\sigma; \boxed{p[n_i]}) \to (\sigma; \boxed{\texttt{abort!}(\text{“attempted to index out of bounds”})})}$$

Since e is a value, we can apply Lemma E.9 to get $\Sigma;\ \bullet \vdash \Gamma \lesssim \Gamma'$. Applying Lemma E.23, then gives us $\Sigma \vdash \sigma : \Gamma'$.

Then, we can apply Lemma E.3 to $\Delta;\ \Gamma' \vdash_\omega p \Rightarrow \{\, \overline{\ell}\, \}$, $\Delta;\ \Gamma' \vdash_\omega p : \tau^{\text{XI}}$, and $\Sigma \vdash \sigma : \Gamma'$ to get $\sigma \vdash p \Downarrow \_ \mapsto v$. By Lemma E.1, we know that $v = [v_0, \ldots, v_n]$ since the type tells us the shape of the resultant value.

Since we wish to step with one of E-IndexCopy and E-IndexCopyOOB, we should observe that we now have their shared requirement: $\sigma \vdash p \Downarrow \_ \mapsto [v_0, \ldots, v_n]$. Their other obligations are a bounds check which together are a tautology (i.e. one of them must hold). Thus, we can step with the appropriate rule based on whether or not the bounds check succeeds.

Case T-Seq:

From premise:

T-Seq

$$\frac{\Sigma;\, \Delta;\, \Gamma \vdash \boxed{e_1} : \tau_1^{\text{SI}} \Rightarrow \Gamma_1 \qquad \Sigma;\, \Delta;\, \text{gc-loans}(\Gamma_1) \vdash \boxed{e_2} : \tau_2^{\text{SI}} \Rightarrow \Gamma_2}{\Sigma;\, \Delta;\, \Gamma \vdash \boxed{e_1;\, e_2} : \tau_2^{\text{SI}} \Rightarrow \Gamma_2}$$

We proceed based on whether or not e_1 is a value. If it is not, we can decompose our expression into the evaluation context $\square;\ e_2$ and redex e_1. Then, by applying our induction hypothesis to e_1, we know either that e_1 steps to some e_1' or is an `abort`! expression. In the former case, this satisfies our requirement since we can plug e_1' back into our evaluation context. In the latter case, we can step with E-EvalCtxAbort.

If e_1 is a value, we can step with:

E-Seq

$$\frac{}{\Sigma \vdash (\sigma;\, \boxed{v;\, e}) \rightarrow (\sigma;\, \boxed{e})}$$

Case T-Branch:

From premise:

T-Branch

$$\frac{\begin{array}{c}\Sigma;\, \Delta;\, \Gamma \vdash \boxed{e_1} : \texttt{bool} \Rightarrow \Gamma_1 \qquad \Sigma;\, \Delta;\, \Gamma_1 \vdash \boxed{e_2} : \tau_2^{\text{SI}} \Rightarrow \Gamma_2 \\ \Sigma;\, \Delta;\, \Gamma_1 \vdash \boxed{e_3} : \tau_3^{\text{SI}} \Rightarrow \Gamma_3 \qquad \tau^{\text{SI}} = \tau_2^{\text{SI}} \vee \tau^{\text{SI}} = \tau_3^{\text{SI}} \\ \Delta;\, \Gamma_2 \vdash \tau_2^{\text{SI}} \lesssim \tau^{\text{SI}} \Rightarrow \Gamma_2' \qquad \Delta;\, \Gamma_3 \vdash \tau_3^{\text{SI}} \lesssim \tau^{\text{SI}} \Rightarrow \Gamma_3' \qquad \Gamma_2' \Cup \Gamma_3' = \Gamma'\end{array}}{\Sigma;\, \Delta;\, \Gamma \vdash \boxed{\texttt{if}\ e_1\ \{\ e_2\ \}\ \texttt{else}\ \{\ e_3\ \}} : \tau^{\text{SI}} \Rightarrow \Gamma'}$$

We proceed based on whether or not e_1 is a value. If it is not, we can decompose our expression into the evaluation context $\texttt{if}\ \square\ \{\ e_2\ \}\ \texttt{else}\ \{\ e_3\ \}$ and redex e_1. Then, by applying our induction hypothesis to e_1, we know either that e_1 steps to some e_1' or is an `abort`! expression. In the former case, this satisfies our requirement since we can plug e_1' back into our evaluation context. In the latter case, we can step with E-EvalCtxAbort.

If e_1 is a value, we would like to step with one of:

E-IfTrue

$$\frac{}{\Sigma \vdash (\sigma;\, \boxed{\texttt{if true}\ \{\ e_1\ \}\ \texttt{else}\ \{\ e_2\ \}}) \rightarrow (\sigma;\, \boxed{e_1})}$$

E-IfFalse

$$\frac{}{\Sigma \vdash (\sigma;\, \boxed{\texttt{if false}\ \{\ e_1\ \}\ \texttt{else}\ \{\ e_2\ \}}) \rightarrow (\sigma;\, \boxed{e_2})}$$

Since e_1 is a value, applying Lemma E.1 tells us that e_1 is either `true` or `false`. In the former case, we can step with E-IfTrue and in the latter case, we can step with E-IfFalse

Case T-Let:

From premise:

T-Let

$$\frac{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e_1} : \tau_1^{\text{SI}} \Rightarrow \Gamma_1 \qquad \Delta;\ \Gamma_1 \vdash \tau_1^{\text{SI}} \lesssim \tau_a^{\text{SI}} \Rightarrow \Gamma_1' \qquad \Sigma;\ \Delta;\ \text{gc-loans}(\Gamma_1',\ x\ :\ \tau_a^{\text{SI}}) \vdash \boxed{e_2} : \tau_2^{\text{SI}} \Rightarrow \Gamma_2,\ x\ :\ \tau^{\text{SD}}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\texttt{let}\ x : \tau_a^{\text{SI}} = e_1;\ e_2} : \tau_2^{\text{SI}} \Rightarrow \Gamma_2}$$

We proceed based on whether or not e_1 is a value. If it is not, we can decompose our expression into the evaluation context $\texttt{let}\ x : \tau_a^{\text{SI}} = \square;\ e_2$ and redex e_1. Then, by applying our induction hypothesis to e_1, we know either that e_1 steps to some e_1' or is an `abort!` expression. In the former case, this satisfies our requirement since we can plug e_1' back into our evaluation context. In the latter case, we can step with E-EvalCtxAbort.

If e_1 is a value, we can step with:

E-Let

$$\frac{}{\Sigma \vdash (\sigma;\ \boxed{\texttt{let}\ x : \tau_a^{\text{SI}} = v;\ e}) \rightarrow (\sigma,\ x \mapsto v;\ \boxed{\texttt{shift}\ e})}$$

Case T-LetProv:

From premise:

T-LetProv

$$\frac{\Sigma;\ \Delta;\ \Gamma,\ r \mapsto \{\} \vdash \boxed{e} : \tau^{\text{SI}} \Rightarrow \Gamma',\ r \mapsto \{\overline{\ell}\}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\texttt{letprov <}r\texttt{> \{ }e\texttt{ \}}} : \tau^{\text{SI}} \Rightarrow \Gamma'}$$

We proceed based on whether or not e is a value. If it is not, we can decompose our expression into the evaluation context $\texttt{letprov <}r\texttt{> \{ }\square\texttt{ \}}$ and redex e. Then, by applying our induction hypothesis to e, we know either that e steps to some e' or is an `abort!` expression. In the former case, this satisfies our requirement since we can plug e' back into our evaluation context. In the latter case, we can step with E-EvalCtxAbort.

If e is a value, we can step with:

E-LetProv

$$\frac{}{\Sigma \vdash (\sigma;\ \boxed{\texttt{letprov <}r\texttt{> \{ }v\texttt{ \}}}) \rightarrow (\sigma;\ \boxed{v})}$$

Case T-Assign:

From premise:

T-Assign

$$\frac{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e} : \tau^{\text{SI}} \Rightarrow \Gamma_1 \qquad \Gamma_1(\pi) = \tau^{\text{SX}} \qquad (\tau^{\text{SX}} = \tau^{\text{SD}} \vee \Delta;\ \Gamma_1 \vdash_{\texttt{uniq}} \pi \Rightarrow \{\ {}^{\texttt{uniq}}\pi\ \}) \qquad \Delta;\ \Gamma_1 \vdash \tau^{\text{SI}} \lesssim \tau^{\text{SX}} \Rightarrow \Gamma'}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\pi \coloneqq e} : \texttt{unit} \Rightarrow \Gamma'[\pi \mapsto \tau^{\text{SI}}] \Vdash \pi}$$

We proceed based on whether or not e is a value. If it is not, we can decompose our expression into the evaluation context $p \coloneqq \square$ and redex e. Then, by applying our induction hypothesis to e, we know either that e steps to some e' or is an `abort!` expression. In the former case, this satisfies our

requirement since we can plug e' back into our evaluation context. In the latter case, we can step with E-EvalCtxAbort.

If e is a value, we would like to step with:

$$\textsc{E-Assign}\ \frac{\sigma \vdash p \Downarrow \mathcal{V} \qquad p = p^{\square}[x]}{\Sigma \vdash (\sigma;\ \boxed{p := v}) \to (\sigma[x \mapsto \mathcal{V}[v]];\ \boxed{()})}$$

Since e is a value, we can apply Lemma E.9 to get $\Sigma;\ \bullet \vdash \Gamma \lesssim \Gamma'$. Applying Lemma E.23, then gives us $\Sigma \vdash \sigma : \Gamma'$.

Then, we can apply Lemma E.3 to $\Delta;\ \Gamma' \vdash_\omega p \Rightarrow \{\, \overline{\ell}\, \}$, $\Delta;\ \Gamma' \vdash_\omega p : \tau^{\text{XI}}$, and $\Sigma \vdash \sigma : \Gamma'$ to get $\sigma \vdash p \Downarrow \mathcal{R} \mapsto v$.

Finally, we apply Lemma E.4 to get $\sigma \vdash p \Downarrow \mathcal{V}$. This allows us to apply E-Assign.

Case T-ForArray:

From premise:

$$\textsc{T-ForArray}\ \frac{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e_1} : [\tau^{\text{SI}};\ n] \Rightarrow \Gamma_1 \qquad \Sigma;\ \Delta;\ \Gamma_1,\ x : \tau^{\text{SI}} \vdash \boxed{e_2} : \texttt{unit} \Rightarrow \Gamma_1,\ x : \tau^{\text{SD}}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\texttt{for}\ x\ \texttt{in}\ e_1\ \{\ e_2\ \}} : \texttt{unit} \Rightarrow \Gamma_1}$$

We proceed based on whether or not e_1 is a value. If it is not, we can decompose our expression into the evaluation context $\texttt{for}\ x\ \texttt{in}\ \square\ \{\ e_2\ \}$ and redex e_1. Then, by applying our induction hypothesis to e_1, we know either that e_1 steps to some e_1' or is an `abort!` expression. In the former case, this satisfies our requirement since we can plug e_1' back into our evaluation context. In the latter case, we can step with E-EvalCtxAbort.

If e_1 is a value, we would like to step with one of:

$$\textsc{E-ForArray}\ \frac{}{\Sigma \vdash (\sigma;\ \boxed{\texttt{for}\ x\ \texttt{in}\ [v_0, \ldots, v_n]\ \{\ e\ \}}) \to (\sigma,\ x \mapsto v_0;\ \boxed{\texttt{shift}\ e;\ \texttt{for}\ x\ \texttt{in}\ [v_1, \ldots, v_n]\ \{\ e\ \}})}$$

$$\textsc{E-ForEmptyArray}\ \frac{}{\Sigma \vdash (\sigma;\ \boxed{\texttt{for}\ x\ \texttt{in}\ []\ \{\ e\ \}}) \to (\sigma;\ \boxed{()})}$$

Since e_1 is a value, then by Lemma E.1, we know that e_1 is of the form $[v_1, \ldots, v_n]$. If $n > 0$, then we can step with E-ForArray, and if $n = 0$, then we can step with E-ForEmptyArray.

Case T-ForSlice:

From premise:

$$\textsc{T-ForSlice}\ \frac{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e_1} : \&\rho\ \omega\ [\tau^{\text{SI}}] \Rightarrow \Gamma_1 \qquad \Sigma;\ \Delta;\ \Gamma_1,\ x : \&\rho\ \omega\ \tau^{\text{SI}} \vdash \boxed{e_2} : \texttt{unit} \Rightarrow \Gamma_1,\ x : \tau_1^{\text{SX}}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\texttt{for}\ x\ \texttt{in}\ e_1\ \{\ e_2\ \}} : \texttt{unit} \Rightarrow \Gamma_2}$$

We proceed based on whether or not e_1 is a value. If it is not, we can decompose our expression into the evaluation context $\texttt{for}\ x\ \texttt{in}\ \square\ \{\ e_2\ \}$ and redex e_1. Then, by applying our induction hypothesis

to e_1, we know either that e_1 steps to some e_1' or is an `abort!` expression. In the former case, this satisfies our requirement since we can plug e_1' back into our evaluation context. In the latter case, we can step with E-EvalCtxAbort.

If e_1 is a value, we would like to step with one of:

E-ForSlice

$$\frac{\sigma \vdash \mathcal{R} \Downarrow \_ \mapsto [v_1, \ldots, v_i, \ldots, v_j, \ldots, v_n] \qquad i < j \qquad i' = i + 1}{\Sigma \vdash (\sigma; \boxed{\texttt{for } x \texttt{ in ptr } \mathcal{R}[i..j] \{ e \}}) \to (\sigma, x \mapsto \texttt{ptr } \mathcal{R}[i]; \boxed{\texttt{shift } e; \texttt{ for } x \texttt{ in ptr } \mathcal{R}[i'..j] \{ e \}})}$$

E-ForEmptySlice

$$\frac{}{\Sigma \vdash (\sigma; \boxed{\texttt{for } x \texttt{ in ptr } \pi[n..n] \{ e \}}) \to (\sigma; \boxed{()})}$$

If e_1 is a value, then by Lemma E.1, we know that e_1 is of the form $\texttt{ptr}\ \mathcal{R}[n_1..n_2]$. Further, by inversion of T-Pointer for the typing derivation of e_1, we get $\Sigma; \Gamma \vdash \mathcal{R}[i..j] : .$ By inversion of WF-RefSliceArray or WF-RefSliceSlice (one of which must apply since the referent ends in a slice), we know that $i \leq j$. If $i < j$, we step with E-ForSlice and if $i = j$, we step with E-ForEmptySlice.

Case T-Closure:

From premise:

T-Closure

$$\frac{\begin{array}{c}\text{free-vars}(e) \setminus \overline{x} = \overline{x_f} \qquad \text{free-nc-vars}_\Gamma(e) = \overline{x_{nc}} \qquad \overline{r} = \overline{\text{free-provs}(\Gamma(x_f))}, \text{ free-provs}(e) \\ \mathcal{F}_c = \overline{r \mapsto \Gamma(r)}, \overline{x_f : \Gamma(x_f)} \qquad \Sigma; \Delta; \Gamma[\overline{x_{nc} \mapsto \Gamma(x_{nc})^\dagger}] \uplus \mathcal{F}_c, x_1 : \tau_1^{\text{SI}}, \ldots, x_n : \tau_n^{\text{SI}} \vdash \boxed{e} : \tau_r^{\text{SI}} \Rightarrow \Gamma' \uplus \mathcal{F}\end{array}}{\Sigma; \Delta; \Gamma \vdash \boxed{|x_1 : \tau_1^{\text{SI}}, \ldots, x_n : \tau_n^{\text{SI}}| \to \tau_r^{\text{SI}} \{ e \}} : (\tau_1^{\text{SI}}, \ldots, \tau_n^{\text{SI}}) \xrightarrow{\mathcal{F}_c} \tau_r^{\text{SI}} \Rightarrow \Gamma'}$$

We want to step with:

E-Closure

$$\frac{\text{free-vars}(e) = \overline{x_f} \qquad \text{free-nc-vars}_\sigma(e) = \overline{x_{nc}} \qquad \varsigma_c = \sigma \mid_{\overline{x_f}}}{\Sigma \vdash (\sigma; \boxed{|x_1 : \tau_1^{\text{S}}, \ldots, x_n : \tau_n^{\text{S}}| \to \tau_r^{\text{S}} \{ e \}}) \to (\sigma[\overline{x_{nc} \mapsto \texttt{dead}}]; \boxed{\langle \varsigma_c, |x_1 : \tau_1^{\text{S}}, \ldots, x_n : \tau_n^{\text{S}}| \to \tau_r^{\text{S}} \{ e \} \rangle})}$$

Since $\text{free-vars}(\cdot)$ and $\text{free-nc-vars}_\sigma(\cdot)$ are total, we can always step with E-Closure.

Case T-App:

From premise:

T-App

$$\frac{\begin{array}{c}\overline{\Sigma; \Delta; \Gamma \vdash \Phi} \qquad \overline{\Delta; \Gamma \vdash \rho} \qquad \overline{\Sigma; \Delta; \Gamma \vdash \tau^{\text{SI}}} \\ \Sigma; \Delta; \Gamma \vdash \boxed{\hat{e}_f} : \forall <\overline{\varphi}, \overline{\varrho}, \overline{\alpha}> (\tau_1^{\text{SI}}, \ldots, \tau_n^{\text{SI}}) \xrightarrow{\Phi_c} \tau_f^{\text{SI}} \texttt{ where } \overline{\varrho_1 : \varrho_2} \Rightarrow \Gamma_0 \\ \forall i \in \{1 \ldots n\}. \Sigma; \Delta; \Gamma_{i-1} \vdash \boxed{\hat{e}_i} : \tau_i^{\text{SI}} \overline{[^\Phi/_\varphi]} \overline{[^\rho/_\varrho]} \overline{[^{\tau^{\text{SI}}}/_\alpha]} \Rightarrow \Gamma_i \qquad \Delta; \Gamma_n \vdash \overline{\varrho_2 \overline{[^\rho/_\varrho]} :> \varrho_1 \overline{[^\rho/_\varrho]}} \Rightarrow \Gamma_b\end{array}}{\Sigma; \Delta; \Gamma \vdash \boxed{\hat{e}_f ::<\overline{\Phi}, \overline{\rho}, \overline{\tau^{\text{SI}}}> (\hat{e}_1, \ldots, \hat{e}_n)} : \tau_f^{\text{SI}} \overline{[^\Phi/_\varphi]} \overline{[^\rho/_\varrho]} \overline{[^{\tau^{\text{SI}}}/_\alpha]} \Rightarrow \Gamma_b}$$

We proceed based on whether or not e_f is a value. If it is not, we can decompose our expression into the evaluation context $\square ::<\overline{\rho'}, \overline{\tau^{\text{SI}}}> (e_1, \ldots, e_n)$ and redex e_f. Then, by applying our induction hypothesis to e_f, we know either that e_f steps to some e_f' or is an `abort!` expression. In the former case, this satisfies our requirement since we can plug e_f' back into our evaluation context. In the latter case, we can step with E-EvalCtxAbort.

Next, we'll proceed based on whether or not each expression e_i is a value. If any of them are not, we can decompose our expression into the evaluation context $v_f ::< \overline{\rho'}, \overline{\tau^{\text{SI}}} > (v_1, \ldots, v_m, \square, e_1, \ldots, e_{n'})$ and redex e_i. Then, by applying our induction hypothesis to e_i, we know either that e_i steps to some e_i' or is an `abort!` expression. In the former case, this satisfies our requirement since we can plug e_i' back into our evaluation context. In the latter case, we can step with E-EvalCtxAbort.

If e_f is a value and every e_i is a value, we would like to step with one of:

E-AppClosure

$$\frac{v_f = \langle \varsigma_c,\ |x_1 : \tau_1^{\text{S}}, \ldots, x_n : \tau_n^{\text{S}}| \ \rightarrow\ \tau_r^{\text{S}}\ \{\ e\ \}\ \rangle}{\Sigma \vdash (\sigma;\ \boxed{v_f(v_1, \ldots, v_n)}) \rightarrow (\sigma \natural \varsigma_c,\ x_1 \mapsto v_1, \ldots, x_n \mapsto v_n;\ \boxed{\texttt{framed}\ e})}$$

E-AppFunction

$$\frac{\Sigma(f) = \texttt{fn}\ f < \overline{\varphi}, \overline{\varrho}, \overline{\alpha} > (x_1 : \tau_1^{\text{S}}, \ldots, x_n : \tau_n^{\text{S}}) \ \rightarrow\ \tau_r^{\text{S}}\ \texttt{where}\ \overline{\varrho : \varrho'}\ \{\ e\ \}}{\Sigma \vdash (\sigma;\ \boxed{f ::< \overline{\Phi}, \overline{r'}, \overline{\tau^{\text{S}}} > (v_1, \ldots, v_n)}) \rightarrow (\sigma \natural x_1 \mapsto v_1, \ldots, x_n \mapsto v_n;\ \boxed{\texttt{framed}\ e[\overline{\Phi}/\overline{\varphi}][\overline{r'}/\overline{\varrho}][\overline{\tau^{\text{S}}}/\overline{\alpha}]})}$$

Since e_f is a value, then by Lemma E.1, we know that it either has the form $\langle \sigma_c,\ |x_1 : \tau_1^{\text{SI}}, \ldots, x_n : \tau_n^{\text{SI}}| \ \rightarrow\ \tau_r^{\text{SI}}\ \{\ e\ \}\ \rangle$ or f. Then, since all of the e_i are values, then we can step using either E-AppClosure or E-AppFunction respectively.

Case T-Unit:

From premise:

T-Unit

$$\frac{}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{()} : \texttt{unit} \Rightarrow \Gamma}$$

By inspection of the value grammar, we know that ( ) is already a value.

Case T-u32:

From premise:

T-u32

$$\frac{}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{n} : \texttt{u32} \Rightarrow \Gamma}$$

By inspection of the value grammar, we know that n is already a value.

Case T-True:

From premise:

T-True

$$\frac{}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\texttt{true}} : \texttt{bool} \Rightarrow \Gamma}$$

By inspection of the value grammar, we know that `true` is already a value.

Case T-False:

From premise:

$$\textsc{T-False}\ \frac{}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\texttt{false}} : \texttt{bool} \Rightarrow \Gamma}$$

By inspection of the value grammar, we know that `false` is already a value.

Case T-Tuple:

From premise:

$$\textsc{T-Tuple}\ \frac{\forall i \in \{\ 1\ \ldots\ n\ \}.\ \Sigma;\ \Delta;\ \Gamma_{i-1} \vdash \boxed{\hat{e}_i} : \tau_i^{\text{SI}} \Rightarrow \Gamma_i}{\Sigma;\ \Delta;\ \Gamma_0 \vdash \boxed{(\hat{e}_1\ ,\ \ldots\ ,\ \hat{e}_n)} : (\tau_1^{\text{SI}}\ ,\ \ldots\ ,\ \tau_n^{\text{SI}}) \Rightarrow \Gamma_n}$$

We'll proceed based on whether or not each expression e_i is a value. If any of them are not, we can decompose our expression into the evaluation context $(v_1\ ,\ \ldots\ ,\ v_m\ ,\ \square\ ,\ e_1\ ,\ \ldots\ ,\ e_{n'})$ and redex e_i. Then, by applying our induction hypothesis to e_i, we know either that e_i steps to some e_i' or to an `abort!` expression. In either case, this satisfies our requirement, since we can plug e_i' back into our evaluation context.

If every expression e_i is a value, then the whole expression is a value by the definition of values.

Case T-Array:

From premise:

$$\textsc{T-Array}\ \frac{\forall i \in \{\ 1\ \ldots\ n\ \}.\ \Sigma;\ \Delta;\ \Gamma_{i-1} \vdash \boxed{\hat{e}_i} : \tau^{\text{SI}} \Rightarrow \Gamma_i}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{[\hat{e}_1\ ,\ \ldots\ ,\ \hat{e}_n]} : [\tau^{\text{SI}};\ n] \Rightarrow \Gamma_n}$$

We'll proceed based on whether or not each expression e_i is a value. If any of them are not, we can decompose our expression into the evaluation context $[v_1\ ,\ \ldots\ ,\ v_m\ ,\ \square\ ,\ e_1\ ,\ \ldots\ ,\ e_{n'}]$ and redex e_i. Then, by applying our induction hypothesis to e_i, we know either that e_i steps to some e_i' or to an `abort!` expression. In either case, this satisfies our requirement, since we can plug e_i' back into our evaluation context.

If every expression e_i is a value, then the whole expression is a value by the definition of values.

Case T-Abort:

From premise:

$$\textsc{T-Abort}\ \frac{}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\texttt{abort!(str)}} : \tau^{\text{SX}} \Rightarrow \Gamma'}$$

By definition, `abort!(` $\ldots$ `)` is an `abort!` expression.

Case T-Framed:

From premise:

$$\textsc{T-Framed}\ \frac{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e} : \tau^{\mathrm{SI}} \Rightarrow \Gamma' \natural \mathcal{F}'}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\texttt{framed}\ e} : \tau^{\mathrm{SI}} \Rightarrow \Gamma'}$$

We proceed based on whether or not e is a value. If it is not, we can decompose our expression into the evaluation context $\texttt{framed}\ \square$ and redex e. Then, by applying our induction hypothesis to e, we know either that e steps to some e' or to an `abort!` expression. In either case, this satisfies our requirement, since we can plug e' back into our evaluation context.

If e is a value, then we would like to step with:

$$\textsc{E-Framed}\ \frac{}{\Sigma \vdash (\sigma \natural \varsigma;\ \boxed{\texttt{framed}\ v}) \rightarrow (\sigma;\ \boxed{v})}$$

In order to do so, we need to know $x \in \mathrm{dom}(\sigma)$. Fortunately, we know from our assumption that $\Sigma \vdash \sigma : \Gamma$ (via WF-Stack). The premise of WF-Stack tells us that $\mathrm{dom}(\sigma) = \mathrm{dom}(\Gamma)$, and thus the $x \in \mathrm{dom}(\Gamma)$ from the premise of T-Framed is sufficient to tell us that $x \in \mathrm{dom}(\sigma)$. Thus, we can step with E-Framed.

Case T-Pointer:

From premise:

$$\textsc{T-Pointer}\ \frac{\Sigma;\ \Gamma \vdash \mathcal{R}^{\square}[\pi] : \tau^{\mathrm{XI}} \qquad {}^{\omega}\pi\ \in \Gamma(r)}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\texttt{ptr}\ \mathcal{R}^{\square}[\pi]} : \&r\ \omega\ \tau^{\mathrm{XI}} \Rightarrow \Gamma}$$

By inspection of the value grammar, we know that $\texttt{ptr}\ \pi$ is already a value.

Case T-ClosureValue:

From premise:

$$\textsc{T-ClosureValue}\ \frac{\begin{array}{c}\text{free-vars}(e) \setminus \overline{x} = \overline{x_f} = \mathrm{dom}(\mathcal{F}_c)|_x \qquad \overline{r} = \overline{\text{free-provs}(\Gamma(x_f))},\ \text{free-provs}(e) = \mathrm{dom}(\mathcal{F}_c)|_r \\ \Sigma;\ \Gamma \vdash \varsigma_c : \mathcal{F}_c \qquad \Sigma;\ \Delta;\ \Gamma \natural \mathcal{F}_c,\ x_1 : \tau_1^{\mathrm{SI}},\ \ldots,\ x_n : \tau_n^{\mathrm{SI}} \vdash \boxed{e} : \tau_r^{\mathrm{SI}} \Rightarrow \Gamma' \natural \mathcal{F}\end{array}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\langle \varsigma_c,\ |x_1 : \tau_1^{\mathrm{SI}},\ \ldots,\ x_n : \tau_n^{\mathrm{SI}}|\ \rightarrow\ \tau_r^{\mathrm{SI}}\ \{\ e\ \}\ \rangle} : (\tau_1^{\mathrm{SI}},\ \ldots,\ \tau_n^{\mathrm{SI}}) \xrightarrow{\mathcal{F}_c} \tau_r^{\mathrm{SI}} \Rightarrow \Gamma}$$

By inspection of the value grammar, we know that $\langle \sigma,\ |x_1 : \tau_1^{\mathrm{SI}},\ \ldots,\ x_n : \tau_n^{\mathrm{SI}}|\ \rightarrow\ \tau_r^{\mathrm{SI}}\ \{\ e\ \}\ \rangle$ is already a value.

Case T-Dead:

From premise:

$$\textsc{T-Dead}\ \frac{}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{v} : \tau^{\mathrm{SI}^{\dagger}} \Rightarrow \Gamma}$$

The type $\tau^{\mathrm{SI}^{\dagger}}$ is not in the grammar of τ^{SI}. Thus, we have a contradiction.

Case T-Drop:

From premise:

$$\textsc{T-Drop}\ \frac{\Gamma(\pi) = \tau_\pi^{\mathrm{SI}} \qquad \Sigma;\ \Delta;\ \Gamma[\pi \mapsto \tau_\pi^{\mathrm{SI}^\dagger}] \vdash \boxed{e} : \tau^{\mathrm{SX}} \Rightarrow \Gamma_f}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e} : \tau^{\mathrm{SX}} \Rightarrow \Gamma_f}$$

By R-Env, we have that $\Sigma;\ \bullet \vdash \Gamma \lesssim \Gamma[\pi \mapsto \tau_\pi^{\mathrm{SI}^\dagger}]$. Then, applying Lemma E.23 with $\Sigma \vdash \sigma : \Gamma$ (from our premise) gives us $\Sigma \vdash \sigma : \Gamma[\pi \mapsto \tau_\pi^{\mathrm{SI}^\dagger}]$. We can then apply our induction hypothesis to this and $\Sigma;\ \bullet;\ \Gamma[\pi \mapsto \tau_\pi^{\mathrm{SI}^\dagger}] \vdash \boxed{e} : \tau^{\mathrm{SX}} \Rightarrow \Gamma_f$ to reach our goal.

E.3 Preservation

Lemma E.31 (Preservation). *If* $\Sigma; \bullet; \Gamma \vdash \boxed{e} : \tau_1^{SI} \Rightarrow \Gamma_f$ *and* $\Sigma \vdash \sigma : \Gamma$ *and* $\Sigma \vdash (\sigma; \boxed{e}) \rightarrow (\sigma'; \boxed{e'})$ *, then there exists* Γ_i *such that* $\Sigma \vdash \sigma' : \Gamma_i$ *and* $\Sigma; \bullet; \Gamma_i \vdash \boxed{e'} : \tau_2^{SI} \Rightarrow \Gamma_f'$ *and* $\bullet; \Gamma_f' \vdash \tau_2^{SI} \lesssim \tau_1^{SI} \Rightarrow \Gamma_s$ *and there exists* Γ_o *such that* $\Gamma_f = \Gamma_s \Cup \Gamma_o$.

Proof. We proceed by induction on the derivation $\Sigma; \bullet; \Gamma \vdash \boxed{e} : \tau \Rightarrow \Gamma_f$

Case T-Move:

From premise:

$$\textsc{T-Move}\ \frac{\Delta; \Gamma \vdash_{\mathsf{uniq}} \pi \Rightarrow \{\ {}^{\mathsf{uniq}}\pi\ \} \quad \Gamma(\pi) = \tau^{\mathrm{SI}} \quad \mathsf{noncopyable}_\Sigma\ \tau^{\mathrm{SI}}}{\Sigma; \Delta; \Gamma \vdash \boxed{\pi} : \tau^{\mathrm{SI}} \Rightarrow \Gamma[\pi \mapsto \tau^{\mathrm{SI}^\dagger}]}$$

Since $e = \pi$, by inspection of the reduction rules, we know that e steps with the following rule:

$$\textsc{E-Move}\ \frac{\sigma \vdash \pi \Downarrow \_ \mapsto v}{\Sigma \vdash (\sigma; \boxed{\pi}) \rightarrow (\sigma[\pi \mapsto \mathsf{dead}]; \boxed{v})}$$

We then pick Γ_i to be $\boxed{\Gamma[\pi \mapsto \tau^{\mathrm{SI}^\dagger}]}$, and need to show:

$\Sigma \vdash \sigma[\pi \mapsto \mathsf{dead}] : \Gamma[\pi \mapsto \tau^{\mathrm{SI}^\dagger}]$	Applying Lemma E.23 to $\Sigma; \bullet \vdash \Gamma \lesssim \Gamma[\pi \mapsto \tau^{\mathrm{SI}^\dagger}]$ (immediate by R-Env) and $\Sigma \vdash \sigma : \Gamma$ (from premise) gives us $\Sigma \vdash \sigma : \Gamma[\pi \mapsto \tau^{\mathrm{SI}^\dagger}]$. Then, since we know $\Sigma; \bullet; \Gamma[\pi \mapsto \tau^{\mathrm{SI}^\dagger}] \vdash \boxed{\mathsf{dead}} : \tau^{\mathrm{SI}^\dagger} \Rightarrow \Gamma[\pi \mapsto \tau^{\mathrm{SI}^\dagger}]$ (by T-Dead), we can conclude $\Sigma \vdash \sigma[\pi \mapsto \mathsf{dead}] : \Gamma[\pi \mapsto \tau^{\mathrm{SI}^\dagger}]$.
$\Sigma; \bullet; \Gamma[\pi \mapsto \tau^{\mathrm{SI}^\dagger}] \vdash \boxed{v} : \tau^{\mathrm{SI}} \Rightarrow \Gamma[\pi \mapsto \tau^{\mathrm{SI}^\dagger}]$	Applying Lemma E.3 to $\bullet; \Gamma \vdash_{\mathsf{uniq}} \pi \Rightarrow \{\ {}^{\mathsf{uniq}}\pi\ \}$, $\bullet; \Gamma \vdash_{\mathsf{uniq}} \pi : \tau^{\mathrm{SI}}$ (immediate by TC-Place with $\Gamma(\pi) = \tau^{\mathrm{SI}}$), and $\Sigma \vdash \sigma : \Gamma$ gives us $\Sigma; \bullet; \Gamma \vdash \boxed{v} : \tau^{\mathrm{SI}} \Rightarrow \Gamma$. Then, by applying Lemma E.20 with $\Sigma; \bullet \vdash \Gamma \lesssim \Gamma[\pi \mapsto \tau^{\mathrm{SI}^\dagger}]$ (immediate by R-Env), we can conclude $\Sigma; \bullet; \Gamma[\pi \mapsto \tau^{\mathrm{SI}^\dagger}] \vdash \boxed{v} : \tau^{\mathrm{SI}} \Rightarrow \Gamma[\pi \mapsto \tau^{\mathrm{SI}^\dagger}]$.
$\bullet; \Gamma[\pi \mapsto \tau^{\mathrm{SI}^\dagger}] \vdash \tau^{\mathrm{SI}} \lesssim \tau^{\mathrm{SI}} \Rightarrow \Gamma[\pi \mapsto \tau^{\mathrm{SI}^\dagger}]$	Immediate by S-Refl.
$\exists \Gamma_o . \Gamma[\pi \mapsto \tau^{\mathrm{SI}^\dagger}] \Cup \Gamma_o = \Gamma[\pi \mapsto \tau^{\mathrm{SI}^\dagger}]$	$\Gamma_o = \Gamma[\pi \mapsto \tau^{\mathrm{SI}^\dagger}]$

Case T-Copy:

From premise:

$$\textsc{T-Copy}\quad \frac{\Delta;\ \Gamma \vdash_{\mathsf{shrd}} p \Rightarrow \{\,\overline{\ell}\,\} \qquad \Delta;\ \Gamma \vdash_{\mathsf{shrd}} p : \tau^{\mathrm{SI}} \qquad \mathsf{copyable}_{\Sigma}\ \tau^{\mathrm{SI}}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{p} : \tau^{\mathrm{SI}} \Rightarrow \Gamma}$$

Since $e = p$, by inspection of the reduction rules, we know that e steps with the following rule:

$$\textsc{E-Copy}\quad \frac{\sigma \vdash p \Downarrow \_ \mapsto v}{\Sigma \vdash (\sigma;\ \boxed{p}) \rightarrow (\sigma;\ \boxed{v})}$$

We then pick Γ_i to be $\boxed{\Gamma}$, and need to show:

$\Sigma \vdash \sigma : \Gamma$	Immediate from our premise.
$\Sigma;\ \bullet;\ \Gamma \vdash \boxed{v} : \tau^{\mathrm{SI}} \Rightarrow \Gamma$	Applying Lemma E.3 to $\bullet;\ \Gamma \vdash_{\mathsf{shrd}} p \Rightarrow \{\,\overline{\ell}\,\}$, $\bullet;\ \Gamma \vdash_{\mathsf{shrd}} p : \tau^{\mathrm{SI}}$, and $\Sigma \vdash \sigma : \Gamma$ gives us $\Sigma;\ \bullet;\ \Gamma \vdash \boxed{v} : \tau^{\mathrm{SI}} \Rightarrow \Gamma$.
$\bullet;\ \Gamma \vdash \tau^{\mathrm{SI}} \lesssim \tau^{\mathrm{SI}} \Rightarrow \Gamma$	Immediate by S-Refl.
$\exists \Gamma_o . \Gamma \ \Cup\ \Gamma_o = \Gamma$	$\Gamma_o = \Gamma$

Case T-Borrow:

From premise:

$$\textsc{T-Borrow}\quad \frac{\Gamma(r) = \emptyset \qquad \Delta;\ \Gamma \vdash_{\omega} p \Rightarrow \{\,\overline{\ell}\,\} \qquad \Delta;\ \Gamma \vdash_{\omega} p : \tau^{\mathrm{XI}}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\&r\ \omega\ p} : \&r\ \omega\ \tau^{\mathrm{XI}} \Rightarrow \Gamma[r \mapsto \{\,\overline{\ell}\,\}]}$$

Since $e = \&\rho\ \omega\ p$, by inspection of the reduction rules, we know that e steps with the following rule:

$$\textsc{E-Borrow}\quad \frac{\sigma \vdash p \Downarrow \mathcal{R} \mapsto \_}{\Sigma \vdash (\sigma;\ \boxed{\&r\ \omega\ p}) \rightarrow (\sigma;\ \boxed{\mathsf{ptr}\ \mathcal{R}})}$$

We then pick Γ_i to be $\boxed{\Gamma[r \mapsto \{\,\overline{\ell}\,\}]}$, and need to show:

$\Sigma \vdash \sigma : \Gamma[r \mapsto \{\,\overline{\ell}\,\}]$	Apply Lemma E.22 to $\Sigma \vdash \sigma : \Gamma$ (from our premise) and $\vdash \Sigma;\ \bullet;\ \Gamma[r \mapsto \{\overline{\ell}\}]$ (from our premise) gives us $\Sigma \vdash \sigma : \Gamma[r \mapsto \{\overline{\ell}\}]$.
$\Sigma;\ \bullet;\ \Gamma_i \vdash \boxed{\mathsf{ptr}\ \mathcal{R}} : \&r\ \omega\ \tau^{\mathrm{XI}} \Rightarrow \Gamma_i$	Applying Lemma E.5 to $\Sigma \vdash \sigma : \Gamma$, and $\sigma \vdash p \Downarrow \mathcal{R} \mapsto \_$ gives us $\Sigma;\ \Gamma \vdash \mathcal{R}^{\square}[\pi] : \tau^{\mathrm{XI}}$. Then, note that referent well-formedness does not depend on the contents of loan sets. This means we can also conclude $\Sigma;\ \Gamma[r \mapsto \{\,\overline{\ell}\,\}] \vdash \mathcal{R}^{\square}[\pi] : \tau^{\mathrm{XI}}$. Applying Lemma E.6 to $\Sigma \vdash \sigma : \Gamma$, $\sigma \vdash p \Downarrow \mathcal{R} \mapsto \_$, and $\bullet;\ \Gamma \vdash_{\omega} p \Rightarrow \{\,\overline{\ell}\,\}$ gives us $\mathcal{R} = \mathcal{R}^{\square}[\pi]$ and ${}^{\omega}\pi \in \{\,\overline{\ell}\,\}$. Finally, we can apply T-Pointer to the two facts above to get $\Sigma;\ \bullet;\ \Gamma[r \mapsto \{\,\overline{\ell}\,\}] \vdash \boxed{\mathsf{ptr}\ \mathcal{R}} : \&r\ \omega\ \tau^{\mathrm{XI}} \Rightarrow \Gamma[r \mapsto \{\,\overline{\ell}\,\}]$.
$\bullet;\ \Gamma_i \vdash \&r\ \omega\ \tau^{\mathrm{XI}} \lesssim \&r\ \omega\ \tau^{\mathrm{XI}} \Rightarrow \Gamma_i$	Immediate by S-Refl.
$\exists \Gamma_o . \Gamma_i \ \Cup\ \Gamma_o = \Gamma_i$	$\Gamma_o = \Gamma_i$

Case T-BorrowIndex:

From premise:

$$\textsc{T-BorrowIndex}\ \frac{\Sigma;\Delta;\Gamma \vdash \boxed{e} : \texttt{u32} \Rightarrow \Gamma' \qquad \Gamma'(r) = \emptyset \qquad \Delta;\Gamma' \vdash_\omega p \Rightarrow \{\,\overline{\ell}\,\} \qquad \Delta;\Gamma' \vdash_\omega p : \tau^{\text{XI}} \qquad \tau^{\text{XI}} = [\tau^{\text{SI}};\, n] \vee \tau^{\text{XI}} = [\tau^{\text{SI}}]}{\Sigma;\Delta;\Gamma \vdash \boxed{\&r\ \omega\ p[e]} : \&r\ \omega\ \tau^{\text{SI}} \Rightarrow \Gamma'[r \mapsto \{\,\overline{\ell}\,\}]}$$

Since $e = \&\rho\ \omega\ p[e_i]$, by inspection of the reduction rules, we know that e steps with the following rule:

$$\textsc{E-BorrowIndex}\ \frac{\sigma \vdash p \Downarrow \mathcal{R} \mapsto [v_0, \ldots, v_n] \qquad 0 \le n_i \le n}{\Sigma \vdash (\sigma;\ \boxed{\&r\ \omega\ p[n_i]}) \to (\sigma;\ \boxed{\texttt{ptr}\ \mathcal{R}[n_i]})}$$

$$\textsc{E-BorrowIndexOOB}\ \frac{\sigma \vdash p \Downarrow \_ \mapsto [v_0, \ldots, v_n] \qquad n_i < 0 \vee n_i > n}{\Sigma \vdash (\sigma;\ \boxed{\&r\ \omega\ *p[n_i]}) \to (\sigma;\ \boxed{\texttt{abort!}(\text{“attempted to index out of bounds”})})}$$

Then, for each possible rule, we'll pick Γ_i separately. The cases proceed as follows:

For E-BorrowIndex, we pick Γ_i to be $\boxed{\Gamma'[r \mapsto \{\,\overline{\ell}\,\}]}$, and need to show:

$\Sigma \vdash \sigma : \Gamma'[r \mapsto \{\,\overline{\ell}\,\}]$	Applying Lemma E.9 to the typing derivation (from T-BorrowIndex) for e (which we know is a value from E-BorrowIndex) gives us $\Sigma;\bullet \vdash \Gamma \lesssim \Gamma'$. Then, applying Lemma E.23 to $\Sigma;\bullet \vdash \Gamma \lesssim \Gamma'$ and $\Sigma \vdash \sigma : \Gamma$ (from premise) gives us $\Sigma \vdash \sigma : \Gamma'$. Finally, applying Lemma E.22 to $\Sigma \vdash \sigma : \Gamma'$ gives us $\Sigma \vdash \sigma : \Gamma'[r \mapsto \{\,\overline{\ell}\,\}]$.
$\Sigma;\bullet;\Gamma_i \vdash \boxed{\texttt{ptr}\ \mathcal{R}[n_i]} : \&r\ \omega\ \tau^{\text{SI}} \Rightarrow \Gamma_i$	Applying Lemma E.9 to the typing derivation (from T-BorrowIndex) for e (which we know is a value from E-BorrowIndex) gives us $\Sigma;\bullet \vdash \Gamma \lesssim \Gamma'$. Then, applying Lemma E.23 to $\Sigma;\bullet \vdash \Gamma \lesssim \Gamma'$ and $\Sigma \vdash \sigma : \Gamma$ (from premise) gives us $\Sigma \vdash \sigma : \Gamma'$. Applying Lemma E.5 to $\Sigma \vdash \sigma : \Gamma'$, and $\sigma \vdash p \Downarrow \mathcal{R} \mapsto [v_0, \ldots, v_n]$ gives us $\Sigma;\Gamma' \vdash \mathcal{R}^{\square}[\pi] : \tau^{\text{XI}}$. Then, note that referent well-formedness does not depend on the contents of loan sets. This means we can also conclude $\Sigma;\Gamma'[r \mapsto \{\,\overline{\ell}\,\}] \vdash \mathcal{R}^{\square}[\pi] : \tau^{\text{XI}}$. We can then apply WF-RefIndexArray or WF-RefIndexSlice to get $\Sigma;\Gamma'[r \mapsto \{\,\overline{\ell}\,\}] \vdash \mathcal{R}^{\square}[\pi][n_i] : \tau^{\text{XI}}$. Applying Lemma E.6 to $\Sigma \vdash \sigma : \Gamma'$, $\sigma \vdash p \Downarrow \mathcal{R} \mapsto [v_0, \ldots, v_n]$, and $\bullet;\Gamma' \vdash_\omega p \Rightarrow \{\,\overline{\ell}\,\}$ gives us $\mathcal{R} = \mathcal{R}[\pi]$ and ${}^{\omega}\pi \in \{\,\overline{\ell}\,\}$. Finally, we can apply T-Pointer to the two facts above to get $\Sigma;\bullet;\Gamma[r \mapsto \{\,\overline{\ell}\,\}] \vdash \boxed{\texttt{ptr}\ \mathcal{R}[n_i]} : \&r\ \omega\ \tau^{\text{XI}} \Rightarrow \Gamma[r \mapsto \{\,\overline{\ell}\,\}]$.
$\bullet;\Gamma_i \vdash \&r\ \omega\ \tau^{\text{SI}} \lesssim \&r\ \omega\ \tau^{\text{SI}} \Rightarrow \Gamma_i$	Immediate by S-Refl.
$\exists \Gamma_o.\Gamma_i \Cup \Gamma_o = \Gamma_i$	$\Gamma_o = \Gamma_i$

For E-BorrowIndexOOB, we pick Γ_i to be $\boxed{\Gamma}$, and need to show:

$\Sigma \vdash \sigma : \Gamma$	This is given as an assumption.
$\Sigma;\ \bullet;\ \Gamma \vdash \boxed{\texttt{abort!}(\ldots)} : \&r\ \omega\ \tau^{\text{SI}} \Rightarrow \Gamma$	An `abort!` expression is well-typed (at any type) via the rule T-Abort.
$\bullet;\ \Gamma \vdash \&r\ \omega\ \tau^{\text{SI}} \lesssim \&r\ \omega\ \tau^{\text{SI}} \Rightarrow \Gamma$	Immediate by S-Refl.
$\exists \Gamma_o . \Gamma_i \ \Cup\ \Gamma_o = \Gamma_i$	$\Gamma_o = \Gamma_i$

Case T-BorrowSlice:

From premise:

$$\text{T-BorrowSlice}\quad \frac{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\hat{e}_1} : \texttt{u32} \Rightarrow \Gamma_1 \qquad \Sigma;\ \Delta;\ \Gamma_1 \vdash \boxed{\hat{e}_2} : \texttt{u32} \Rightarrow \Gamma_2 \qquad \Gamma_2(r) = \emptyset \qquad \Delta;\ \Gamma_2 \vdash_\omega p \Rightarrow \{\ \overline{\ell}\ \} \qquad \Delta;\ \Gamma_2 \vdash_\omega p : [\tau^{\text{SI}}]}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\&r\ \omega\ p[\hat{e}_1..\hat{e}_2]} : \&r\ \omega\ [\tau^{\text{SI}}] \Rightarrow \Gamma_2[r \mapsto \{\ \overline{\ell}\ \}]}$$

Since $e = \&r\ \omega\ p[e_1..e_2]$, by inspection of the reduction rules, we know that e steps with the following rule:

$$\text{E-BorrowSlice}\quad \frac{\sigma \vdash p \Downarrow \mathcal{R} \mapsto [v_0, \ldots, v_n] \qquad 0 \le n_1 \le n_2 \le n}{\Sigma \vdash (\sigma;\ \boxed{\&r\ \omega\ p[n_1..n_2]}) \rightarrow (\sigma;\ \boxed{\texttt{ptr}\ \mathcal{R}[n_1..n_2]})}$$

$$\text{E-BorrowSliceOOB}\quad \frac{\sigma \vdash p \Downarrow \_ \mapsto [v_0, \ldots, v_n] \qquad n_1 < 0 \vee n_1 > n \vee n_2 < 0 \vee n_2 > n \vee n_1 > n_2}{\Sigma \vdash (\sigma;\ \boxed{\&r\ \omega\ p[n_1..n_2]}) \rightarrow (\sigma;\ \boxed{\texttt{abort!}(\text{“attempted to slice out of bounds”})})}$$

Then, for each possible rule, we'll pick Γ_i separately. The cases proceed as follows:

For E-BorrowSlice, we pick Γ_i to be $\boxed{\Gamma_2[r \mapsto \{\ \overline{\ell}\ \}]}$, and need to show:

$\Sigma \vdash \sigma : \Gamma_2[r \mapsto \{\,\overline{\ell}\,\}]$	Applying Lemma E.9 to the typing derivation (from T-BorrowSlice) for e_1 (which we know is a value from E-BorrowSlice) gives us $\Sigma;\ \bullet \vdash \Gamma \lesssim \Gamma_1$. Then, applying Lemma E.9 to the typing derivation (from T-BorrowSlice) for e_2 (which we know is a value from E-BorrowSlice) gives us $\Sigma;\ \bullet \vdash \Gamma_1 \lesssim \Gamma_2$. Then, by transitivity, we have $\Sigma;\ \bullet \vdash \Gamma \lesssim \Gamma_2$. Then, applying Lemma E.23 to $\Sigma;\ \bullet \vdash \Gamma \lesssim \Gamma_2$ and $\Sigma \vdash \sigma : \Gamma$ (from premise) gives us $\Sigma \vdash \sigma : \Gamma_2$. Finally, applying Lemma E.22 to $\Sigma \vdash \sigma : \Gamma_2$ gives us $\Sigma \vdash \sigma : \Gamma_2[r \mapsto \{\,\overline{\ell}\,\}]$.
$\Sigma;\ \bullet;\ \Gamma_i \vdash \boxed{\texttt{ptr}\ \mathcal{R}[n_1..n_2]} : \&r\ \omega\ [\tau^{\text{SI}}] \Rightarrow \Gamma_i$	Applying Lemma E.9 to the typing derivation (from T-BorrowSlice) for e_1 (which we know is a value from E-BorrowSlice) gives us $\Sigma;\ \bullet \vdash \Gamma \lesssim \Gamma_1$. Then, applying Lemma E.9 to the typing derivation (from T-BorrowSlice) for e_2 (which we know is a value from E-BorrowSlice) gives us $\Sigma;\ \bullet \vdash \Gamma_1 \lesssim \Gamma_2$. Then, by transitivity, we have $\Sigma;\ \bullet \vdash \Gamma \lesssim \Gamma_2$. Then, applying Lemma E.23 to $\Sigma;\ \bullet \vdash \Gamma \lesssim \Gamma_2$ and $\Sigma \vdash \sigma : \Gamma$ (from premise) gives us $\Sigma \vdash \sigma : \Gamma_2$. Applying Lemma E.5 to $\Sigma \vdash \sigma : \Gamma_2$, and $\sigma \vdash p \Downarrow \mathcal{R} \mapsto [v_0, \ldots, v_n]$ gives us $\Sigma;\ \Gamma_2 \vdash \mathcal{R}^{\square}[\pi] : \tau^{\text{XI}}$. Then, note that referent well-formedness does not depend on the contents of loan sets. This means we can also conclude $\Sigma;\ \Gamma_2[r \mapsto \{\,\overline{\ell}\,\}] \vdash \mathcal{R}^{\square}[\pi] : \tau^{\text{XI}}$. We can then apply WF-RefSliceArray or WF-RefSliceSlice to get $\Sigma;\ \Gamma'[r \mapsto \{\,\overline{\ell}\,\}] \vdash \mathcal{R}^{\square}[\pi][n_1..n_2] : \tau^{\text{XI}}$. Applying Lemma E.6 to $\Sigma \vdash \sigma : \Gamma_2$, $\sigma \vdash p \Downarrow \mathcal{R} \mapsto [v_0, \ldots, v_n]$, and $\bullet;\ \Gamma_2 \vdash_\omega p \Rightarrow \{\,\overline{\ell}\,\}$ gives us ${}^{\omega}\pi \in \{\,\overline{\ell}\,\}$. Finally, we can apply T-Pointer to the two facts above to get $\Sigma;\ \bullet;\ \Gamma[r \mapsto \{\,\overline{\ell}\,\}] \vdash \boxed{\texttt{ptr}\ \mathcal{R}[\pi][n_1..n_2]} : \&r\ \omega\ \tau^{\text{XI}} \Rightarrow \Gamma[r \mapsto \{\,\overline{\ell}\,\}]$.
$\bullet;\ \Gamma_i \vdash \&r\ \omega\ [\tau^{\text{SI}}] \lesssim \&r\ \omega\ [\tau^{\text{SI}}] \Rightarrow \Gamma_i$	Immediate by S-Refl.
$\exists \Gamma_o . \Gamma_i \Cup \Gamma_o = \Gamma_i$	$\Gamma_o = \Gamma_i$

For E-BorrowSliceOOB, we pick Γ_i to be $\boxed{\Gamma}$, and need to show:

$\Sigma \vdash \sigma : \Gamma$	This is given as an assumption.
$\Sigma;\ \bullet;\ \Gamma \vdash \boxed{\texttt{abort!(}\ldots\texttt{)}} : \&r\ \omega\ [\tau^{\text{SI}}] \Rightarrow \Gamma'$	An `abort!` expression is well-typed (at any type) via the rule T-Abort.
$\bullet;\ \Gamma \vdash \&r\ \omega\ [\tau^{\text{SI}}] \lesssim \&r\ \omega\ [\tau^{\text{SI}}] \Rightarrow \Gamma$	Immediate by S-Refl.
$\exists \Gamma_o . \Gamma_i \Cup \Gamma_o = \Gamma_i$	$\Gamma_o = \Gamma_i$

Case T-IndexCopy:

From premise:

$$\textsc{T-IndexCopy}\ \frac{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e} : \mathtt{u32} \Rightarrow \Gamma' \qquad \Delta;\ \Gamma' \vdash_{\mathtt{shrd}} p \Rightarrow \{\ \bar{\ell}\ \} \qquad \mathtt{copyable}_\Sigma\ \tau^{\mathrm{SI}} \qquad \Delta;\ \Gamma' \vdash_{\mathtt{shrd}} p : \tau^{\mathrm{XI}} \qquad \tau^{\mathrm{XI}} = [\tau^{\mathrm{SI}};\ n] \vee \tau^{\mathrm{XI}} = [\tau^{\mathrm{SI}}]}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{p[e]} : \tau^{\mathrm{SI}} \Rightarrow \Gamma'}$$

Since $e = p[e_i]$, by inspection of the reduction rules, we know that e steps with the following rule:

$$\textsc{E-IndexCopy}\ \frac{\sigma \vdash p \Downarrow \_ \mapsto [v_0, \ldots, v_{n_i}, \ldots, v_n]}{\Sigma \vdash (\sigma;\ \boxed{p[n_i]}) \rightarrow (\sigma;\ \boxed{v_{n_i}})}$$

We then pick Γ_i to be $\boxed{\Gamma'}$, and need to show:

$\Sigma \vdash \sigma : \Gamma'$	Applying Lemma E.9 to the typing derivation (from T-IndexCopy) for e (which we know is a value from E-IndexCopy) gives us $\Sigma;\ \bullet \vdash \Gamma \lesssim \Gamma'$. Then, applying Lemma E.23 to $\Sigma;\ \bullet \vdash \Gamma \lesssim \Gamma'$ and $\Sigma \vdash \sigma : \Gamma$ (from premise) gives us $\Sigma \vdash \sigma : \Gamma'$.
$\Sigma;\ \bullet;\ \Gamma' \vdash \boxed{v_{n_i}} : \tau^{\mathrm{SI}} \Rightarrow \Gamma'$	Applying Lemma E.9 to the typing derivation (from T-IndexCopy) for e (which we know is a value from E-IndexCopy) gives us $\Sigma;\ \bullet \vdash \Gamma \lesssim \Gamma'$. Then, applying Lemma E.23 to $\Sigma;\ \bullet \vdash \Gamma \lesssim \Gamma'$ and $\Sigma \vdash \sigma : \Gamma$ (from premise) gives us $\Sigma \vdash \sigma : \Gamma'$. Applying Lemma E.3 to $\bullet;\ \Gamma' \vdash_{\mathtt{shrd}} p \Rightarrow \{\ \bar{\ell}\ \}$, $\bullet;\ \Gamma' \vdash_{\mathtt{shrd}} p : \tau^{\mathrm{SI}}$, and $\Sigma \vdash \sigma : \Gamma'$ T-Slice (based on whether $\tau^{\mathrm{XI}} = [\tau^{\mathrm{SI}};\ n]$ or $[\tau^{\mathrm{SI}}]$ respectively), we get $\forall i \in \{\ 1 \ldots n\ \}.\ \Sigma;\ \Delta;\ \Gamma' \vdash \boxed{v_i} : \tau^{\mathrm{SI}} \Rightarrow \Gamma'$ (after accounting for the fact that the constituent expressions are values and the output environment matches the input environment). Thus, we can pick out specifically that $\Sigma;\ \Delta;\ \Gamma' \vdash \boxed{v_i} : \tau^{\mathrm{SI}} \Rightarrow \Gamma'$.
$\bullet;\ \Gamma' \vdash \tau^{\mathrm{SI}} \lesssim \tau^{\mathrm{SI}} \Rightarrow \Gamma'$	Immediate by S-Refl.
$\exists \Gamma_o . \Gamma' \Cup \Gamma_o = \Gamma'$	$\Gamma_o = \Gamma'$

Case T-Seq:

From premise:

$$\textsc{T-Seq}\ \frac{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e_1} : \tau_1^{\mathrm{SI}} \Rightarrow \Gamma_1 \qquad \Sigma;\ \Delta;\ \mathrm{gc\text{-}loans}(\Gamma_1) \vdash \boxed{e_2} : \tau_2^{\mathrm{SI}} \Rightarrow \Gamma_2}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e_1;\ e_2} : \tau_2^{\mathrm{SI}} \Rightarrow \Gamma_2}$$

Since $e = e_1;\ e_2$, by inspection of the reduction rules, we know that e steps with the following rule:

$$\textsc{E-Seq}\ \frac{}{\Sigma \vdash (\sigma;\ \boxed{v;\ e}) \rightarrow (\sigma;\ \boxed{e})}$$

We then pick Γ_i to be $\boxed{\mathrm{gc\text{-}loans}(\Gamma_1)}$, and need to show:

$\Sigma \vdash \sigma : \text{gc-loans}(\Gamma_1)$	Applying Lemma E.9 to the typing derivation (from T-Seq) for e_1 (which we know is a value from E-Seq) gives us $\Sigma; \bullet \vdash \Gamma \lesssim \Gamma_1$. Applying Lemma E.23 with this and $\Sigma \vdash \sigma : \Gamma$ (from premise) gives us $\Sigma \vdash \sigma : \Gamma_1$. Then, applying Lemma E.28 gives us $\Sigma \vdash \sigma : \text{gc-loans}(\Gamma_1)$.
$\Sigma; \bullet; \text{gc-loans}(\Gamma_1) \vdash \boxed{e_2} : \tau_2^{\text{SI}} \Rightarrow \Gamma_2$	Immediate from the premise of T-Seq.
$\bullet; \Gamma_2 \vdash \tau_2^{\text{SI}} \lesssim \tau_2^{\text{SI}} \Rightarrow \Gamma_2$	Immediate by S-Refl.
$\exists \Gamma_o . \Gamma_2 \Cup \Gamma_o = \Gamma_2$	$\Gamma_o = \Gamma_2$

Case T-Branch:

From premise:

$$\textsc{T-Branch}\ \frac{\Sigma; \Delta; \Gamma \vdash \boxed{e_1} : \texttt{bool} \Rightarrow \Gamma_1 \qquad \Sigma; \Delta; \Gamma_1 \vdash \boxed{e_2} : \tau_2^{\text{SI}} \Rightarrow \Gamma_2 \qquad \Sigma; \Delta; \Gamma_1 \vdash \boxed{e_3} : \tau_3^{\text{SI}} \Rightarrow \Gamma_3 \qquad \tau^{\text{SI}} = \tau_2^{\text{SI}} \vee \tau^{\text{SI}} = \tau_3^{\text{SI}} \qquad \Delta; \Gamma_2 \vdash \tau_2^{\text{SI}} \lesssim \tau^{\text{SI}} \Rightarrow \Gamma_2' \qquad \Delta; \Gamma_3 \vdash \tau_3^{\text{SI}} \lesssim \tau^{\text{SI}} \Rightarrow \Gamma_3' \qquad \Gamma_2' \Cup \Gamma_3' = \Gamma'}{\Sigma; \Delta; \Gamma \vdash \boxed{\texttt{if}\ e_1\ \{\ e_2\ \}\ \texttt{else}\ \{\ e_3\ \}} : \tau^{\text{SI}} \Rightarrow \Gamma'}$$

Since $e = \texttt{if}\ e_1\ \{\ e_2\ \}\ \texttt{else}\ \{\ e_3\ \}$, by inspection of the reduction rules, we know that e steps with the following rule:

$$\textsc{E-IfTrue}\ \frac{}{\Sigma \vdash (\sigma; \boxed{\texttt{if true}\ \{\ e_1\ \}\ \texttt{else}\ \{\ e_2\ \}}) \rightarrow (\sigma; \boxed{e_1})} \qquad \textsc{E-IfFalse}\ \frac{}{\Sigma \vdash (\sigma; \boxed{\texttt{if false}\ \{\ e_1\ \}\ \texttt{else}\ \{\ e_2\ \}}) \rightarrow (\sigma; \boxed{e_2})}$$

Then, for each possible rule, we'll pick Γ_i separately. The cases proceed as follows:
For E-IfTrue, we pick Γ_i to be $\boxed{\Gamma_1}$, and need to show:

$\Sigma \vdash \sigma : \Gamma_1$	Applying Lemma E.9 to the typing derivation (from T-Branch) for e_1 (which we know is a value from E-IfTrue) gives us $\Sigma; \bullet \vdash \Gamma \lesssim \Gamma_1$. Then, applying Lemma E.23 with this and $\Sigma \vdash \sigma : \Gamma$ (from premise) gives us $\Sigma \vdash \sigma : \Gamma_1$.
$\Sigma; \bullet; \Gamma_1 \vdash \boxed{e_2} : \tau^{\text{SI}} \Rightarrow \Gamma_2$	Immediate from premise of T-Branch.
$\bullet; \Gamma_2 \vdash \tau_2^{\text{SI}} \lesssim \tau_2^{\text{SI}} \Rightarrow \Gamma_2'$	Immediate from premise of T-Branch.
$\exists \Gamma_o . \Gamma_2' \Cup \Gamma_o = \Gamma'$	$\Gamma_o = \Gamma_3'$

For E-IfFalse, we pick Γ_i to be $\boxed{\Gamma_1}$, and need to show:

$\Sigma \vdash \sigma : \Gamma_1$	Applying Lemma E.9 to the typing derivation (from T-Branch) for e_1 (which we know is a value from E-IfFalse) gives us $\Sigma; \bullet \vdash \Gamma \lesssim \Gamma_1$. Then, applying Lemma E.23 with this and $\Sigma \vdash \sigma : \Gamma$ (from premise) gives us $\Sigma \vdash \sigma : \Gamma_1$.
$\Sigma; \bullet; \Gamma_1 \vdash \boxed{e_3} : \tau_3^{\text{SI}} \Rightarrow \Gamma_3$	Immediate from premise of T-Branch.
$\bullet; \Gamma_3 \vdash \tau_3^{\text{SI}} \lesssim \tau^{\text{SI}} \Rightarrow \Gamma_3'$	Immediate from premise of T-Branch.
$\exists \Gamma_o . \Gamma_3' \Cup \Gamma_o = \Gamma'$	$\Gamma_o = \Gamma_2'$ (note that $\Cup$ commutes)

Case T-Assign:

From premise:

$$\textsc{T-Assign}\ \frac{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e} : \tau^{\mathrm{SI}} \Rightarrow \Gamma_1 \qquad \Gamma_1(\pi) = \tau^{\mathrm{SX}} \qquad (\tau^{\mathrm{SX}} = \tau^{\mathrm{SD}} \vee \Delta;\ \Gamma_1 \vdash_{\mathsf{uniq}} \pi \Rightarrow \{\ {}^{\mathsf{uniq}}\pi\ \}) \qquad \Delta;\ \Gamma_1 \vdash \tau^{\mathrm{SI}} \lesssim \tau^{\mathrm{SX}} \Rightarrow \Gamma'}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\pi := e} : \mathsf{unit} \Rightarrow \Gamma'[\pi \mapsto \tau^{\mathrm{SI}}] \Subset \pi}$$

Since $e = \pi := e_a$, by inspection of the reduction rules, we know that e steps with the following rule:

$$\textsc{E-Assign}\ \frac{\sigma \vdash p \Downarrow \mathcal{V} \qquad p = p^{\square}[x]}{\Sigma \vdash (\sigma;\ \boxed{p := v}) \rightarrow (\sigma[x \mapsto \mathcal{V}[v]];\ \boxed{()})}$$

We then pick Γ_i to be $\boxed{\Gamma'[\pi \mapsto \tau^{\mathrm{SI}}] \Subset \pi}$, and need to show:

$\Sigma \vdash \sigma[\pi \mapsto \mathcal{V}[v]] : \Gamma'[\pi \mapsto \tau^{\mathrm{SI}}] \Subset \pi$	Applying Lemma E.27 to $\Sigma \vdash \sigma : \Gamma$ (from our premise), Σ; $\bullet$; $\Gamma \vdash \boxed{v} : \tau^{\mathrm{SI}} \Rightarrow \Gamma_1$ (from premise of T-Assign and knowledge that e is a value from E-Assign), $\bullet$; $\Gamma_1 \vdash_{\mathsf{uniq}} \pi : \tau^{\mathrm{SX}}$ (immediate by TC-Place on $\Gamma_1(\pi) = \tau^{\mathrm{SX}}$), $\bullet$; $\Gamma_1 \vdash \tau^{\mathrm{SI}} \lesssim \tau^{\mathrm{SX}} \Rightarrow \Gamma'$ (from premise of T-Assign), $\sigma \vdash \pi \Downarrow \mathcal{V}$ (from premise of E-Assign), and $\tau^{\mathrm{SX}} = \tau^{\mathrm{SD}} \vee \bullet$; $\Gamma_1 \vdash_{\mathsf{uniq}} \pi \Rightarrow \{\ {}^{\mathsf{uniq}}\pi\ \}$ (from premise of T-Assign) gives us $\Sigma \vdash \sigma[x \mapsto \mathcal{V}[v]] : \Gamma'[\pi \mapsto \tau^{\mathrm{SI}}] \Subset \pi$.
$\Sigma;\ \bullet;\ \Gamma'[\pi \mapsto \tau^{\mathrm{SI}}] \Subset \pi \vdash \boxed{()} : \mathsf{unit} \Rightarrow \Gamma'[\pi \mapsto \tau^{\mathrm{SI}}] \Subset \pi$	Immediate by T-Unit.
$\bullet;\ \Gamma'[\pi \mapsto \tau^{\mathrm{SI}}] \Subset \pi \vdash \mathsf{unit} \lesssim \mathsf{unit} \Rightarrow \Gamma'[\pi \mapsto \tau^{\mathrm{SI}}] \Subset \pi$	Immediate by S-Refl.
$\exists \Gamma_o.\Gamma'[\pi \mapsto \tau^{\mathrm{SI}}] \Subset \pi \Cup \Gamma_o = \Gamma'[\pi \mapsto \tau^{\mathrm{SI}}] \Subset \pi$	$\Gamma_o = \Gamma'[\pi \mapsto \tau^{\mathrm{SI}}] \Subset \pi$

Case T-AssignDeref:

From premise:

$$\textsc{T-AssignDeref}\ \frac{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e} : \tau_n^{\mathrm{SI}} \Rightarrow \Gamma_1 \qquad \Delta;\ \Gamma_1 \vdash_{\mathsf{uniq}} p : \tau_o^{\mathrm{SI}} \qquad \Delta;\ \Gamma_1 \vdash_{\mathsf{uniq}} p \Rightarrow \{\ \bar{\ell}\ \} \qquad \Delta;\ \Gamma_1 \vdash \tau_n^{\mathrm{SI}} \lesssim \tau_o^{\mathrm{SI}} \Rightarrow \Gamma'}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{p := e} : \mathsf{unit} \Rightarrow \Gamma' \Subset p}$$

Since $e = p := e_a$, by inspection of the reduction rules, we know that e steps with the following rule:

$$\textsc{E-Assign}\ \frac{\sigma \vdash p \Downarrow \mathcal{V} \qquad p = p^{\square}[x]}{\Sigma \vdash (\sigma;\ \boxed{p := v}) \rightarrow (\sigma[x \mapsto \mathcal{V}[v]];\ \boxed{()})}$$

We then pick Γ_i to be $\boxed{\Gamma' \Subset p}$, and need to show:

$\Sigma \vdash \sigma[\pi \mapsto \mathcal{V}[v]] : \Gamma' \Subset p$	Applying Lemma E.27 to $\Sigma \vdash \sigma : \Gamma$ (from our premise), Σ; $\bullet$; $\Gamma \vdash \boxed{v} : \tau^{\text{SI}} \Rightarrow \Gamma_1$ (from premise of T-AssignDeref and knowledge that e is a value from E-Assign), $\bullet$; $\Gamma_1 \vdash_{\texttt{uniq}} p : \tau^{\text{SX}}$ (from premise of T-AssignDeref), $\bullet$; $\Gamma_1 \vdash \tau^{\text{SI}} \lesssim \tau^{\text{SX}} \Rightarrow \Gamma'$ (from premise of T-AssignDeref), $\sigma \vdash p \Downarrow \mathcal{V}$ (from premise of E-Assign), and $\bullet$; $\Gamma_1 \vdash_{\texttt{uniq}} p \Rightarrow \{\ {}^{\texttt{uniq}}\pi\ \}$ (from premise of T-AssignDeref) gives us $\Sigma \vdash \sigma[x \mapsto \mathcal{V}[v]] : \Gamma' \Subset p$.
Σ; $\bullet$; $\Gamma' \Subset p \vdash \boxed{()} : \texttt{unit} \Rightarrow \Gamma' \Subset p$	Immediate by T-Unit.
$\bullet$; $\Gamma' \Subset p \vdash \texttt{unit} \lesssim \texttt{unit} \Rightarrow \Gamma' \Subset p$	Immediate by S-Refl.
$\exists \Gamma_o . \Gamma' \Subset p \Cup \Gamma_o = \Gamma' \Subset p$	$\Gamma_o = \Gamma' \Subset p$

Case T-Let:

From premise:

$$\text{T-Let}\quad \frac{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e_1} : \tau_1^{\text{SI}} \Rightarrow \Gamma_1 \qquad \Delta;\ \Gamma_1 \vdash \tau_1^{\text{SI}} \lesssim \tau_a^{\text{SI}} \Rightarrow \Gamma_1' \qquad \Sigma;\ \Delta;\ \text{gc-loans}(\Gamma_1',\ x\ :\ \tau_a^{\text{SI}}) \vdash \boxed{e_2} : \tau_2^{\text{SI}} \Rightarrow \Gamma_2,\ x\ :\ \tau^{\text{SD}}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\texttt{let}\ x : \tau_a^{\text{SI}} = e_1;\ e_2} : \tau_2^{\text{SI}} \Rightarrow \Gamma_2}$$

Since $e = \texttt{let}\ x : \tau^{\text{SI}} = e_1;\ e_2$, by inspection of the reduction rules, we know that e steps with the following rule:

$$\text{E-Let}\quad \frac{}{\Sigma \vdash (\sigma;\ \boxed{\texttt{let}\ x : \tau_a^{\text{SI}} = v;\ e}) \rightarrow (\sigma,\ x \mapsto v;\ \boxed{\texttt{shift}\ e})}$$

We then pick Γ_i to be $\boxed{\text{gc-loans}(\Gamma_1',\ x\ :\ \tau_a^{\text{SI}})}$, and need to show:

$\Sigma \vdash \sigma,\ x \mapsto v : \text{gc-loans}(\Gamma_1',\ x\ :\ \tau_a^{\text{SI}})$	Applying Lemma E.9 to the typing derivation (from T-Let) for e_1 (which we know is a value from E-Let) gives us Σ; $\bullet \vdash \Gamma \lesssim \Gamma_1$. Then, we can apply Lemma E.23 to get $\Sigma \vdash \sigma : \Gamma_1$. Then, applying Lemma E.8 to $\bullet$; $\Gamma_1 \vdash \tau_1^{\text{SI}} \lesssim \tau_a^{\text{SI}} \Rightarrow \Gamma_1'$ (from premise of T-Let) gives us $\Sigma \vdash \sigma : \Gamma_1'$. We can also apply Lemma E.26 to Σ; Δ; $\Gamma \vdash \boxed{v} : \tau_1^{\text{SI}} \Rightarrow \Gamma_1$ (from premise of T-Let) and Δ; $\Gamma_1 \vdash \tau_1^{\text{SI}} \lesssim \tau_a^{\text{SI}} \Rightarrow \Gamma_1'$ (from premise of T-Let) gives us Σ; Δ; $\Gamma' \vdash \boxed{e_1} : \tau_a^{\text{SI}} \Rightarrow \Gamma'$ Then, apply Lemma E.25 to $\Sigma \vdash \sigma : \Gamma_1'$ and Σ; Δ; $\Gamma' \vdash \boxed{e_1} : \tau_a^{\text{SI}} \Rightarrow \Gamma'$ gives us $\Sigma \vdash \sigma, x \mapsto v : \Gamma_1', x : \tau_a^{\text{SI}}$. We can then apply Lemma E.28 to conclude $\Sigma \vdash \sigma,\ x \mapsto v : \text{gc-loans}(\Gamma_1',\ x\ :\ \tau_a^{\text{SI}})$.
Σ; $\bullet$; $\text{gc-loans}(\Gamma_1',\ x\ :\ \tau_a^{\text{SI}}) \vdash \boxed{\texttt{shift}\ e} : \tau_2^{\text{SI}} \Rightarrow \Gamma_2$	Immediate by applying T-Shift to the derivation Σ; $\bullet$; $\text{gc-loans}(\Gamma_1',\ x\ :\ \tau_a^{\text{SI}}) \vdash \boxed{e} : \tau_2^{\text{SI}} \Rightarrow \Gamma_2,\ x\ :\ \tau^{\text{SD}}$ (from premise of T-Let).
$\bullet$; $\Gamma_2 \vdash \tau_2^{\text{SI}} \lesssim \tau_2^{\text{SI}} \Rightarrow \Gamma_2$	Immediate by S-Refl.
$\exists \Gamma_o . \Gamma_2 \Cup \Gamma_o = \Gamma_2$	$\Gamma_o = \Gamma_2$

Case T-LetProv:

From premise:

$$\textsc{T-LetProv}\ \frac{\Sigma;\ \Delta;\ \Gamma,\ r \mapsto \{\} \vdash \boxed{e} : \tau^{\mathrm{SI}} \Rightarrow \Gamma',\ r \mapsto \{\bar{\ell}\}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\texttt{letprov <}r\texttt{> \{ }e\texttt{ \}}} : \tau^{\mathrm{SI}} \Rightarrow \Gamma'}$$

Since $e = \texttt{letprov <}r\texttt{> \{ }e\texttt{ \}}$, by inspection of the reduction rules, we know that e steps with the following rule:

$$\textsc{E-LetProv}\ \frac{}{\Sigma \vdash (\sigma;\ \boxed{\texttt{letprov <}r\texttt{> \{ }v\texttt{ \}}}) \rightarrow (\sigma;\ \boxed{v})}$$

We then pick Γ_i to be $\boxed{\Gamma'}$, and need to show:

$\Sigma \vdash \sigma : \Gamma'$	Applying Lemma E.9 to the typing derivation (from T-LetProv) for e (which we know is a value from E-LetProv) gives us $\Sigma;\ \bullet \vdash \Gamma \lesssim \Gamma'$. Then, applying Lemma E.23 with this and $\Sigma \vdash \sigma : \Gamma$ (from premise) gives us $\Sigma \vdash \sigma : \Gamma'$.
$\Sigma;\ \bullet;\ \Gamma' \vdash \boxed{v} : \tau^{\mathrm{SI}} \Rightarrow \Gamma'$	We know from E-LetProv that e is a value v. Thus, we can apply Lemma E.21 to $\Sigma;\ \bullet;\ \Gamma \vdash \boxed{v} : \tau^{\mathrm{SI}} \Rightarrow \Gamma',\ r \mapsto \{\ \bar{\ell}\ \}$ to get $\Sigma;\ \bullet;\ \Gamma',\ r \mapsto \{\ \bar{\ell}\ \} \vdash \boxed{v} : \tau^{\mathrm{SI}} \Rightarrow \Gamma',\ r \mapsto \{\ \bar{\ell}\ \}$. We now wish to show that $\Sigma;\ \bullet;\ \Gamma' \vdash \boxed{v} : \tau^{\mathrm{SI}} \Rightarrow \Gamma'$. By inspecting the grammar of values and their typing rules, we know that the only values who depend on the context are pointers and closure values. But by inversion on $\Sigma;\ \bullet;\ \Gamma \vdash \boxed{\texttt{letprov <}r\texttt{> \{ }v\texttt{ \}}} : \tau^{\mathrm{SI}} \Rightarrow \Gamma'$, we know that $\Sigma;\ \bullet;\ \Gamma' \vdash \tau^{\mathrm{SI}}$. Since the type is valid without the frame, we know that the values cannot depend on that frame. Thus, we can conclude $\Sigma;\ \bullet;\ \Gamma' \vdash \boxed{v} : \tau^{\mathrm{SI}} \Rightarrow \Gamma'$.
$\bullet;\ \Gamma' \vdash \tau^{\mathrm{SI}} \lesssim \tau^{\mathrm{SI}} \Rightarrow \Gamma'$	Immediate by S-Refl.
$\exists \Gamma_o . \Gamma' \Cup \Gamma_o = \Gamma'$	$\Gamma_o = \Gamma'$

Case T-While:

From premise:

$$\textsc{T-While}\ \frac{\begin{array}{ll}\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e_1} : \texttt{bool} \Rightarrow \Gamma_1 & \Sigma;\ \Delta;\ \Gamma_1 \vdash \boxed{e_2} : \texttt{unit} \Rightarrow \Gamma_2 \\ \Sigma;\ \Delta;\ \Gamma_2 \vdash \boxed{e_1} : \texttt{bool} \Rightarrow \Gamma_2 & \Sigma;\ \Delta;\ \Gamma_2 \vdash \boxed{e_2} : \texttt{unit} \Rightarrow \Gamma_2\end{array}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\texttt{while } e_1 \texttt{ \{ } e_2 \texttt{ \}}} : \texttt{unit} \Rightarrow \Gamma_2}$$

Since $e = \texttt{while } e_1 \texttt{ \{ } e_2 \texttt{ \}}$, by inspection of the reduction rules, we know that e steps with the following rule:

$$\textsc{E-While}\ \frac{}{\Sigma \vdash (\sigma;\ \boxed{\texttt{while } e_1 \texttt{ \{ } e_2 \texttt{ \}}}) \rightarrow (\sigma;\ \boxed{\texttt{if } e_1 \texttt{ \{ } e_2\texttt{; while } e_1 \texttt{ \{ } e_2 \texttt{ \} \} else \{ ( ) \}}})}$$

We then pick Γ_i to be $\boxed{\Gamma}$, and need to show:

$\Sigma \vdash \sigma : \Gamma$	Immediate from our premise.
$\Sigma;\ \bullet;\ \Gamma \vdash \boxed{e'} : \texttt{unit} \Rightarrow \Gamma_2$	We would like to build a derivation to show that the expression $\texttt{if}\ e_1\ \{\ e_2;\ \texttt{while}\ e_1\ \{\ e_2\ \}\ \}\ \texttt{else}\ \{\ ()\ \}$ is well-typed. We thus start by applying T-Branch. This requires us to show three things. First, $\Sigma;\ \bullet;\ \Gamma \vdash \boxed{e_1} : \texttt{bool} \Rightarrow \Gamma_1$ which we have from the premise of T-While. Second, $\Sigma;\ \bullet;\ \Gamma_1 \vdash \boxed{e_2;\ \texttt{while}\ e_1\ \{\ e_2\ \}} : \texttt{unit} \Rightarrow \Gamma_2$. We build this by applying T-Seq to $\Sigma;\ \bullet;\ \Gamma_1 \vdash \boxed{e_2} : \texttt{unit} \Rightarrow \Gamma_2$ and $\Sigma;\ \bullet;\ \Gamma_2 \vdash \boxed{\texttt{while}\ e_1\ \{\ e_2\ \}} : \Gamma_2 \Rightarrow$. The former is directly in the premise of T-While and the latter can be built by applying T-While to $\Sigma;\ \Delta;\ \Gamma_2 \vdash \boxed{e_1} : \texttt{bool} \Rightarrow \Gamma_2$ and $\Sigma;\ \Delta;\ \Gamma_2 \vdash \boxed{e_2} : \texttt{unit} \Rightarrow \Gamma_2$, both from the premise of our original T-While. Finally, we need to show $\Sigma;\ \Delta;\ \Gamma_2 \vdash \boxed{()} : \texttt{unit} \Rightarrow \Gamma_2$, which is immediate from T-Unit.
$\bullet;\ \Gamma_2 \vdash \texttt{unit} \lesssim \texttt{unit} \Rightarrow \Gamma_2$	Immediate by S-Refl.
$\exists \Gamma_o . \Gamma_2 \Cup \Gamma_o = \Gamma_2$	$\Gamma_o = \Gamma_2$

Case T-ForArray:

From premise:

$$\textsc{T-ForArray}\ \frac{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e_1} : [\tau^{\text{SI}};\ n] \Rightarrow \Gamma_1 \qquad \Sigma;\ \Delta;\ \Gamma_1,\ x\ :\ \tau^{\text{SI}} \vdash \boxed{e_2} : \texttt{unit} \Rightarrow \Gamma_1,\ x\ :\ \tau^{\text{SD}}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\texttt{for}\ x\ \texttt{in}\ e_1\ \{\ e_2\ \}} : \texttt{unit} \Rightarrow \Gamma_1}$$

Since $e = \texttt{for}\ x\ \texttt{in}\ e_1\ \{\ e_2\ \}$, by inspection of the reduction rules, we know that e steps with the following rule:

$$\textsc{E-ForArray}\ \frac{}{\Sigma \vdash (\sigma;\ \boxed{\texttt{for}\ x\ \texttt{in}\ [v_0,\ \ldots,\ v_n]\ \{\ e\ \}}) \rightarrow (\sigma,\ x \mapsto v_0;\ \boxed{\texttt{shift}\ e;\ \texttt{for}\ x\ \texttt{in}\ [v_1,\ \ldots,\ v_n]\ \{\ e\ \}})}$$

$$\textsc{E-ForEmptyArray}\ \frac{}{\Sigma \vdash (\sigma;\ \boxed{\texttt{for}\ x\ \texttt{in}\ []\ \{\ e\ \}}) \rightarrow (\sigma;\ \boxed{()})}$$

Then, for each possible rule, we'll pick Γ_i separately. The cases proceed as follows:
For E-ForArray, we pick Γ_i to be $\boxed{\Gamma_1,\ x\ :\ \tau^{\text{SI}}}$, and need to show:

$\Sigma \vdash \sigma,\ x \mapsto v_0 : \Gamma_1,\ x\ :\ \tau^{\mathrm{SI}}$	Applying Lemma E.9 to the typing derivation (from T-ForArray) for e_1 (which we know is a value from E-ForArray) gives us $\Sigma;\ \bullet \vdash \Gamma \lesssim \Gamma_1$. Then, we can apply Lemma E.23 to get $\Sigma \vdash \sigma : \Gamma_1$. Applying Lemma E.21 to the derivation $\Sigma;\ \bullet;\ \Gamma \vdash \boxed{[v_0,\ \ldots,\ v_n]} : [\tau^{\mathrm{SI}};\ n] \Rightarrow \Gamma_1$ gives us $\Sigma;\ \bullet;\ \Gamma_1 \vdash \boxed{[v_0,\ \ldots,\ v_n]} : [\tau^{\mathrm{SI}};\ n] \Rightarrow \Gamma_1$. Then, using inversion (of T-Array), we get $\Sigma;\ \bullet;\ \Gamma_1 \vdash \boxed{v_0} : \tau^{\mathrm{SI}} \Rightarrow \Gamma_1$. Finally, applying Lemma E.25 to $\Sigma \vdash \sigma : \Gamma_1$ and $\Sigma;\ \Delta;\ \Gamma_1 \vdash \boxed{v_0} : \tau^{\mathrm{SI}} \Rightarrow \Gamma_1$ gives us $\Sigma \vdash \sigma,\ x \mapsto v_0 : \Gamma_1,\ x\ :\ \tau^{\mathrm{SI}}$.
$\Sigma;\ \bullet;\ \Gamma_1,\ x\ :\ \tau^{\mathrm{SI}} \vdash \boxed{e'} : \mathtt{unit} \Rightarrow \Gamma_1$	We need to build a derivation for the expression $\mathtt{shift}\ e;\ \mathtt{for}\ x\ \mathtt{in}\ [v_1,\ \ldots,\ v_n]\ \{\ e\ \}$. The bottom of this derivation will be T-Seq which requires us to show that $\Sigma;\ \bullet;\ \Gamma_1,\ x\ :\ \tau^{\mathrm{SI}} \vdash \boxed{\mathtt{shift}\ e} : \mathtt{unit} \Rightarrow \Gamma_1$ and that $\Sigma;\ \bullet;\ \Gamma_1 \vdash \boxed{\mathtt{for}\ x\ \mathtt{in}\ [v_1,\ \ldots,\ v_n]\ \{\ e\ \}} : \mathtt{unit} \Rightarrow \Gamma_1$. To show $\Sigma;\ \bullet;\ \Gamma_1,\ x\ :\ \tau^{\mathrm{SI}} \vdash \boxed{\mathtt{shift}\ e} : \mathtt{unit} \Rightarrow \Gamma_1$, we apply T-Shift to $\Sigma;\ \bullet;\ \Gamma_1,\ x\ :\ \tau^{\mathrm{SI}} \vdash \boxed{e} : \mathtt{unit} \Rightarrow \Gamma_1,\ x\ :\ \tau^{\mathrm{SD}}$ (from the premise of T-ForArray). To show $\Sigma;\ \bullet;\ \Gamma_1 \vdash \boxed{\mathtt{for}\ x\ \mathtt{in}\ [v_1,\ \ldots,\ v_n]\ \{\ e\ \}} : \mathtt{unit} \Rightarrow \Gamma_1$, we apply Lemma E.21 to the derivation $\Sigma;\ \bullet;\ \Gamma \vdash \boxed{[v_0,\ \ldots,\ v_n]} : [\tau^{\mathrm{SI}};\ n] \Rightarrow \Gamma_1$ to get $\Sigma;\ \bullet;\ \Gamma_1 \vdash \boxed{[v_0,\ \ldots,\ v_n]} : [\tau^{\mathrm{SI}};\ n] \Rightarrow \Gamma_1$. Then, we rewrite the derivation (inverting and then reapply T-Array) to exclude v_0 giving us $\Sigma;\ \bullet;\ \Gamma_1 \vdash \boxed{[v_1,\ \ldots,\ v_n]} : [\tau^{\mathrm{SI}};\ n-1] \Rightarrow \Gamma_1$. Finally, we apply T-ForArray using this combined with $\Sigma;\ \bullet;\ \Gamma_1 \vdash \boxed{e} : \mathtt{unit} \Rightarrow \Gamma_1$ (from the premise of T-ForArray).
$\bullet;\ \Gamma_1 \vdash \mathtt{unit} \lesssim \mathtt{unit} \Rightarrow \Gamma_1$	Immediate by S-Refl.
$\exists \Gamma_o . \Gamma_1\ \mathbin{\mathbb{U}}\ \Gamma_o = \Gamma_1$	$\Gamma_o = \Gamma_1$

For E-ForEmptyArray, we pick Γ_i to be $\boxed{\Gamma}$, and need to show:

$\Sigma \vdash \sigma : \Gamma$	Immediate from our premise.
$\Sigma;\ \bullet;\ \Gamma \vdash \boxed{()} : \mathtt{unit} \Rightarrow \Gamma$	Immediate by T-Unit.
$\bullet;\ \Gamma \vdash \mathtt{unit} \lesssim \mathtt{unit} \Rightarrow \Gamma$	Immediate by S-Refl.
$\exists \Gamma_o . \Gamma\ \mathbin{\mathbb{U}}\ \Gamma_o = \Gamma$	$\Gamma_o = \Gamma$

Case T-ForSlice:

From premise:

$$\textsc{T-ForSlice}\quad \frac{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e_1} : \&\rho\ \omega\ [\tau^{\text{SI}}] \Rightarrow \Gamma_1 \qquad \Sigma;\ \Delta;\ \Gamma_1,\ x\ :\ \&\rho\ \omega\ \tau^{\text{SI}} \vdash \boxed{e_2} : \texttt{unit} \Rightarrow \Gamma_1,\ x\ :\ \tau_1^{\text{SX}}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\texttt{for}\ x\ \texttt{in}\ e_1\ \{\ e_2\ \}} : \texttt{unit} \Rightarrow \Gamma_2}$$

Since $e = \texttt{for}\ x\ \texttt{in}\ e_1\ \{\ e_2\ \}$, by inspection of the reduction rules, we know that e steps with the following rule:

$$\textsc{E-ForSlice}\quad \frac{\sigma \vdash \mathcal{R} \Downarrow \_ \mapsto [v_1, \ldots, v_i, \ldots, v_j, \ldots, v_n] \qquad i < j \qquad i' = i + 1}{\Sigma \vdash (\sigma;\ \boxed{\texttt{for}\ x\ \texttt{in}\ \texttt{ptr}\ \mathcal{R}[i..j]\ \{\ e\ \}}) \rightarrow (\sigma,\ x \mapsto \texttt{ptr}\ \mathcal{R}[i];\ \boxed{\texttt{shift}\ e;\ \texttt{for}\ x\ \texttt{in}\ \texttt{ptr}\ \mathcal{R}[i'..j]\ \{\ e\ \}})}$$

$$\textsc{E-ForEmptySlice}\quad \frac{}{\Sigma \vdash (\sigma;\ \boxed{\texttt{for}\ x\ \texttt{in}\ \texttt{ptr}\ \pi[n..n]\ \{\ e\ \}}) \rightarrow (\sigma;\ \boxed{(\,)})}$$

Then, for each possible rule, we'll pick Γ_i separately. The cases proceed as follows:
For E-ForSlice, we pick Γ_i to be $\boxed{\Gamma_1,\ x\ :\ \&r\ \omega\ \tau^{\text{SI}}}$, and need to show:

$\Sigma \vdash \sigma,\ x \mapsto \texttt{ptr}\ \mathcal{R}[n_1] : \Gamma_1,\ x\ :\ \&r\ \omega\ \tau^{\text{SI}}$	Applying Lemma E.9 to the typing derivation (from T-ForSlice) for e_1 (which we know is a value from E-ForSlice) gives us $\Sigma;\ \bullet \vdash \Gamma \lesssim \Gamma_1$. Then, we can apply Lemma E.23 to get $\Sigma \vdash \sigma : \Gamma_1$. Applying Lemma E.21 to the derivation $\Sigma;\ \bullet;\ \Gamma \vdash \boxed{\texttt{ptr}\ \mathcal{R}[i..j]} : \&r\ \omega\ [\tau^{\text{SI}}] \Rightarrow \Gamma_1$ gives us $\Sigma;\ \bullet;\ \Gamma_1 \vdash \boxed{\texttt{ptr}\ \mathcal{R}[i..j]} : \&r\ \omega\ [\tau^{\text{SI}}] \Rightarrow \Gamma_1$. Then, using inversion (on T-Pointer), we get $\Sigma;\ \Gamma_1 \vdash \mathcal{R}[i..j] : \tau^{\text{XI}}$ (where $\tau^{\text{XI}} = [\tau^{\text{SI}};\ n]$ or $[\tau^{\text{SI}}]$) and ${}^{\omega}\pi\ \in \Gamma_1(r)$ (where $\mathcal{R} = \mathcal{R}^{\square}[\pi]$). We can invert WF-RefSliceArray or WF-RefSliceSlice (based on τ^{XI}) for $\Sigma;\ \Gamma_1 \vdash \mathcal{R}[i..j] : \tau^{\text{XI}}$ and then apply WF-RefIndexArray or WF-RefIndexSlice appropriately to get $\Sigma;\ \Gamma_1 \vdash \mathcal{R}[i] : \tau^{\text{XI}}$. We can then use T-Pointer to get $\Sigma;\ \bullet;\ \Gamma_1 \vdash \boxed{\texttt{ptr}\ \mathcal{R}[n_1]} : \&r\ \omega\ \tau^{\text{SI}} \Rightarrow \Gamma_1$. Finally, applying Lemma E.25 to $\Sigma \vdash \sigma : \Gamma_1$ and $\Sigma;\ \Delta;\ \Gamma_1 \vdash \boxed{\texttt{ptr}\ \mathcal{R}[n_1]} : \&r\ \omega\ \tau^{\text{SI}} \Rightarrow \Gamma_1$ gives us $\Sigma \vdash \sigma,\ x \mapsto \texttt{ptr}\ \mathcal{R}[n_1] : \Gamma_1,\ x\ :\ \&r\ \omega\ \tau^{\text{SI}}$.
$\Sigma;\ \bullet;\ \Gamma_1,\ x\ :\ \&r\ \omega\ \tau^{\text{SI}} \vdash \boxed{e'} : \texttt{unit} \Rightarrow \Gamma_1$	We need to build a derivation for the expression $\texttt{shift}\ e;\ \texttt{for}\ x\ \texttt{in}\ \texttt{ptr}\ \mathcal{R}[n_1'..n_2]\ \{\ e\ \}$. The bottom of this derivation will be T-Seq which requires us to show that $\Sigma;\ \bullet;\ \Gamma_1,\ x\ :\ \tau^{\text{SI}} \vdash \boxed{\texttt{shift}\ e} : \texttt{unit} \Rightarrow \Gamma_1$ and that $\Sigma;\ \bullet;\ \Gamma_1 \vdash \boxed{\texttt{for}\ x\ \texttt{in}\ \texttt{ptr}\ \mathcal{R}[n_1'..n_2]\ \{\ e\ \}} : \texttt{unit} \Rightarrow \Gamma_1$. To show $\Sigma;\ \bullet;\ \Gamma_1,\ x\ :\ \tau^{\text{SI}} \vdash \boxed{\texttt{shift}\ e} : \texttt{unit} \Rightarrow \Gamma_1$, we apply T-Shift to $\Sigma;\ \bullet;\ \Gamma_1, x : \tau^{\text{SI}} \vdash \boxed{e} : \texttt{unit} \Rightarrow \Gamma_1, x : \tau^{\text{SD}}$ (from the premise of T-ForSlice). To show $\Sigma;\ \bullet;\ \Gamma_1 \vdash \boxed{\texttt{for}\ x\ \texttt{in}\ \texttt{ptr}\ \mathcal{R}[n_1'..n_2]\ \{\ e\ \}} : \texttt{unit} \Rightarrow \Gamma_1$, we apply Lemma E.21 to the derivation $\Sigma;\ \bullet;\ \Gamma \vdash \boxed{\texttt{ptr}\ \mathcal{R}[n_1..n_2]} : \&r\ \omega\ [\tau^{\text{SI}}] \Rightarrow \Gamma_1$ to get $\Sigma;\ \bullet;\ \Gamma_1 \vdash \boxed{\texttt{ptr}\ \mathcal{R}[n_1..n_2]} : \&r\ \omega\ [\tau^{\text{SI}}] \Rightarrow \Gamma_1$. Then, we rewrite the derivation (inverting and then reapply T-Pointer) to increment n_1 to n_1' giving us $\Sigma;\ \bullet;\ \Gamma_1 \vdash \boxed{\texttt{ptr}\ \mathcal{R}[n_1'..n_2]} : \&r\ \omega\ [\tau^{\text{SI}}] \Rightarrow \Gamma_1$. Finally, we apply T-ForSlice using this combined with $\Sigma;\ \bullet;\ \Gamma_1 \vdash \boxed{e} : \texttt{unit} \Rightarrow \Gamma_1$ (from the premise of T-ForSlice).
$\bullet;\ \Gamma_1 \vdash \texttt{unit} \lesssim \texttt{unit} \Rightarrow \Gamma_1$	Immediate by S-Refl.
$\exists \Gamma_o . \Gamma_1\ \Cup\ \Gamma_o = \Gamma_1$	$\Gamma_o = \Gamma_1$

For E-ForEmptySlice, we pick Γ_i to be $\boxed{\Gamma}$, and need to show:

$\Sigma \vdash \sigma : \Gamma$	Immediate from our premise.
$\Sigma;\ \bullet;\ \Gamma \vdash \boxed{()} : \texttt{unit} \Rightarrow \Gamma$	Immediate by T-Unit.
$\bullet;\ \Gamma \vdash \texttt{unit} \lesssim \texttt{unit} \Rightarrow \Gamma$	Immediate by S-Refl.
$\exists \Gamma_o . \Gamma\ \Cup\ \Gamma_o = \Gamma$	$\Gamma_o = \Gamma$

Case T-App:

From premise:

T-App

$$\frac{\overline{\Sigma;\ \Delta;\ \Gamma \vdash \Phi} \qquad \overline{\Delta;\ \Gamma \vdash \rho} \qquad \overline{\Sigma;\ \Delta;\ \Gamma \vdash \tau^{\mathrm{SI}}} \qquad \Sigma;\ \Delta;\ \Gamma \vdash \boxed{\hat{e}_f} : \forall <\overline{\varphi},\ \overline{\varrho},\ \overline{\alpha}> (\tau_1^{\mathrm{SI}},\ \ldots,\ \tau_n^{\mathrm{SI}}) \xrightarrow{\Phi_c} \tau_f^{\mathrm{SI}}\ \mathtt{where}\ \overline{\varrho_1 : \varrho_2} \Rightarrow \Gamma_0 \qquad \forall i \in \{1 \ldots n\}.\ \Sigma;\ \Delta;\ \Gamma_{i-1} \vdash \boxed{\hat{e}_i} : \tau_i^{\mathrm{SI}}\ \overline{[\Phi/\varphi]}\ \overline{[\rho/\varrho]}\ \overline{[\tau^{\mathrm{SI}}/\alpha]} \Rightarrow \Gamma_i \qquad \Delta;\ \Gamma_n \vdash \overline{\varrho_2\ \overline{[\rho/\varrho]} :> \varrho_1\ \overline{[\rho/\varrho]}} \Rightarrow \Gamma_b}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\hat{e}_f ::<\overline{\Phi},\ \overline{\rho},\ \overline{\tau^{\mathrm{SI}}}> (\hat{e}_1,\ \ldots,\ \hat{e}_n)} : \tau_f^{\mathrm{SI}}\ \overline{[\Phi/\varphi]}\ \overline{[\rho/\varrho]}\ \overline{[\tau^{\mathrm{SI}}/\alpha]} \Rightarrow \Gamma_b}$$

Since $e = e_f ::<\overline{\rho'},\ \overline{\tau^{\mathrm{SI}}}> (e_1,\ \ldots,\ e_n)$, by inspection of the reduction rules, we know that e steps with the following rule:

E-AppClosure

$$\frac{v_f = \langle \varsigma_c,\ |x_1 : \tau_1^{\mathrm{S}},\ \ldots,\ x_n : \tau_n^{\mathrm{S}}|\ \rightarrow\ \tau_r^{\mathrm{S}}\ \{\ e\ \}\ \rangle}{\Sigma \vdash (\sigma;\ \boxed{v_f(v_1,\ \ldots,\ v_n)}) \rightarrow (\sigma \natural \varsigma_c,\ x_1 \mapsto v_1,\ \ldots,\ x_n \mapsto v_n;\ \boxed{\mathtt{framed}\ e})}$$

E-AppFunction

$$\frac{\Sigma(f) = \mathtt{fn}\ f <\overline{\varphi},\ \overline{\varrho},\ \overline{\alpha}> (x_1 : \tau_1^{\mathrm{S}},\ \ldots,\ x_n : \tau_n^{\mathrm{S}})\ \rightarrow\ \tau_r^{\mathrm{S}}\ \mathtt{where}\ \overline{\varrho : \varrho'}\ \{\ e\ \}}{\Sigma \vdash (\sigma;\ \boxed{f ::<\overline{\Phi},\ \overline{r'},\ \overline{\tau^{\mathrm{S}}}> (v_1,\ \ldots,\ v_n)}) \rightarrow (\sigma \natural x_1 \mapsto v_1,\ \ldots,\ x_n \mapsto v_n;\ \boxed{\mathtt{framed}\ e[\overline{\Phi}/\overline{\varphi}][\overline{r'}/\overline{\varrho}][\overline{\tau^{\mathrm{S}}}/\overline{\alpha}]})}$$

Then, for each possible rule, we'll pick Γ_i separately. The cases proceed as follows:
For E-AppClosure, we pick Γ_i to be $\boxed{\Gamma_b \natural \mathcal{F}_c,\ x_1 : \tau_1^{\mathrm{SI}},\ \ldots,\ x_n : \tau_n^{\mathrm{SI}}}$, and need to show:

$\Sigma \vdash \sigma' : \Gamma_i$	Applying Lemma E.9 to the derivation for v_f gives us Σ; $\bullet \vdash \Gamma \lesssim \Gamma_0$. Then, applying Lemma E.9 to the derivations for every v_i gives us $\forall i \in \{ 1 \ \ldots \ n \}$. Σ; $\bullet \vdash \Gamma_{i-1} \lesssim \Gamma_i$. By transitivity, we then have Σ; $\bullet \vdash \Gamma \lesssim \Gamma_n$. Since the function being applied is a closure, we know syntactically that it does not have where bounds, and thus $\Gamma_b = \Gamma_n$. Thus, we can rewrite this to be Σ; $\bullet \vdash \Gamma \lesssim \Gamma_b$. We can then apply Lemma E.23 with $\Sigma \vdash \sigma : \Gamma$ to get $\Sigma \vdash \sigma : \Gamma_b$. Then, inversion of T-ClosureValue for the typing derivation for v_f gives us Σ; $\Gamma \vdash \varsigma_c : \mathcal{F}_c$. We can then invert WF-Frame here to get $\text{dom}(\varsigma) = \text{dom}(\mathcal{F}_c)|_x$ $\forall x \in \text{dom}(\varsigma)$. Σ; $\bullet$; $\Gamma \uplus \mathcal{F}_c \vdash \boxed{\varsigma(x)} : \mathcal{F}_c(x) \Rightarrow \Gamma \uplus \mathcal{F}_c$ which we can then use with $\Sigma \vdash \sigma : \Gamma_b$ in WF-StackFrame to get $\Sigma \vdash \sigma \uplus \varsigma_c : \Gamma_b \uplus \mathcal{F}_c$. Finally, we repeatedly apply Lemma E.25 to the derivations for the arguments $(v_1 \ \ldots \ v_n)$ to get $\Sigma \vdash \sigma' : \Gamma_i$.
Σ; $\bullet$; $\Gamma_i \vdash \boxed{e'} : \tau_f^{\text{SI}} \Rightarrow \Gamma_b$	Applying Lemma E.9 to the derivation for v_f gives us Σ; $\bullet \vdash \Gamma \lesssim \Gamma_0$. Then, applying Lemma E.9 to the derivations for every v_i gives us $\forall i \in \{ 1 \ \ldots \ n \}$. Σ; $\bullet \vdash \Gamma_{i-1} \lesssim \Gamma_i$. By transitivity, we then have Σ; $\bullet \vdash \Gamma \lesssim \Gamma_n$. Since the function being applied is a closure, we know syntactically that it does not have where bounds, and thus $\Gamma_b = \Gamma_n$. Thus, we can rewrite this to be Σ; $\bullet \vdash \Gamma \lesssim \Gamma_b$. Adding the same frame $\mathcal{F}_c$ to both sides gives us Σ; $\bullet \vdash \Gamma \uplus \mathcal{F}_c \lesssim \Gamma_b \uplus \mathcal{F}_c$. Further, adding identical argument entries to both sides gives us Σ; $\bullet \vdash \Gamma \uplus \mathcal{F}_c, x_1 : \tau_1^{\text{SI}}, \ldots, x_n : \tau_n^{\text{SI}} \lesssim \Gamma_b \uplus \mathcal{F}_c, x_1 : \tau_1^{\text{SI}}, \ldots, x_n : \tau_n^{\text{SI}}$. Inversion on T-ClosureValue for the typing derivation of $\underline{v_f}$ gives us free-vars$(e) \setminus \overline{x} = \overline{x_f} = \text{dom}(\mathcal{F}_c)|_x$, $\overline{r} = \overline{\text{free-provs}(\Gamma(x_f))}$, free-provs$(e) = \text{dom}(\mathcal{F}_c)|_r$, and Σ; Δ; $\Gamma \uplus \mathcal{F}_c, x_1 : \tau_1^{\text{SI}}, \ldots, x_n : \tau_n^{\text{SI}} \vdash \boxed{e} : \tau_r^{\text{SI}} \Rightarrow \Gamma' \uplus \mathcal{F}$. We can then apply Lemma E.18 with all of these facts to get Σ; $\bullet$; $\Gamma_b \uplus \mathcal{F}_c, x_1 : \tau_1^{\text{SI}}, \ldots, x_n : \tau_n^{\text{SI}} \vdash \boxed{e} : \tau_f^{\text{SI}} \Rightarrow \Gamma_b \uplus \mathcal{F}$. We can then apply T-Framed to get Σ; $\bullet$; $\Gamma_b \uplus \mathcal{F}_c, x_1 : \tau_1^{\text{SI}}, \ldots, x_n : \tau_n^{\text{SI}} \vdash \boxed{e} : \tau_f^{\text{SI}} \Rightarrow \Gamma_b$.
$\bullet$; $\Gamma_b \vdash \tau_f^{\text{SI}} \lesssim \tau_f^{\text{SI}} \Rightarrow \Gamma_b$	Immediate by S-Refl.
$\exists \Gamma_o . \Gamma_b \ \Cup \ \Gamma_o = \Gamma_b$	$\Gamma_o = \Gamma_b$

For E-AppFunction, we pick Γ_i to be $\boxed{\Gamma_b \uplus x_1 : \tau_1^{\text{SI}}, \ldots, x_n : \tau_n^{\text{SI}}}$, and need to show:

$\Sigma \vdash \sigma' : \Gamma_i$	Applying Lemma E.9 to the derivation for v_f gives us $\Sigma; \bullet \vdash \Gamma \lesssim \Gamma_0$. Then, applying Lemma E.9 to the derivations for every v_i gives us $\forall i \in \{1 \ldots n\}.\ \Sigma; \bullet \vdash \Gamma_{i-1} \lesssim \Gamma_i$. Inversion on $\bullet;\ \Gamma_n \vdash \varrho_2\ \overline{[\rho/\varrho]} :> \varrho_1\ \overline{[\rho/\varrho]} \Rightarrow \Gamma_b$ gives us a sequence of outlives relations with intermediate contexts. Applying Lemma E.16 to each of them and then combining the result by transitivity gives us $\Sigma; \bullet \vdash \Gamma_n \lesssim \Gamma_b$. Combining both by transitivity, we have $\Sigma; \bullet \vdash \Gamma \lesssim \Gamma_b$. We can then apply Lemma E.23 with $\Sigma \vdash \sigma : \Gamma$ to get $\Sigma \vdash \sigma : \Gamma_b$. We can apply WF-StackFrame to get $\Sigma \vdash \sigma \natural \bullet : \Gamma_b \natural \bullet$. Then, we repeatedly apply Lemma E.25 to the derivations for the arguments $(v_1 \ \ldots \ v_n)$ to get $\Sigma \vdash \sigma \natural x_1 \mapsto v_1, \ldots, x_n \mapsto v_n : \Gamma \natural x_1 : \tau_1^{\text{SI}}, \ldots, x_n : \tau_n^{\text{SI}}$.
$\Sigma; \bullet; \Gamma_i \vdash \boxed{e'} : \tau_f^{\text{SI}}[\overline{\Phi}/\overline{\varphi}][\overline{r}/\overline{\varrho}][\overline{\tau^{\text{SI}}}/\overline{\alpha}] \Rightarrow \Gamma_b$	Applying Lemma E.29 with $\vdash\ \Sigma; \bullet; \Gamma_b$ and $\Sigma(f) = \texttt{fn}\, f\texttt{<}\overline{\varphi}, \overline{\varrho}, \overline{\alpha}\texttt{>}(x_1{:}\tau_1^{\text{SI}}, \ldots, x_n{:}\tau_n^{\text{SI}}) \rightarrow \tau_r^{\text{SI}}\, \texttt{where}\, \overline{\varrho_1 : \varrho_2}\, \{e\}$ (from the premise of on E-AppFunction) gives us $\Sigma;\ \overline{\varphi : \texttt{FRM}}, \overline{\varrho : \texttt{PRV}}, \overline{\varrho_1 :> \varrho_2}, \overline{\alpha : \star}; \Gamma_b \natural x_1 : \tau_1^{\text{SI}}, \ldots, x_n : \tau_n^{\text{SI}} \vdash \boxed{\texttt{framed}\, e} : \tau_f^{\text{SI}} \Rightarrow \Gamma_b$. We then repeatedly apply Lemma E.10 for all of our type, provenance, and environment variables to get $\Sigma; \bullet; \Gamma_b \natural x_1 : \tau_{s1}^{\text{SI}}, \ldots, x_n : \tau_{sn}^{\text{SI}} \vdash \boxed{\texttt{framed}\, e} : \tau_f^{\text{SI}}[\overline{\Phi}/\overline{\varphi}][\overline{r}/\overline{\varrho}][\overline{\tau^{\text{SI}}}/\overline{\alpha}] \Rightarrow \Gamma_b$ where each $\tau_{si}^{\text{SI}} = \tau_i^{\text{SI}}\ \overline{[\Phi/\varphi]}\ \overline{[\rho/\varrho]}\ \overline{[\tau^{\text{SI}}/\alpha]}$.
$\bullet; \Gamma_b \vdash \tau' \lesssim \tau' \Rightarrow \Gamma_b$	Immediate by S-Refl.
$\exists \Gamma_o . \Gamma_b \Cup \Gamma_o = \Gamma_b$	$\Gamma_o = \Gamma_b$

Case T-Function:

From premise:

$$\text{T-Function}\quad \frac{\Sigma(f) = \texttt{fn}\, f\texttt{<}\overline{\varphi}, \overline{\varrho}, \overline{\alpha}\texttt{>}(x_1 : \tau_1^{\text{SI}}, \ldots, x_n : \tau_n^{\text{SI}}) \ \rightarrow\ \tau_r^{\text{SI}}\ \texttt{where}\ \overline{\varrho_1 : \varrho_2}\ \{\ e\ \}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{f} : \forall\texttt{<}\overline{\varphi}, \overline{\varrho}, \overline{\alpha}\texttt{>}(\tau_1^{\text{SI}}, \ldots, \tau_n^{\text{SI}}) \ \rightarrow\ \tau_r^{\text{SI}}\ \texttt{where}\ \overline{\varrho_1 : \varrho_2} \Rightarrow \Gamma}$$

Since $e = \forall\texttt{<}\overline{\rho}, \overline{\alpha}\texttt{>}(\tau_1^{\text{SI}}, \ldots, \tau_n^{\text{SI}}) \ \rightarrow\ \tau_r^{\text{SI}}$, by inspection of the reduction rules, we know that e steps with the following rule:

$$\text{E-Function}\quad \frac{\Sigma(f) = \texttt{fn}\, f\texttt{<}\overline{\varphi}, \overline{\varrho}, \overline{\alpha}\texttt{>}(x_1 : \tau_1^{\text{S}}, \ldots, x_n : \tau_n^{\text{S}}) \ \rightarrow\ \tau_r^{\text{S}}\ \texttt{where}\ \overline{\varrho : \varrho'}\ \{\ e\ \}}{\Sigma \vdash (\sigma;\ \boxed{f}) \rightarrow (\sigma;\ \boxed{\langle \bullet,\ \texttt{forall}\texttt{<}\overline{\varphi}, \overline{\varrho}, \overline{\alpha}\texttt{>}\ |x_1 : \tau_1^{\text{S}}, \ldots, x_n : \tau_n^{\text{S}}|\ \rightarrow\ \tau_r^{\text{S}}\ \{\ e\ \}\ \rangle})$$

We then pick Γ_i to be $\boxed{\Gamma}$, and need to show:

$\Sigma \vdash \sigma : \Gamma$	Immediate from our premise.
$\Sigma; \bullet; \Gamma \vdash \boxed{e'} : \tau' \Rightarrow \Gamma$	By T-ClosureValue since σ_c is empty, and we know that the body itself is well-typed as a consequence of inversion on WF-FunctionDefinition for f.
$\bullet; \Gamma \vdash \tau' \lesssim \tau \Rightarrow \Gamma$	Immediate by S-Refl.
$\exists \Gamma_o . \Gamma \Cup \Gamma_o = \Gamma$	$\Gamma_o = \Gamma$

Case T-Closure:

From premise:

$$\textsc{T-Closure}\ \frac{\text{free-vars}(e) \setminus \overline{x} = \overline{x_f} \qquad \text{free-nc-vars}_\Gamma(e) = \overline{x_{nc}} \qquad \overline{r} = \overline{\text{free-provs}(\Gamma(x_f))},\ \text{free-provs}(e) \qquad \mathcal{F}_c = \overline{r \mapsto \Gamma(r)},\ \overline{x_f : \Gamma(x_f)} \qquad \Sigma;\ \Delta;\ \Gamma[x_{nc} \mapsto \Gamma(x_{nc})^\dagger] \uplus \mathcal{F}_c,\ x_1 : \tau_1^{\text{SI}}, \ldots, x_n : \tau_n^{\text{SI}} \vdash \boxed{e} : \tau_r^{\text{SI}} \Rightarrow \Gamma' \uplus \mathcal{F}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{|x_1 : \tau_1^{\text{SI}}, \ldots, x_n : \tau_n^{\text{SI}}|\ \rightarrow\ \tau_r^{\text{SI}}\ \{\ e\ \}} : (\tau_1^{\text{SI}}, \ldots, \tau_n^{\text{SI}}) \xrightarrow{\mathcal{F}_c} \tau_r^{\text{SI}} \Rightarrow \Gamma'}$$

Since $e = \texttt{forall<}\overline{\varphi}, \overline{\rho}, \overline{\alpha}\texttt{>}\ |x_1 : \tau_1^{\text{SI}}, \ldots, x_n : \tau_n^{\text{SI}}|\ \rightarrow\ \tau_r^{\text{SI}}\ \{\ e\ \}$, by inspection of the reduction rules, we know that e steps with the following rule:

$$\textsc{E-Closure}\ \frac{\text{free-vars}(e) = \overline{x_f} \qquad \text{free-nc-vars}_\sigma(e) = \overline{x_{nc}} \qquad \varsigma_c = \sigma\ |_{\overline{x_f}}}{\Sigma \vdash (\sigma;\ \boxed{|x_1 : \tau_1^{\text{S}}, \ldots, x_n : \tau_n^{\text{S}}|\ \rightarrow\ \tau_r^{\text{S}}\ \{\ e\ \}}) \rightarrow (\sigma[\overline{x_{nc} \mapsto \texttt{dead}}];\ \boxed{\langle \varsigma_c,\ |x_1 : \tau_1^{\text{S}}, \ldots, x_n : \tau_n^{\text{S}}|\ \rightarrow\ \tau_r^{\text{S}}\ \{\ e\ \}\ \rangle})}$$

We then pick Γ_i to be $\boxed{\Gamma[\overline{x_{nc} \mapsto \Gamma(x_{nc})^\dagger}]}$, and need to show:

$\Sigma \vdash \sigma[\overline{x_{nc} \mapsto \texttt{dead}}] : \Gamma[\overline{x_{nc} \mapsto \Gamma(x_{nc})^\dagger}]$	Compared to Γ, we know that $\Gamma[\overline{x_{nc} \mapsto \Gamma(x_{nc})^\dagger}]$ has more things marked dead and no other changes. Thus, we can apply R-Env to get $\Sigma;\ \bullet \vdash \Gamma \lesssim \Gamma[\overline{x_{nc} \mapsto \Gamma(x_{nc})^\dagger}]$. Then, we can apply Lemma E.23 to get $\Sigma \vdash \sigma : \Gamma[\overline{x_{nc} \mapsto \Gamma(x_{nc})^\dagger}]$. Since dead is good at every dead type τ^{SD} by T-Dead, we can then build a new derivation using that rule instead for every x_{nc} that is now at a dead type. This gives us $\Sigma \vdash \sigma[\overline{x_{nc} \mapsto \texttt{dead}}] : \Gamma[\overline{x_{nc} \mapsto \Gamma(x_{nc})^\dagger}]$.
$\Sigma;\ \bullet;\ \Gamma_i \vdash \boxed{e'} : \tau' \Rightarrow \Gamma_i$	Immediate by inversion of T-Closure and application of T-ClosureValue. Note that they have identical premises.
$\bullet;\ \Gamma_i \vdash \tau' \lesssim \tau' \Rightarrow \Gamma_i$	Immediate by S-Refl.
$\exists \Gamma_o . \Gamma_i \uplus \Gamma_o = \Gamma_i$	$\Gamma_o = \Gamma_i$

Case T-Shift:

From premise:

$$\textsc{T-Shift}\ \frac{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e} : \tau^{\text{SI}} \Rightarrow \Gamma',\ x : \tau^{\text{SD}}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\texttt{shift}\ e} : \tau^{\text{SI}} \Rightarrow \Gamma'}$$

Since $e = \texttt{shift}\ e_i$, by inspection of the reduction rules, we know that e steps with the following rule:

$$\textsc{E-Shift}\ \frac{}{\Sigma \vdash (\sigma,\ x \mapsto v';\ \boxed{\texttt{shift}\ v}) \rightarrow (\sigma;\ \boxed{v})}$$

We then pick Γ_i to be $\boxed{\Gamma'}$, and need to show:

$\Sigma \vdash \sigma : \Gamma'$	By inversion of WF-StackFrame on $\Sigma \vdash \sigma, x \mapsto v' : \Gamma', x : \tau^{\text{SD}}$, we get $\Sigma \vdash \sigma : \Gamma_i$, $\text{dom}(\varsigma, x \mapsto v') = \text{dom}(\mathcal{F}, x : \tau^{\text{SD}})|_x$, and $\forall x \in \text{dom}(\sigma \natural \varsigma, x \mapsto v')$. $\Sigma; \bullet; \Gamma_i \natural \mathcal{F}, x : \tau^{\text{SD}} \vdash \boxed{(\sigma \natural \varsigma, x \mapsto v')(x)} : (\Gamma_i \natural \mathcal{F}, x : \tau^{\text{SD}})(x) \Rightarrow \Gamma_i \natural \mathcal{F}, x : \tau^{\text{SD}}$. Note that $\Gamma_i \natural \mathcal{F} = \Gamma'$. We can then immediately see that the above implies $\text{dom}(\varsigma) = \text{dom}(\mathcal{F})|_x$ and $\forall x \in \text{dom}(\sigma \natural \varsigma, x \mapsto v')$. $\Sigma; \bullet; \Gamma_i \natural \mathcal{F} \vdash \boxed{(\sigma \natural \varsigma)(x)} : (\Gamma_i \natural \mathcal{F})(x) \Rightarrow \Gamma_i \natural \mathcal{F}$. Thus, we can apply WF-StackFrame to get $\Sigma \vdash \sigma : \Gamma_i \natural \varsigma$ which can be rewritten as $\Sigma \vdash \sigma : \Gamma'$.
$\Sigma; \bullet; \Gamma' \vdash \boxed{v} : \tau^{\text{SI}} \Rightarrow \Gamma'$	We know from E-Shift that e is a value v. Thus, we can apply Lemma E.21 to $\Sigma; \bullet; \Gamma \vdash \boxed{v} : \tau^{\text{SI}} \Rightarrow \Gamma', x : \tau^{\text{SD}}$ to get $\Sigma; \bullet; \Gamma', x : \tau^{\text{SD}} \vdash \boxed{v} : \tau^{\text{SI}} \Rightarrow \Gamma', x : \tau^{\text{SD}}$. We now wish to show that $\Sigma; \bullet; \Gamma' \vdash \boxed{v} : \tau^{\text{SI}} \Rightarrow \Gamma'$. By inspecting the grammar of values and their typing rules, we know that the only values who depend on the context are pointers and closure values. But by inversion on $\Sigma; \bullet; \Gamma \vdash \boxed{\texttt{shift}\, v} : \tau^{\text{SI}} \Rightarrow \Gamma'$, we know that $\Sigma; \bullet; \Gamma' \vdash \tau^{\text{SI}}$. Since the type is valid without the frame, we know that the values cannot depend on that frame. Thus, we can conclude $\Sigma; \bullet; \Gamma' \vdash \boxed{v} : \tau^{\text{SI}} \Rightarrow \Gamma'$.
$\bullet; \Gamma' \vdash \tau^{\text{SI}} \lesssim \tau^{\text{SI}} \Rightarrow \Gamma'$	Immediate by S-Refl.
$\exists \Gamma_o . \Gamma' \Cup \Gamma_o = \Gamma'$	$\Gamma_o = \Gamma'$

Case T-Framed:

From premise:

$$\text{T-Framed}\ \frac{\Sigma; \Delta; \Gamma \vdash \boxed{e} : \tau^{\text{SI}} \Rightarrow \Gamma' \natural \mathcal{F}'}{\Sigma; \Delta; \Gamma \vdash \boxed{\texttt{framed}\, e} : \tau^{\text{SI}} \Rightarrow \Gamma'}$$

Since $e = \texttt{framed}\ e_i$, by inspection of the reduction rules, we know that e steps with the following rule:

$$\text{E-Framed}\ \frac{}{\Sigma \vdash (\sigma \natural \varsigma; \boxed{\texttt{framed}\, v}) \rightarrow (\sigma; \boxed{v})}$$

We then pick Γ_i to be $\boxed{\Gamma'}$, and need to show:

$\Sigma \vdash \sigma : \Gamma'$	Applying Lemma E.24 to $\Sigma \vdash \sigma \natural \varsigma : \Gamma' \natural \mathcal{F}$ gives us $\Sigma \vdash \sigma : \Gamma'$.
$\Sigma; \bullet; \Gamma' \vdash \boxed{v} : \tau^{\text{SI}} \Rightarrow \Gamma'$	We know from E-Framed that e is a value v. Thus, we can apply Lemma E.21 to $\Sigma; \bullet; \Gamma \natural \mathcal{F} \vdash \boxed{v} : \tau^{\text{SI}} \Rightarrow \Gamma' \natural \mathcal{F}'$ to get $\Sigma; \bullet; \Gamma' \natural \mathcal{F}' \vdash \boxed{v} : \tau^{\text{SI}} \Rightarrow \Gamma' \natural \mathcal{F}'$. We now wish to show that $\Sigma; \bullet; \Gamma' \vdash \boxed{v} : \tau^{\text{SI}} \Rightarrow \Gamma'$. By inspecting the grammar of values and their typing rules, we know that the only values who depend on the context are pointers and closure values. But by inversion on $\Sigma; \bullet; \Gamma \vdash \boxed{\texttt{framed}\, v} : \tau^{\text{SI}} \Rightarrow \Gamma'$, we know that $\Sigma; \bullet; \Gamma' \vdash \tau^{\text{SI}}$. Since the type is valid without the frame, we know that the values cannot depend on that frame. Thus, we can conclude $\Sigma; \bullet; \Gamma' \vdash \boxed{v} : \tau^{\text{SI}} \Rightarrow \Gamma'$.
$\bullet; \Gamma' \vdash \tau^{\text{SI}} \lesssim \tau^{\text{SI}} \Rightarrow \Gamma'$	Immediate by S-Refl.
$\exists \Gamma_o . \Gamma' \Cup \Gamma_o = \Gamma'$	$\Gamma_o = \Gamma'$

Case T-Drop:

From premise:

$$\textsc{T-Drop}\quad \frac{\Gamma(\pi) = \tau_\pi^{\text{SI}} \qquad \Sigma;\, \Delta;\, \Gamma[\pi \mapsto \tau_\pi^{\text{SI}^\dagger}] \vdash \boxed{e} : \tau^{\text{SX}} \Rightarrow \Gamma_f}{\Sigma;\, \Delta;\, \Gamma \vdash \boxed{e} : \tau^{\text{SX}} \Rightarrow \Gamma_f}$$

Since T-Drop applies to *any* expression e, we cannot determine anything about what rule we stepped with. So, we will instead try to apply our induction hypothesis to the typing derivation in the premise of T-Drop $(\Sigma;\, \bullet;\, \Gamma[\pi \mapsto \tau_\pi^{\text{SI}^\dagger}] \vdash \boxed{e} : \tau^{\text{SX}} \Rightarrow \Gamma_f)$. To do this, we need to establish the premises of our inductive hypothesis.

Namely, we need to show:

(1) $\Sigma;\, \bullet;\, \Gamma[\pi \mapsto \tau_\pi^{\text{SI}^\dagger}] \vdash \boxed{e} : \tau^{\text{SX}} \Rightarrow \Gamma_f$,
(2) $\Sigma \vdash \sigma : \Gamma[\pi \mapsto \tau_\pi^{\text{SI}^\dagger}]$,
(3) $\Sigma \vdash (\sigma;\, \boxed{e}) \rightarrow (\sigma';\, \boxed{e'})$.

(1) follows immediately from our premise.
(2) follows almost directly from our premise, which tells us that $\Sigma \vdash \sigma : \Gamma$. We just need to show that the value at x (where x is the root of π) is still valid at its new type. Fortunately, it's old derivation works almost perfectly except for typing the part that corresponds directly to π. In this case, we can use T-Dead on the value to get the new derivation with that part of the aggregate structure at the uninitialized type $\tau_\pi^{\text{SI}^\dagger}$.
(3) follows immediately from our premise.

This allows us to use our induction hypothesis to conclude that there exists Γ_i' such that:

(5) $\Sigma \vdash \sigma' : \Gamma_i'$,
(6) $\Sigma;\, \bullet;\, \Gamma_i' \vdash \boxed{e'} : \tau' \Rightarrow \Gamma_f'$,
(7) $\bullet;\, \Gamma_f' \vdash \tau' \lesssim \tau^{\text{SX}} \Rightarrow \Gamma_s$, and
(8) $\exists \Gamma_o.\ \Gamma_s \ \mathbin{\mathbb{U}}\ \Gamma_o = \Gamma_f$.

□

E.4 Type Safety

Theorem E.32 (Type Safety). *If* $\Sigma;\, \bullet;\, \bullet \vdash \boxed{e} : \tau^{SI} \Rightarrow \Gamma$ *and* $\vdash \Sigma$, *then* $\Sigma \vdash (\bullet;\, \boxed{e}) \rightarrow^* (\sigma';\, \boxed{v})$ *or evaluation of* e *aborts or diverges.*

Proof. By the interleaved use of **Progress** and **Preservation**. □