Oxide: The Essence of Rust

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

Rust is a major advancement in industrial programming languages due in large part to its success in bridging the gap between *low-level* systems programming and *high-level* application programming. This success has ultimately empowered programmers to more easily build reliable and efficient software, and at its heart lies a novel approach to *ownership* that balances type system expressivity with usability.

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. To that end, 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 *approximate provenances* of references, and our type system is able to automatically compute this information through a flow-sensitive substructural typing judgment for which we prove syntactic type safety using progress and preservation. The result 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.

CCS Concepts: • **Theory of computation → Semantics and reasoning**; • **Software and its engineering → Formal language definitions**.

Additional Key Words and Phrases: Rust, type systems, semantics, ownership

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*. Thus, we hold that Rust has something interesting to teach us about making ownership *practical* for programming.

Despite its success, Rust has also developed something of a reputation for its complexity among programmers. It's not uncommon to hear folks exchange tales of *fighting the borrow checker* after dipping their toes into Rust. It is natural then to wonder if this experience is *inevitable*. We say not! While there is likely some inherent complexity to a more powerful type system, we feel that the fundamental issue at hand is clearer — namely, learning new semantics is *hard*. Rust's designers have spoken at length about language complexity, and how they've worked to manage it by making much of the syntax of Rust similar to languages like C++ and Java [Klabnik 2015; Turon and Hicks 2015]. But a deeper approach to lowering the perceived complexity of the language is to develop a more thorough understanding of its complexity — to find its simpler, more explainable essence! Featherweight Java [Igarashi et al. 2001] did just that — illuminating the language being studied *and* providing a foundation for future research. This inspired our own effort, and so we endeavor in this work to distill *the essence of Rust* through our formalization, Oxide.

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

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] whose λ_{Rust} has already proven useful in verifying that major parts of Rust's standard library (written using `unsafe` code) do not violate its safety guarantees. Yet, for our purposes, its continuation-passing style and low-level nature make it difficult to use for *source-level* reasoning.

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 formal complexities of type inference,[1] and, as we will discuss in more detail in Section 2.3, we do not model some simple syntactic checks on the mutability of variable bindings. Further, since we are interested in understanding ownership, we focus on the *safe* portion of Rust without standard library abstractions implemented using `unsafe` code. In Sections 6.1 and 6.2, we discuss extensions to Oxide that address this simplification, including a sketch of support for heap allocation with Rust's `Vec` type. As a result of its high-level nature and these simplifications, we are able to prove Oxide is type-safe syntactically [Wright and Felleisen 1992] using an operational semantics instrumented with runtime safety checks. Ultimately, the balance Oxide strikes allows us to develop a more explainable *essence* of Rust.

The rest of the paper is organized as follows: Section 2 describes the essence of Rust and Oxide at an intuitive level. Section 3 presents the formal details of Oxide including the syntax (Section 3.1), type system (Section 3.2), operational semantics (Section 3.3), and metatheory (Section 3.4). Section 4 works through interesting examples of borrow-checking in Rust and Oxide to capture how the latter models the former. Section 5 discusses related work on formal semantics for Rust, and more broadly on *practical substructural programming*. Section 6 explores avenues for future work building on Oxide, and finally, Section 7 concludes.

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 will 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

[1] Rust uses a variant of Hindley-Milner type inference [Hindley 1969; Matsakis and Contributors 2018; Milner 1978].

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 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.[2] 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
3  let mut pt = Point(6, 9);
4  let x = &pt;
5  let y = &pt; // no error, sharing is okay!
```

In the above example, we replaced the *move* expressions on lines 4 and 5 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. 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 ones instead:

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

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 another important change happened — we added some additional code afterward that somehow makes use of `x` and `y`. This is important because of a feature in Rust known as *non-lexical lifetimes*

[2] The use of "such" here is 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.

(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. The reason we say that y is also used is more subtle (and the details are not entirely resolved in Rust), but we'll return to the subject in Section 6.3.

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

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

In this case, we've changed the borrow expression on line 5 to create a shared, rather than unique, reference. We've also chosen to add explicit type annotations to our bindings on lines 3–5. 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 (including the uniqueness of the loan and its origin) by the creation of a reference. Regions[3] 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 [Ahmed et al. 2005; 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 through the same object, as in the following example:

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

[3]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 Section 4.3.

2.3 Formalizing Rust

Notably, in Oxide, these programs are largely unchanged. The main differences from Rust are threefold. First, we must explicitly annotate the type of every binding. Secondly, acknowledging that **mut** plays two distinct roles in Rust, we have simplified our semantics by focusing 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 the simple associated check. Finally, we also shift the language we use to talk about regions or lifetimes. In particular, acknowledging that regions capture approximations of a reference's origin, we instead use a more precise term, *approximate provenances*, and refer to their variable form ('a, 'b, etc.) as *provenance variables*. Translating our last example into Oxide results in the following program:

```
1  struct Point(u32, u32);
2
3  let pt: Point = Point(6, 9);
4  let x: &'x uniq u32 = &uniq pt.0;
5  let y: &'y uniq u32 = &uniq pt.1;
6  // 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. The remainder of the program is otherwise unchanged. As in the Rust version, the Oxide version type checks correctly because the origins of the loans do not overlap. That is, `x` can only have originated from `pt.0` and `y` only from `pt.1`.

During type-checking, Oxide will determine concrete values for the provenance variables in the program ('x and 'y), i.e. 'x will be mapped to { $^{\texttt{uniq}}$`pt.0` } and 'y to { $^{\texttt{uniq}}$`pt.1` }. When type-checking each borrow expression, Oxide will look at the existing places in its place environment Γ to determine that there are no live loans to any place that overlaps with the place being borrowed. At runtime, the program evaluates with a stack σ that is analogous to the place environment Γ. That is, it maps places to *shapes* which have their types given in Γ. We will explain shapes in more detail in Section 3.3, but intuitively, they are a destructuring of values that enable us to refer to each individually-borrowable component of a value without duplication.

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 alive. As such, we combine the return types and environments from branches. We will come back to this with a more formal treatment in Section 3.2.

2.4 Non-Lexical Lifetimes in Oxide

In Oxide, we allow non-lexical lifetimes through the use of *weakening*. As one might likely have expected, Oxide has an *affine* type system. To illustrate how weakening gives us support for non-lexical lifetimes, let us consider a variant of one of our earlier examples where our first unique pointer is instead *not* used in the remainder of the program:

```
struct Point(u32, u32);

let pt: Point = Point(6, 9);
let x: &'x uniq Point = &uniq pt;
let y: &'y uniq Point = &uniq pt;
... // additional code using y, but not x
```

In this case, 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, by weakening, we can drop x from our context, ending all of the loans in 'x, and allowing us to successfully proceed with the rest of the program. If x *was* used in the rest of the program, we would then encounter a new point where we cannot make a typing derivation since the variable x would be unbound. However, if x is not used, the rest of the program will succeed since it necessarily did not depend on the existence of x. Intuitively, we are saying that unique references that are unused may as well not exist.

2.5 Oxide, More Formally

At this point, we've now seen enough 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 flow-sensitive typing judgment in an environment-passing style. The shape of our judgment is $\Sigma;\ \Delta;\ \Gamma \vdash e : \tau \Rightarrow \Gamma'$, where Σ is the global environment denoting the top-level definitions of the program including both function and type definitions, Δ is the type variable environment tracking in-scope type and provenance variables and their kinds, and Γ is the place environment mapping *places* π (as a sort of generalization of variables) to their respective types. The output environment Γ' denotes the place environment *after* typing this expression, which is essential in capturing the substructural aspects of Oxide.

$$\frac{\Gamma \vdash_{\mathsf{uniq}} \pi \ :\ \tau^{\mathrm{s}} \qquad \mathsf{noncopyable}\ \tau^{\mathrm{s}}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\pi} : \tau^{\mathrm{s}} \Rightarrow \Gamma - \pi}\ \textsc{T-Move} \qquad \frac{\Gamma \vdash_{\omega} \pi \ :\ \tau}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\&\omega\ \pi} : \&\{\ {}^{\omega}\pi\ \}\ \omega\ \tau \Rightarrow \Gamma}\ \textsc{T-Borrow}$$

Fig. 1. The essence of Oxide.

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 judgment in their premises: $\Gamma \vdash_{\omega} \pi \ :\ \tau$. We can read it as saying "in the environment Γ, it is safe to use π (of type τ) ω-ly." That is, if we have a derivation when ω is uniq, we know that we can use the place π uniquely because we have a proof that there are no live loans against the section of memory that π represents. This instance of the judgment appears in the premise of T-Move because we know that it is only safe to move a value out of the environment when it is the sole name for that value. Further, when we have a derivation of this ω-safety judgment where ω is shrd, we know that we can use the place π sharedly because we have a proof that there are no live *unique* loans against the section of memory that π represents. In the case of borrowing (as in T-Borrow), these two meanings of ω-safety correspond exactly to the intuition behind when a ω-loan is safe to create.

Since it is precisely this 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, the ω-safety judgment looks through all of the approximate provenances found within types in Γ, and ensures that none of the loans they contain conflict with the place π in question. For a `uniq` loan, a conflict occurs if any loan maps to an overlapping place, but for a `shrd` loan, a conflict occurs only when a `uniq` loan maps to an overlapping place. In the rest of the paper, we will explore the formalism in more detail along with the possibilities and consequences of this new model for Rust.

Variables	x	Type Variables	α	Provenance Variables	ρ
Struct Names	S	String	`str`	Naturals	m, n, k
		Function Names	f		

Ownership Qualifiers	ω	::=	`shrd` | `uniq`
Places	π	::=	$x \mid *\pi \mid \pi.x \mid \pi.n$
Loans	ℓ	::=	${}^{\omega}\pi$
Approx. Provenance	ϱ	::=	$\rho \mid \{ \ell_1, \ldots, \ell_n \}$
Kinds	κ	::=	$\star$ | `PRV`
Base Types	τ^{B}	::=	`bool` | `u32` | `unit`
Unsized Types	τ^{U}	::=	$[\tau^{\mathrm{S}}]$
Sized Types	τ^{S}	::=	$\tau^{\mathrm{B}} \mid \alpha \mid \&\varrho\ \omega\ \tau$
		|	$\forall <\overline{\rho}, \overline{\alpha}> (\tau_1^{\mathrm{S}}, \ldots, \tau_n^{\mathrm{S}}) \xrightarrow{\Gamma} \tau_r^{\mathrm{S}}$
		|	$[\tau^{\mathrm{S}}; n] \mid (\tau_1^{\mathrm{S}}, \ldots, \tau_n^{\mathrm{S}}) \mid \mathrm{S}<\overline{\varrho}, \overline{\tau}>$
Types	τ	::=	$\tau^{\mathrm{U}} \mid \tau^{\mathrm{S}}$
Constants	c	::=	$() \mid n \mid$ `true` | `false`
Expressions	e	::=	$c \mid \pi$
		|	$\&\omega\ \pi \mid \&\omega\ \pi[e] \mid \&\omega\ \pi[e_1..e_2]$
		|	`let` $x : \tau^{\mathrm{S}} = e_1;\ e_2 \mid \pi := e \mid e_1;\ e_2$
		|	`forall`$<\overline{\rho}, \overline{\alpha}> \lvert x_1 : \tau_1^{\mathrm{S}}, \ldots, x_n : \tau_n^{\mathrm{S}} \rvert \rightarrow \tau_r^{\mathrm{S}} \{ e \}$
		|	$e_f ::<\overline{\rho}, \overline{\alpha}> (e_1, \ldots, e_n)$
		|	$\pi[e] \mid$ `abort!(str)`
		|	`if` $e_1 \{ e_2 \}$ `else` $\{ e_3 \}$
		|	`for` x `in` $e_1 \{ e_2 \}$
		|	$(e_1, \ldots, e_n) \mid [e_1, \ldots, e_n]$
		|	$\mathrm{S} ::<\overline{\rho}, \overline{\alpha}> \{ x_1 : e_1, \ldots, x_n : e_n \}$
		|	$\mathrm{S} ::<\overline{\rho}, \overline{\alpha}> (e_1, \ldots, e_n)$
Global Environment	Σ	::=	$\bullet$
		|	Σ, `fn` $f <\overline{\rho}, \overline{\alpha}> (x_1 : \tau_1^{\mathrm{S}}, \ldots, x_n : \tau_n^{\mathrm{S}}) \rightarrow \tau_r^{\mathrm{S}} \{ e \}$
		|	Σ, `struct` $\mathrm{S}<\overline{\rho}, \overline{\alpha}> (\tau_1^{\mathrm{S}}, \ldots, \tau_n^{\mathrm{S}})$
		|	Σ, `struct` $\mathrm{S}<\overline{\rho}, \overline{\alpha}> \{ x_1 : \tau_1^{\mathrm{S}}, \ldots, x_n : \tau_n^{\mathrm{S}} \}$
Type Var. Environment	Δ	::=	$\bullet \mid \Delta, \rho : \mathtt{PRV} \mid \Delta, \alpha : \star$
Place Environment	Γ	::=	$\bullet \mid \Gamma, \pi : \tau$

Fig. 2. Selected Oxide Syntax

3 OXIDE

3.1 Syntax

Figure 2 presents most of the syntax of Oxide. In Oxide, we annotate references with ownership qualifiers ω, where references that can be shared are marked `shrd` and unique references are marked `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 *ownership*, 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>` *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 Rust works.

Places. As discussed at a high-level in Section 2.2, 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, a projection of a tuple, or the referent of a pointer. One might think of these places as a sort of syntactic generalization of variables. They're analogous to what is often called `lvalues` in the world of C.

Approximate Provenances. Approximate provenances (sometimes shortened to provenances) are either a provenance variable ρ or a set of loans ℓ that denote a place π of possible origin qualified by ω (written ${}^{\omega}\pi$). Intuitively, these loans tell us a single possible origin for the reference, and an approximate provenance gives us *all* possible origins. Approximate provenances are essential to enabling our type system to guarantee the correct use of unique and shared references.

Types and Kinds. Oxide has two kinds, $\star$, the kind of ordinary types, and `PRV`, the kind of provenances. To aid the reader in differentiating between variables of kind $\star$ and variables of kind `PRV`, we write the former as type variables α and the latter as provenance variables ρ. Since Rust is a fairly low-level language, we need to distinguish between types that have a statically-known size and types that do not. In Oxide, we call the former sized types (τ^{S}), and the latter unsized types (τ^{U}). In our case, the only type whose size cannot be known statically is the type of slices ($[\tau^{\mathrm{S}}]$), which represents a segment of a fixed-sized array ($[\tau^{\mathrm{S}};\ n]$). Since its size is statically unknown, it would be impossible to place a value at the type $[\tau^{\mathrm{S}}]$ onto the stack, and as such, we can only create and use such values behind a reference type (whose size is always known). We write reference types as $\&\varrho\ \omega\ \tau$, where we see both provenances ϱ and ownership qualifiers ω at play. Since references in Rust are the types that express borrowing, their types involve many of the novel parts of Oxide.

Our other sized types include tuples $(\tau_1, \ldots, \tau_n)$ and structs $\mathrm{S}\langle\overline{\varrho}, \overline{\tau^{\mathrm{S}}}\rangle$, the latter of which is instantiated with provenances and types since struct definitions allow polymorphism over both provenance and type variables. Here, $\overline{\varrho}$ and $\overline{\tau^{\mathrm{S}}}$ are abbreviations for $\varrho_1, \ldots, \varrho_m$ and $\tau_1^{\mathrm{S}}, \ldots, \tau_n^{\mathrm{S}}$ respectively. We use this notation consistently to make things easier to read. Finally, we have functions of type $\forall\langle\overline{\rho}, \overline{\alpha}\rangle(\tau_1^{\mathrm{S}}, \ldots, \tau_n^{\mathrm{S}}) \xrightarrow{\Gamma} \tau_r^{\mathrm{S}}$, which are similarly polymorphic over provenance and type variables and take n arguments to a single result. Critically, our function type has a type environment Γ over the arrow which corresponds to the collection of captured places and their types in the body of the closure. For top-level functions (from Σ), there is nothing to capture, leaving this environment empty, and so their types are written with nothing over the arrow.

Expressions. Expressions 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 usage (written simply π) 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

$\boxed{\Gamma \vdash_\omega \pi : \tau}$

$$\textsc{V-Safe}\ \frac{}{\pi : \tau \vdash_\omega \pi : \tau} \qquad \textsc{V-UniqueSafe}\ \frac{\forall\, {}^{\omega'}\pi' \in \varrho.\ \neg \texttt{overlapping}(\pi, \pi') \qquad \pi_b \neq \pi \qquad \Gamma \vdash_{\texttt{uniq}} \pi : \tau}{\Gamma, \pi_b : \&\varrho\ \omega\ \tau_b \vdash_{\texttt{uniq}} \pi : \tau}$$

$$\textsc{V-SharedSafe}\ \frac{\forall\, {}^{\texttt{uniq}}\pi' \in \varrho.\ \neg \texttt{overlapping}(\pi, \pi') \qquad \pi_b \neq \pi \qquad \Gamma \vdash_{\texttt{shrd}} \pi : \tau}{\Gamma, \pi_b : \&\varrho\ \texttt{uniq}\ \tau_b \vdash_{\texttt{shrd}} \pi : \tau} \qquad \textsc{V-SharedPassthrough}\ \frac{\Gamma \vdash_{\texttt{shrd}} \pi : \tau}{\Gamma, \pi_b : \&\varrho\ \texttt{shrd}\ \tau_b \vdash_{\texttt{shrd}} \pi : \tau}$$

$$\textsc{V-SafeClosure}\ \frac{\Gamma, \Gamma_c \vdash_\omega \pi : \tau}{\Gamma, \pi_b : \forall <\overline{\rho}, \overline{\alpha}> (\overline{\tau^{\texttt{s}}}) \xrightarrow{\Gamma_c} \tau_r^{\texttt{s}} \vdash_\omega \pi : \tau} \qquad \textsc{V-SafePassthrough}\ \frac{\tau_b \neq \&\varrho\ \omega'\ \tau' \qquad \tau_b \neq \forall <\overline{\rho}, \overline{\alpha}> (\overline{\tau^{\texttt{s}}}) \xrightarrow{\Gamma_c} \tau_r^{\texttt{s}} \qquad \pi_b \neq \pi \qquad \Gamma \vdash_\omega \pi : \tau}{\Gamma, \pi_b : \tau_b \vdash_\omega \pi : \tau}$$

Fig. 3. μ-Safety in Oxide

borrowing forms overall, and all work fundamentally the same — they are each used as introduction forms for references. The simplest case is written $\&\omega\ \pi$, introducing a ω reference directly to π. The next form borrows from $\pi[e]$ instead of simply π, and is used to borrow an element out of an array or slice at the index given by e. The final form borrows from $\pi[e_1..e_2]$, and is used to borrow a slice of π 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, and our 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 equivalently create two unique references to different indices of an array.

The remainder of our expressions are standard, but we will draw attention to a few points of note. Our closure syntax follows the syntax of Rust, and thus quantifies over both provenance and type variables *and* uses vertical bars to denote the closure's parameters. Similarly, function application additionally includes instantiation of these provenance and type variables 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.

Environments. As we have already seen in Section 2.5, we have three environments in Oxide. First, we have a global environment Σ that consists of top-level function definitions and definitions for struct types, either with positional or named fields. Next, we have a type variable environment that can contain both provenance variables ρ and type variables α with their respective kinds (PRV and $\star$). Finally, our place environment Γ maps in-scope places π to their types.

3.2 Type System

Figure 4 presents a selection of Oxide typing rules. In every rule, we highlight expressions with frameboxes. As described in Section 2.5, the shape of our typing judgement is $\Sigma;\ \Delta;\ \Gamma \vdash e : \tau \Rightarrow \Gamma'$, where Σ denotes the global environment containing top-level definitions including both function and type definitions, Δ is the type variable environment tracking in-scope type and provenance variables and their kinds, and Γ is the place environment mapping *places* π to their respective types.

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

T-Move
$$\frac{\Gamma \vdash_{\mathsf{uniq}} \pi\ :\ \tau^{\mathrm{S}} \qquad \mathsf{noncopyable}\ \tau^{\mathrm{S}}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\pi} : \tau^{\mathrm{S}} \Rightarrow \Gamma - \pi}$$

T-Copy
$$\frac{\Gamma \vdash_{\mathsf{shrd}} \pi\ :\ \tau^{\mathrm{S}} \qquad \mathsf{copyable}\ \tau^{\mathrm{S}}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\pi} : \tau^{\mathrm{S}} \Rightarrow \Gamma}$$

T-Borrow
$$\frac{\Gamma \vdash_{\omega} \pi\ :\ \tau}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\&\omega\ \pi} : \&\{\ ^{\omega}\pi\ \}\ \omega\ \tau \Rightarrow \Gamma}$$

T-Seq
$$\frac{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e_1} : \tau_1^{\mathrm{S}} \Rightarrow \Gamma_1 \qquad \Sigma;\ \Delta;\ \Gamma_1 \vdash \boxed{e_2} : \tau_2^{\mathrm{S}} \Rightarrow \Gamma_2}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e_1;\ e_2} : \tau_2^{\mathrm{S}} \Rightarrow \Gamma_2}$$

T-Branch
$$\frac{\begin{array}{c}\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e_1} : \mathsf{bool} \Rightarrow \Gamma_1 \\ \Sigma;\ \Delta;\ \Gamma_1 \vdash \boxed{e_2} : \tau_2^{\mathrm{S}} \Rightarrow \Gamma_2 \qquad \Sigma;\ \Delta;\ \Gamma_1 \vdash \boxed{e_3} : \tau_3^{\mathrm{S}} \Rightarrow \Gamma_3 \\ \tau_2^{\mathrm{S}} \sim \tau_3^{\mathrm{S}} \Rightarrow \tau^{\mathrm{S}} \qquad \Gamma_2 \Cap \Gamma_3 = \Gamma' \qquad \Gamma' \vdash \tau^{\mathrm{S}}\end{array}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\mathtt{if}\ e_1\ \{\ e_2\ \}\ \mathtt{else}\ \{\ e_3\ \}} : \tau^{\mathrm{S}} \Rightarrow \Gamma'}$$

T-Assign
$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{uniq}} \pi\ :\ \tau_o \qquad \tau_o \sim \tau_u \Rightarrow \tau_n \\ \Sigma;\ \Delta;\ \Gamma \vdash \boxed{e} : \tau_u \Rightarrow \Gamma_1 \\ \mathsf{places\text{-}typ}(\pi,\ \tau_u) = \overline{\pi\ :\ \tau}\end{array}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\pi := e} : \mathsf{unit} \Rightarrow \Gamma_1 - \pi,\ \overline{\pi\ :\ \tau}}$$

T-Let
$$\frac{\begin{array}{c}\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e_1} : \tau_1^{\mathrm{S}} \Rightarrow \Gamma_1 \qquad \Sigma \vdash \tau_1^{\mathrm{S}} <: \tau_a^{\mathrm{S}} \rightsquigarrow \delta \\ \mathsf{places\text{-}typ}(x,\ \delta(\tau_a^{\mathrm{S}})) = \overline{\pi\ :\ \tau} \\ \Sigma;\ \Delta;\ \Gamma_1,\ \overline{\pi\ :\ \tau} \vdash \boxed{\delta(e_2)} : \tau_2^{\mathrm{S}} \Rightarrow \Gamma_2\end{array}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\mathtt{let}\ x : \tau_a^{\mathrm{S}} = e_1;\ e_2} : \tau_2^{\mathrm{S}} \Rightarrow \Gamma_2 - x}$$

T-App
$$\frac{\begin{array}{c}\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e_f} : \forall\!<\!\overline{\rho},\ \overline{\alpha}\!>\!(\tau_1^{\mathrm{S}},\ \ldots,\ \tau_n^{\mathrm{S}}) \xrightarrow{\Gamma_c} \tau_f^{\mathrm{S}} \Rightarrow \Gamma_f \\ \Sigma;\ \Delta;\ \Gamma_f \vdash \boxed{e_1} : \tau_1^{\mathrm{S}}\,\overline{[\varrho/\rho]}\,\overline{[\tau^{\mathrm{S}}/\alpha]} \Rightarrow \Gamma_1 \qquad \ldots \qquad \Sigma;\ \Delta;\ \Gamma_{n-1} \vdash \boxed{e_n} : \tau_n^{\mathrm{S}}\,\overline{[\varrho/\rho]}\,\overline{[\tau^{\mathrm{S}}/\alpha]} \Rightarrow \Gamma_n\end{array}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e_f ::\!<\!\overline{\varrho},\ \overline{\tau^{\mathrm{S}}}\!>\!(e_1,\ \ldots,\ e_n)} : \tau_f^{\mathrm{S}}\,\overline{[\varrho/\rho]}\,\overline{[\tau^{\mathrm{S}}/\alpha]} \Rightarrow \Gamma_n}$$

T-Closure
$$\frac{\begin{array}{c}\mathsf{places\text{-}typ}(x_1,\ \tau_1^{\mathrm{S}}) = \overline{\pi_1\ :\ \tau_1} \qquad \ldots \qquad \mathsf{places\text{-}typ}(x_n,\ \tau_n^{\mathrm{S}}) = \overline{\pi_n\ :\ \tau_n} \\ \Gamma_n \vdash \tau_1^{\mathrm{S}},\ \ldots,\ \tau_n^{\mathrm{S}} \qquad \Sigma;\ \Delta,\ \overline{\rho : \mathsf{PRV}},\ \overline{\alpha : \star};\ \Gamma,\ \overline{\pi_1\ :\ \tau_1},\ \ldots,\ \overline{\pi_n\ :\ \tau_n} \vdash \boxed{e} : \tau_r^{\mathrm{S}} \Rightarrow \Gamma' \qquad \Gamma_c = \Gamma \setminus \Gamma'\end{array}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\mathtt{forall}\!<\!\overline{\rho},\ \overline{\alpha}\!>\ |x_1 : \tau_1^{\mathrm{S}},\ \ldots,\ x_n : \tau_n^{\mathrm{S}}|\ \rightarrow\ \tau_r^{\mathrm{S}}\ \{\ e\ \}} : \forall\!<\!\overline{\rho},\ \overline{\alpha}\!>\!(\tau_1^{\mathrm{S}},\ \ldots,\ \tau_n^{\mathrm{S}}) \xrightarrow{\Gamma_c} \tau_r^{\mathrm{S}} \Rightarrow \Gamma \setminus \Gamma_c}$$

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

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

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

T-Tuple
$$\frac{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e_1} : \tau_1^{\mathrm{S}} \Rightarrow \Gamma_1 \qquad \ldots \qquad \Sigma;\ \Delta;\ \Gamma_{n-1} \vdash \boxed{e_n} : \tau_n^{\mathrm{S}} \Rightarrow \Gamma_n \qquad \Gamma_n \vdash \tau_1^{\mathrm{S}},\ \ldots,\ \tau_n^{\mathrm{S}}}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{(e_1,\ \ldots,\ e_n)} : (\tau_1^{\mathrm{S}},\ \ldots,\ \tau_n^{\mathrm{S}}) \Rightarrow \Gamma_n}$$

Fig. 4. Selected Oxide Typing Rules

$\boxed{\tau_1 \sim \tau_2 \Rightarrow \tau}$

$$\textsc{U-Symmetry}\ \frac{\tau_2 \sim \tau_1 \Rightarrow \tau}{\tau_1 \sim \tau_2 \Rightarrow \tau} \qquad \textsc{U-BaseTypes}\ \frac{}{\tau^{\mathrm{B}} \sim \tau^{\mathrm{B}} \Rightarrow \tau^{\mathrm{B}}} \qquad \textsc{U-TypeVars}\ \frac{}{\alpha \sim \alpha \Rightarrow \alpha} \qquad \textsc{U-Array}\ \frac{\tau_1 \sim \tau_2 \Rightarrow \tau}{[\tau_1;\ n] \sim [\tau_2;\ n] \Rightarrow [\tau;\ n]}$$

$$\textsc{U-Ref}\ \frac{\tau_1 \sim \tau_2 \Rightarrow \tau \qquad \varrho_1 \cup \varrho_2 = \varrho}{\&\varrho_1\ \omega\ \tau_1 \sim \&\varrho_2\ \omega\ \tau_2 \Rightarrow \&\varrho\ \omega\ \tau} \qquad \textsc{U-RefDiffOmega}\ \frac{\tau_1 \sim \tau_2 \Rightarrow \tau \qquad \omega_1 \neq \omega_2 \qquad \varrho_1 \cup \varrho_2 = \varrho}{\&\varrho_1\ \omega_1\ \tau_1 \sim \&\varrho_2\ \omega_2\ \tau_2 \Rightarrow \&\varrho\ \mathtt{shrd}\ \tau} \qquad \textsc{U-Slice}\ \frac{\tau_1 \sim \tau_2 \Rightarrow \tau}{[\tau_1] \sim [\tau_2] \Rightarrow [\tau]}$$

$$\textsc{U-Tuple}\ \frac{\overline{\tau_1 \sim \tau_2 \Rightarrow \tau}}{(\overline{\tau_1}) \sim (\overline{\tau_2}) \Rightarrow (\overline{\tau})} \qquad \textsc{U-Struct}\ \frac{\overline{\tau_1 \sim \tau_2 \Rightarrow \tau} \qquad \overline{\varrho_1 \cup \varrho_2 = \varrho}}{\mathrm{S}::<\overline{\varrho_1},\ \overline{\tau_1}> \sim \mathrm{S}::<\overline{\varrho_2},\ \overline{\tau_2}> \Rightarrow \mathrm{S}::<\overline{\varrho},\ \overline{\tau}>}$$

Fig. 5. Type Unification for Oxide

Moving. The T-Move rule, which was first introduced in Section 2.5, types place usages that must *move* the value out of π. As such, it requires two things: (1) π must be able to be used `uniq`-ly (checked using the ω-safety judgment in Figure 3), and (2) the type of π is `noncopyable`. The former requirement is essential to ensure that we do not invalidate any existing references to π by moving it. The latter reflects that we prefer to use T-Copy, which is more permissive, when allowed by the type of π. When both parts of the premise hold, the output environment removes the place π, reflecting that it has been moved after type-checking the expression. When combined with larger terms, we can see that the value may have moved into a new place via the use of `let`. If the type *is* instead `copyable`,[4] we use T-Copy, which only requires that π is safe to use as `shrd`. In this case, since dynamically the value will be copied, it is safe to leave the output environment unchanged.

Borrowing. In Section 2.5, we also introduced T-Borrow, which requires that the place π be safe to use (again checked with ω-safety in Figure 3), and if so, introduces a *concrete* provenance containing a single ω-qualified π. It is important to note that although T-Borrow only introduces trivial provenances, these provenances are indeed *approximate*. We will see in later rules how these are combined via type unification (defined in Figure 5) to yield larger sets.

Branching and Sequencing. The T-Seq rule captures how variable environments are threaded through larger programs, since type-checking some expressions makes changes to Γ. To do so, we type-check the second expression in the sequence under the environment we got from type-checking the first expression. The rule is otherwise standard. By contrast, T-Branch, our rule for `if` expressions, is non-standard in a few ways: first, as one might now expect, the environments are threaded, but the reader will also note that the output types from each branch are not required to be equal, but merely to unify via $\tau_1 \sim \tau_2 \Rightarrow \tau_3$. This unification, defined in Figure 5, combines the provenances found in reference types, while requiring the other aspects of the types to be identical. As such, its most interesting rule is U-Ref where we union the provenance sets in the two reference types to yield a new reference type. This is important as it allows branches to yield references to different places on each side of the branch. Finally, for the output environment, we take the intersection of the output environments from each branch. This intersection is denoted $\Cap$ rather

[4] We elide these judgments from the paper, but their definition is straightforward. Intuitively, a type is safe to copy if none of its constituent parts must be 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 treating `Copy` as an automatically-derived trait (autotrait).

$\boxed{\Sigma \vdash \tau_1 <: \tau_2 \rightsquigarrow \delta}$

$$\textsc{S-Refl}\ \frac{}{\Sigma \vdash \tau <: \tau \rightsquigarrow \bullet} \qquad \textsc{S-Trans}\ \frac{\Sigma \vdash \tau_1 <: \tau_2 \rightsquigarrow \delta_1 \qquad \Sigma \vdash \tau_2 <: \tau_3 \rightsquigarrow \delta_2}{\Sigma \vdash \tau_1 <: \tau_3 \rightsquigarrow \delta_1,\ \delta_2}$$

$$\textsc{S-Ref}\ \frac{\varrho_1 \subseteq \varrho_2 \qquad \omega_1 \leq \omega_2 \qquad \Sigma \vdash \tau_1 <: \tau_2 \rightsquigarrow \delta}{\Sigma \vdash \&\varrho_1\ \omega_1\ \tau_1 <: \&\varrho_2\ \omega_2\ \tau_2 \rightsquigarrow \delta} \qquad \textsc{S-RefVar}\ \frac{\omega_1 \leq \omega_2 \qquad \Sigma \vdash \tau_1 <: \tau_2 \rightsquigarrow \delta}{\Sigma \vdash \&\varrho\ \omega_1\ \tau_1 <: \&\rho\ \omega_2\ \tau_2 \rightsquigarrow \delta[\rho \mapsto \varrho]}$$

$$\textsc{S-Function}\ \frac{\begin{array}{c}\delta_a = [\overline{\rho_2 \mapsto \rho_1},\ \overline{\alpha_2 \mapsto \alpha_1}] \\ \Sigma \vdash \delta_a(\tau^{\mathrm{s}}_{12}) <: \tau^{\mathrm{s}}_{11} \rightsquigarrow \delta_1 \qquad \ldots \qquad \Sigma \vdash \delta_a(\tau^{\mathrm{s}}_{n2}) <: \tau^{\mathrm{s}}_{n1} \rightsquigarrow \delta_n \qquad \Sigma \vdash \tau^{\mathrm{s}}_{r1} <: \delta_a(\tau^{\mathrm{s}}_{r2}) \rightsquigarrow \delta_r\end{array}}{\Sigma \vdash \forall\overline{<\rho_1}, \overline{\alpha_1}>(\tau^{\mathrm{s}}_{11}, \ldots, \tau^{\mathrm{s}}_{n1}) \xrightarrow{\Gamma_1} \tau^{\mathrm{s}}_{r1} <: \forall\overline{<\rho_2}, \overline{\alpha_2}>(\tau^{\mathrm{s}}_{12}, \ldots, \tau^{\mathrm{s}}_{n2}) \xrightarrow{\Gamma_2} \tau^{\mathrm{s}}_{r2} \rightsquigarrow \delta_1, \ldots, \delta_n, \delta_r}$$

$$\textsc{S-Array}\ \frac{\Sigma \vdash \tau_1 <: \tau_2 \rightsquigarrow \delta}{\Sigma \vdash [\tau_1;\ n] <: [\tau_2;\ n] \rightsquigarrow \delta} \qquad \textsc{S-Tuple}\ \frac{\Sigma \vdash \tau_1 <: \tau'_1 \rightsquigarrow \delta_1 \qquad \ldots \qquad \Sigma \vdash \tau_n <: \tau'_n \rightsquigarrow \delta_n}{\Sigma \vdash (\tau_1, \ldots, \tau_n) <: (\tau'_1, \ldots, \tau'_n) \rightsquigarrow \delta_1, \ldots, \delta_n}$$

$$\textsc{S-Struct}\ \frac{\Sigma \vdash \tau_1 <: \tau'_1 \rightsquigarrow \delta_1 \qquad \ldots \qquad \Sigma \vdash \tau_n <: \tau'_n \rightsquigarrow \delta_n \qquad \delta_e = [\rho_1 \mapsto \varrho_1, \ldots, \rho_n \mapsto \varrho_n]}{\Sigma \vdash \mathrm{S}<\varrho_1, \ldots, \varrho_m, \tau_1, \ldots, \tau_n> <: \mathrm{S}<\rho_1, \ldots, \rho_m, \tau'_1, \ldots, \tau'_n> \rightsquigarrow \delta_1, \ldots, \delta_n, \delta_e}$$

Fig. 6. Subtyping in Oxide

than $\cap$ to draw attention to the fact that it is an intersection on their domains, but requires that the types of each place unify to a combined type (rather than already be exactly the same).

Binding. In Oxide, our T-Let rule is interesting in two ways. First, it relies crucially on the metafunction `places-typ`$(\cdot, \cdot)$, which essentially expands to all the places rooted at the given variable given the type of the binding. For example, if the variable has a struct type, the places would include all the fields of the struct, and all the places reachable from them. If the variable has a tuple type, the places would be similar except projections, rather than fields. Subsequently, rather than add the single variable x, we add *all* the places we computed to the environment when type-checking the body of the let. Secondly, T-Let relies on a substitution-yielding subtyping relation (defined in Figure 6) loosely inspired by differential subtyping from Rastogi et al. [2015].

In particular, this subtyping relation produces a mapping from provenance variables ρ in the annotated type to the provenance sets computed by our type-checker for e_1. This is motivated by a desire to allow the programmer to only write provenance variables ρ in their type annotations throughout the program. While there are a number of rules for the judgment, the majority of them are uninteresting, simply recurring in the standard way (i.e. covariantly for tuples, contravariantly for function arguments, etc.). The most interesting rule is S-RefVar where we actually extend the substitution when we find a provenance variable on the right side and a concrete provenance set on the left. In this case, we can say that the provenance variable in the annotated type stood for the provenance set we computed, and so we extend the substitution with this mapping. Then, with the substitution in hand, we apply it to e_2 to get a version with the *concrete* provenance sets that we have already computed. We then continue type-checking with the updated e_2.

Assignment. Like let bindings, assignment is also interesting in two ways. First, we require π to be safe to use `uniq`-ly to ensure that unchecked mutation does not happen in the presence of aliasing. Second, as in branching, we actually allow the type of the new expression to differ from the place's type as long as they unify. This allows us to change the provenance sets in reference types through assignment, as well as branching. After doing this, we replace the types for π and its subplaces with the version from our newly-assigned expression.

Closures and Application. The rule for application (T-App) is essentially an ordinary function application rule in environment-passing style, with the only difference being that type and provenance variables are substituted in the types since functions and closures are polymorphic. By contrast, the closure rule (T-Closure) is more interesting. As in T-Let, it expands variables and their types into places, and uses them, along with the parameterized provenance and type variables, to type check the body of the closure. However, in doing so, we also check $\Gamma \vdash \tau_1, \ldots, \tau_n$. This judgment, elided from this paper but included in the technical appendix, ensures that each of the types are mutually compatible, meaning that if any of the types contain references, it must be safe for all of them to exist simultaneously. Finally, closures are able to capture additional places from the place environment Γ. The rule computes the captured places Γ_c by taking the difference of the output environment Γ' and the input environment Γ, and places the resultant Γ_c over the function arrow. This is critical to ensure that references that have been captured in a closure are not simply forgotten about, as doing so would allow the programmer to create multiple `uniq` references to the same place by storing one in a closure. This environment is used in the ω-safety judgment, as while closures can be applied in the same way regardless of the captured environment, the environment has a different effect on which places are still safe to use and how.

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 and structs, check the types of their components while threading through the environments in left-to-right order. Like in T-Closure, these rules also require us to check that the types are mutually compatible since the loans from any borrow expressions that appear in our subexpressions will not be live until we've type checked the whole expression.

3.3 Operational Semantics

For our operational semantics, we extend the syntax of Oxide in Figure 7 with terms that only arise at runtime, specifically pointers to places, array indices, and slices, and closures packaged with their environment. Then, we define a new notion of shapes (denoted w), which we use to capture the shape of objects in our stacks σ. In particular, there is a correspondence between values and shapes, where shapes correspond to one level of an aggregate structure. Overall, the structure of shapes corresponds to the paths that can be referred to with places, and thus also the possible places that can be borrowed from. In our operational semantics, we use the metafunctions $\texttt{places-val}(\cdot, \cdot)$ and $\texttt{value}(\cdot, \cdot)$ to move between the value and shape representations. For example, $\texttt{places-val}(\texttt{x}, (5,))$ will give us two mappings — $\texttt{x}$ to $(\square,)$ and $\texttt{x.0}$ to 5 — while $\texttt{value}(\texttt{x} \mapsto (\square), \texttt{x.0} \mapsto 5, \texttt{x})$ will result in the value $(5,)$. Finally, we define stacks σ as mapping places π to shapes w.

In Figure 8, we also present a selection of our small-step operational semantics which is defined using Felleisen and Hieb [1992] style evaluation contexts over configurations of the form $(\sigma; \boxed{e})$. A number of the rules in our dynamics employ a dynamic notion of ω-safety, shown in Figure 9, that is analogous to the static notion of ω-safety described earlier in Section 3.2. The main difference is that at runtime, we know precisely which pointers are bound to a place, rather than statically where we have only approximate provenances, and as such, we need only check that the two places do not overlap *without* any quantification.

$$
\begin{array}{llll}
\text{Expressions} & e & ::= & \ldots \\
& & | & \texttt{ptr}^{\omega}\ \pi \\
& & | & \texttt{ptr}^{\omega}\ \pi[n] \\
& & | & \texttt{ptr}^{\omega}\ \pi[n_1..n_2] \\
& & | & \langle \sigma\,,\ \texttt{forall<}\overline{\rho}\,,\ \overline{\alpha}\texttt{>}\ |x_1 : \tau_1^{\mathrm{s}}\,,\ \ldots\,,\ x_n : \tau_n^{\mathrm{s}}|\ \rightarrow\ \tau_r^{\mathrm{s}}\ \{\ e\ \}\ \rangle \\
\text{Values} & v & ::= & c\ |\ (v_1\,,\ \ldots\,,\ v_n)\ |\ [v_1\,,\ \ldots\,,\ v_n] \\
& & | & \mathrm{S}\texttt{::<}\varrho_1\,,\ \ldots\,,\ \varrho_m\,,\ \tau_1^{\mathrm{s}}\ \ldots\ \tau_n^{\mathrm{s}}\texttt{>}\ \{\ x_1 : v_1\,,\ \ldots\,,\ x_k : v_k\ \} \\
& & | & \mathrm{S}\texttt{::<}\varrho_1\,,\ \ldots\,,\ \varrho_m\,,\ \tau_1^{\mathrm{s}}\ \ldots\ \tau_n^{\mathrm{s}}\texttt{>}(v_1\,,\ \ldots\,,\ v_k) \\
& & | & \texttt{ptr}^{\omega}\ \pi\ |\ \texttt{ptr}^{\omega}\ \pi[n]\ |\ \texttt{ptr}^{\omega}\ \pi[n_1..n_2] \\
& & | & \langle \sigma\,,\ \texttt{forall<}\overline{\rho}\,,\ \overline{\alpha}\texttt{>}\ |x_1 : \tau_1^{\mathrm{s}}\,,\ \ldots\,,\ x_n : \tau_n^{\mathrm{s}}|\ \rightarrow\ \tau_r^{\mathrm{s}}\ \{\ e\ \}\ \rangle \\
\text{Shapes} & w & ::= & \square\ |\ c\ |\ (\square_1\,,\ \ldots\,,\ \square_n)\ |\ [v_1\,,\ \ldots\,,\ v_n] \\
& & | & \mathrm{S}\texttt{::<}\varrho_1\,,\ \ldots\,,\ \varrho_m\,,\ \tau_1^{\mathrm{s}}\ \ldots\ \tau_n^{\mathrm{s}}\texttt{>}\ \{\ x_1 : \square_1\,,\ \ldots\,,\ x_k : \square_k\ \} \\
& & | & \mathrm{S}\texttt{::<}\varrho_1\,,\ \ldots\,,\ \varrho_m\,,\ \tau_1^{\mathrm{s}}\ \ldots\ \tau_n^{\mathrm{s}}\texttt{>}(\square_1\,,\ \ldots\,,\ \square_k) \\
& & | & \texttt{ptr}^{\omega}\ \pi\ |\ \texttt{ptr}^{\omega}\ \pi[n]\ |\ \texttt{ptr}^{\omega}\ \pi[n_1..n_2] \\
& & | & \langle \sigma\,,\ \texttt{forall<}\overline{\rho}\,,\ \overline{\alpha}\texttt{>}\ |x_1 : \tau_1^{\mathrm{s}}\,,\ \ldots\,,\ x_n : \tau_n^{\mathrm{s}}|\ \rightarrow\ \tau_r^{\mathrm{s}}\ \{\ e\ \}\ \rangle \\
\text{Stacks} & \sigma & ::= & \bullet\ |\ \sigma\,,\ \pi \mapsto w
\end{array}
$$

Fig. 7. Oxide Syntax Extensions for Dynamics

With dynamic ω-safety in hand, the evaluation rules become otherwise quite mundane: E-Move returns a value by moving it off of the stack σ. E-Copy copies the value from the stack. E-Borrow creates a pointer value to the place π. In all these cases, we employ the dynamic ω-safety check just as we employed the static equivalent in their typing rules. Branching is completely standard.

Assignment. In E-Assign, we require (as in T-Move) that the place π being updated is safe to use `uniq`-ly. This is important since assignment will overwrite the value, and without ensuring uniqueness, we could introduce dead pointers or data races. To add the updated value to our stack, we use the metafunction $\texttt{places-val}(\pi\,,\ v)$ to flatten the value into a collection of mappings from places to their respective shapes for that part of v.

Binding and the Stack. In the case of E-Let, and E-App, we are introducing new bindings, and thus, similar to assignment, use `places-val` to turn values into a collection of mappings from places to their respective shapes. In addition, since we want these bindings to be well scoped, we introduce a `pop` instruction to pop the new binding off of the stack. The semantics of this instruction are given in E-Pop, where we see that a `pop` instruction will step to unit, while removing the most recent collection of places rooted at its argument x from the stack. The use of `pop` is important to ensuring that bindings are well-scoped, and that variable shadowing works properly.

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

In this section, we present metatheoretic results for Oxide. In particular, we prove syntactic type safety for Oxide using progress and preservation [Wright and Felleisen 1992]. The proof of these lemmas are themselves fairly standard — relying on structural induction on the typing derivation in the former and on the reduction relation in the latter — but rely on an instrumented dynamic semantics (as described in Section 3.3) where we have inserted runtime checks to ensure that a newly-created `uniq` reference will indeed be unique. Fortunately, we can provably erase them as we'll see in Section 3.5. For now, we will focus on type safety.

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

E-Move
$$\frac{\sigma \vdash_{\mathsf{uniq}} \pi \qquad \mathsf{value}(\sigma,\ \pi) = v}{\Sigma \vdash (\sigma;\ \boxed{\pi}) \longrightarrow (\sigma - \pi;\ \boxed{v})}$$

E-Copy
$$\frac{\sigma \vdash_{\mathsf{shrd}} \pi \qquad \mathsf{value}(\sigma,\ \pi) = v}{\Sigma \vdash (\sigma;\ \boxed{\pi}) \longrightarrow (\sigma;\ \boxed{v})}$$

E-Borrow
$$\frac{\sigma \vdash_{\omega} \pi}{\Sigma \vdash (\sigma;\ \boxed{\&\omega\ \pi}) \longrightarrow (\sigma;\ \boxed{\mathsf{ptr}^{\omega}\ \pi})}$$

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

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

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

E-Assign
$$\frac{\sigma \vdash_{\mathsf{uniq}} \pi \qquad \mathsf{places\text{-}val}(\pi,\ v) = \overline{\pi \mapsto w}}{\Sigma \vdash (\sigma;\ \boxed{\pi := v}) \longrightarrow (\sigma - \pi,\ \overline{\pi \mapsto w};\ \boxed{()})}$$

E-Let
$$\frac{\Sigma \vdash v <: \tau \rightsquigarrow \delta \qquad \mathsf{places\text{-}val}(x,\ v) = \overline{\pi \mapsto w}}{\Sigma \vdash (\sigma;\ \boxed{\mathtt{let}\ x : \tau = v;\ e}) \longrightarrow (\sigma,\ \overline{\pi \mapsto w};\ \boxed{\delta(e);\ \mathsf{pop}\ x})}$$

E-Pop
$$\frac{}{\Sigma \vdash (\sigma;\ \boxed{\mathsf{pop}\ x}) \longrightarrow (\sigma - x;\ \boxed{()})}$$

E-App
$$\frac{\begin{array}{c} v_f = \langle \sigma_c,\ \mathtt{forall{<}}\overline{\rho},\ \overline{\alpha}\mathtt{{>}}\ |x_1 : \tau_1^{\mathsf{s}},\ \ldots,\ x_n : \tau_n^{\mathsf{s}}|\ \rightarrow\ \tau_r^{\mathsf{s}}\ \{\ e\ \}\ \rangle \\ \mathsf{places\text{-}val}(x_1,\ v_1) = \overline{\pi_1 \mapsto w_1} \qquad \ldots \qquad \mathsf{places\text{-}val}(x_n,\ v_n) = \overline{\pi_n \mapsto w_n} \end{array}}{\begin{array}{l} \Sigma \vdash (\sigma;\ \boxed{v_f ::< \overline{\varrho},\ \overline{\tau} >(v_1,\ \ldots,\ v_n)}) \longrightarrow \\ (\sigma,\ \sigma_c,\ \overline{\pi_1 \mapsto w_1},\ \ldots,\ \overline{\pi_n \mapsto w_n};\ \boxed{e[\overline{\varrho}/\overline{\rho}][\overline{\tau}/\overline{\alpha}];\ \mathsf{pop}\ \mathrm{dom}(\sigma_c),\ x_1,\ \ldots,\ x_n}) \end{array}}$$

Fig. 8. Selected Oxide Reduction Rules

$\boxed{\sigma \vdash_{\omega} \pi}$

V-DynSafe
$$\frac{}{\pi \mapsto w \vdash_{\omega} \pi}$$

V-DynUniqueSafe
$$\frac{\neg\,\mathsf{overlapping}(\pi,\ \pi') \qquad \pi_b \neq \pi \qquad \sigma \vdash_{\mathsf{uniq}} \pi}{\sigma,\ \pi_b \mapsto \mathsf{ptr}^{\omega}\ \pi' \vdash_{\mathsf{uniq}} \pi}$$

V-DynSharedSafe
$$\frac{\neg\,\mathsf{overlapping}(\pi,\ \pi') \qquad \pi_b \neq \pi \qquad \sigma \vdash_{\mathsf{shrd}} \pi}{\sigma,\ \pi_b \mapsto \mathsf{ptr}^{\mathsf{uniq}}\ \pi' \vdash_{\mathsf{shrd}} \pi}$$

V-DynSharedPassthrough
$$\frac{\sigma \vdash_{\mathsf{shrd}} \pi}{\sigma,\ \pi_b \mapsto \mathsf{ptr}^{\mathsf{shrd}}\ \pi' \vdash_{\mathsf{shrd}} \pi}$$

V-DynSafeClosure
$$\frac{\sigma,\ \sigma_c \vdash_{\omega} \pi}{\sigma,\ \pi_b\ :\ \langle \sigma_c,\ \mathtt{forall{<}}\overline{\rho},\ \overline{\alpha}\mathtt{{>}}\ |\overline{\tau^{\mathsf{s}}}|\ \rightarrow\ \tau_r^{\mathsf{s}}\ \{\ e\ \}\ \rangle \vdash_{\omega} \pi}$$

V-DynSafePassthrough
$$\frac{\begin{array}{c} w \neq \langle \sigma_c,\ \mathtt{forall{<}}\overline{\rho},\ \overline{\alpha}\mathtt{{>}}\ |\overline{\tau^{\mathsf{s}}}|\ \rightarrow\ \tau_r^{\mathsf{s}}\ \{\ e\ \}\ \rangle \\ w \neq \mathsf{ptr}^{\omega'}\ \pi' \qquad \pi_b \neq \pi \qquad \sigma \vdash_{\omega} \pi \end{array}}{\sigma,\ \pi_b \mapsto w \vdash_{\omega} \pi}$$

Fig. 9. Dynamic μ-Safety in Oxide

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

In Lemma 3.1, we say that if we can type-check e under a valid global environment Σ and have a stack σ that satisfies our place environment Γ, then either e is a value, an abort expression, or we can take a step to an updated stack σ' and expression e'. The proof proceeds by structural induction on the typing derivation for e.

Lemma 3.2 (Preservation).
If $\Sigma; \bullet; \Gamma \vdash \boxed{e} : \tau \Rightarrow \Gamma_f$ *and* $\Sigma \vdash \sigma : \Gamma$ *and* $\vdash \Sigma$ *and* $\Sigma \vdash (\sigma; \boxed{e}) \longrightarrow (\sigma'; \boxed{e'})$ *, then* $\exists \Gamma_i.$ $\Sigma; \bullet; \Gamma_i \vdash \boxed{e'} : \tau' \Rightarrow \Gamma'_f$ *and* $\Sigma \vdash \tau' <: \tau \rightsquigarrow \delta$ *and* $\Sigma \vdash \Gamma'_f <: \Gamma_f$ *and* $\Sigma \vdash \sigma' : \Gamma_i$.

Then, in Lemma 3.2, we say that if we can type-check e under a valid global environment Σ, have a stack σ that satisfies our place environment Γ and can take a step to an updated configuration $(\sigma'; \boxed{e'})$, then there exists an intermediate place environment Γ_i which our updated stack σ' satisfies and under which our updated expression e type-checks with a potentially more-specific type τ' and a potentially more-specific output environment Γ'_f. In this case, the proof proceeds by structural induction on the reduction relation for $(\sigma; \boxed{e})$.

With Lemma 3.1 and Lemma 3.2 in hand, we are then able to prove a type safety theorem in Theorem 3.3 by interleaving the usage of progress and preservation. For the interested reader, the full proofs of all of these theorems are included in our technical appendix.

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

3.5 Instrumentation Erasure

As alluded to in Section 3.4, our metatheory includes evidence that we can eliminate the instrumentation used in proving type safety from our semantics. In particular, Lemma 3.4 tells us that as long as we maintain $\Sigma \vdash \sigma : \Gamma$ throughout, any static ω-safety check we make will ensure that the corresponding dynamic ω-safety check will succeed. Lemma 3.2 then tells us that $\Sigma \vdash \sigma : \Gamma$ is preserved by reduction. Then, by inspecting our typing and reduction rules, we can see that each dynamic check always corresponds exactly to a static check in the type system. Thus, we can use Lemma 3.4 to conclude that we do not need to actually run any of the dynamic checks.

Lemma 3.4 (Static ω-Safety Implies Dynamic ω-Safety).
If $\Sigma \vdash \sigma : \Gamma$ *and* $\Gamma \vdash_\omega \pi : \tau$ *, then* $\sigma \vdash_\omega \pi$.

Additionally, we know by inspecting the rules of our operational semantics that the ω annotations on references and pointer values are *only* used in the dynamic ω-safety checks. Since our lemma has shown us that these checks will always succeed in a well-typed program, we know we can erase them, and thus, we also know that we can erase these now-unnecessary annotations.

4 (IRON) OXIDE IS RUST

In this section, we will work through a number of example programs in Rust, and their corresponding form in Oxide to explore how we are able to model interesting parts of the borrow checker. Our aim in doing so is to explore how effective Oxide is as a model for source-level Rust. Then, in Section 4.3, we will go further and 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 = ("Hello",);
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  let msg: &'msg shrd String = {
3    // Γ1 = Γ0
4    let m: (String,) = ("Hello",);
5    // Γ2 = Γ1, m : (String), m.0 : String
6    &shrd m.0 // ERROR. τ = &{ shrd m.0 } shrd String
7    // Γ3 = Γ2 − m = Γ1
8  };
9  msg
```

To translate to Oxide, we again made the usual set of changes — annotating bindings with types, and adding explicit `shrd` qualifiers, but to aid comprehension, we also added comments that describe the state of the place environment Γ while type-checking the program. Like the Rust version, the Oxide version also 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 place environment at the end of the binding (seen on line 7). Further, note that the type we derive for `&shrd` $m.0$ is &{ $^{\texttt{shrd}}m$ } `shrd String`. Then, since type well-formedness requires that the places present in the approximate provenances of every type must be bound in the environment, we are unable to derive a type for the let binding for `msg`. That is — •; •; $\Gamma_3 \vdash \tau$ 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 Section 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 place environment Γ while type-checking the program):

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

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 5, according to T-Branch, we check the type of each side of the branch under the same place environment Γ (visible in the annotations on lines 6 and 14). Since they have the same input place environments, they are each able to create their own unique reference to parts of `pt` (lines 7 and 15). Then, the bindings to `x` and `p` both end at the end of their respective branch before returning unit (lines 12 and 20). This means that when we intersect the *output* place environments of each branch on line 21, we get exactly their input environment Γ, 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:

```
1  let mut m: u32 = 6;
2  let mut n: u32 = 5;
3  let x: &u32 = &n;
4  if false {
5    x = &m;
6  }
7  &mut m; // ERROR: cannot borrow m mutably while already borrowed
8  ... // 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 place environments Γ):

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

Now, using the Oxide version of the example, we can explain more formally why the program fails to type-check. On line 6, when we borrow from `n`, we produce a reference of the type `&{ shrd n } shrd u32` and add it to our environment 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 10). According to T-Assign, this will cause us to replace the type of `x` with the type of the new reference: `&{ shrd m } shrd u32`, but in the other side of the branch, we don't change `x` and so its type remains the same (lines 13 and 17 respectively). When we exit the branch, in T-Branch, we will combine the two output place environments from each side using a variant of intersection that employs *type unification* (as seen in Section 3.2) to

combine the types for every place π in both place environments (line 18). The result is that after the branch, `x` is given the type `&{ shrd m , shrd n } shrd u32` in the place environment. Thus, when we attempt to derive `uniq`-safety in T-Borrow for the borrow expression on line 19, we find an overlapping shared loan against `m` in the approximate provenance of `x` and yield an error.

4.3 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 approximate 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 a dual view: when a reference is dropped from the place environment π, all the loans in its approximate provenance are ended. Though we have not formally explored a connection between the two, 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. Additionally, unlike Oxide, Patina provided a more direct memory model which is problematic as Rust itself does not yet have a well-defined memory model.

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, but fortunately, our goals are actually 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.

KRust and K-Rust. There have been two efforts [Kan et al. 2018; Wang et al. 2018] after RustBelt to develop an executable formal semantics for Rust in the K Framework [Rosu and Şerbănuţă 2010]. Despite the unfortunately similar names, the two efforts, K-Rust and KRust, are distinct. Neither project comes with an appropriate treatment of ownership in the type systems, and they unfortunately did not describe any metatheory (such as soundness) for their semantics.

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 sort of design space, finding some balance of 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 with a judgment to determine 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. Additionally, Tov and Pucella presented a core calculus ${}^a\lambda_{ms}$ that formalized the essence of Alms. We view this effort as analogous to our effort to develop Oxide as a formalization of core Rust.

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 Section 2.2, the meaning of regions in Rust (and Oxide) is different. Approximate 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.

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 (though, we will return to discuss **`unsafe`** further in Section 6.2). In many other languages, key data structures like hash maps are implemented as built-in types within the

$$
\begin{array}{lrcl}
\text{Types} & \tau & ::= & \ldots \\
 & & | & \texttt{Vec<}\tau\texttt{>} \\
 & & & \\
\text{Expressions} & e & ::= & \ldots \\
 & & | & \texttt{Vec::<}\tau\texttt{>::new()} \\
 & & | & e_1\texttt{.push(}e_2\texttt{)} \mid e\texttt{.pop()} \mid e_1\texttt{.swap(}e_2, e_3\texttt{)} \\
 & & | & e\texttt{.len()}
\end{array}
$$

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

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

$$\textsc{T-VecPush}\quad \frac{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e_1} : \&\varrho\ \texttt{uniq Vec<}\tau\texttt{>} \Rightarrow \Gamma_1 \qquad \Sigma;\ \Delta;\ \Gamma_1 \vdash \boxed{e_2} : \tau \Rightarrow \Gamma_2}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e_1\texttt{.push(}e_2\texttt{)}} : \texttt{unit} \Rightarrow \Gamma_2}$$

$$\textsc{T-VecPop}\quad \frac{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e} : \&\varrho\ \texttt{uniq Vec<}\tau\texttt{>} \Rightarrow \Gamma'}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e\texttt{.pop()}} : \tau \Rightarrow \Gamma'} \qquad \textsc{T-VecLen}\quad \frac{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e} : \&\varrho\ \texttt{shrd Vec<}\tau\texttt{>} \Rightarrow \Gamma'}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e\texttt{.len()}} : \texttt{u32} \Rightarrow \Gamma'}$$

$$\textsc{T-VecSwap}\quad \frac{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e_1} : \&\varrho\ \texttt{uniq Vec<}\tau\texttt{>} \Rightarrow \Gamma_1 \qquad \Sigma;\ \Delta;\ \Gamma_1 \vdash \boxed{e_2} : \tau \Rightarrow \Gamma_2 \qquad \Sigma;\ \Delta;\ \Gamma_2 \vdash \boxed{e_3} : \tau \Rightarrow \Gamma_3}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{e_1\texttt{.swap(}e_2, e_3\texttt{)}} : \texttt{unit} \Rightarrow \Gamma_3}$$

Fig. 10. Extending Oxide with Vectors

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 already 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. Toward 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 10, which adds support for `Vec` to the language. 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 $\texttt{Vec<}\tau\texttt{>}$. Then, we extend the grammar of expressions with some of the key operations on vectors. $\texttt{Vec::<}\tau\texttt{>::new()}$ is used to create a new empty vector with the element type τ. Then, $e_1\texttt{.push(}e_2\texttt{)}$ and $e\texttt{.pop()}$ are used to add and remove elements from the vector, while $e_1\texttt{.swap(}e_2, e_3\texttt{)}$ is used to swap the values in the vector at indices e_2 and e_3. Finally, of course, $e\texttt{.len()}$ 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 Unsafe and Dynamic Safety Enforcement

As we saw in Section 3.3, our operational semantics contains a runtime check ($\sigma \vdash_\omega \pi$) which is used to check that a particular use of a place is ω-safe at runtime. Within the scope of Oxide (i.e. *safe* Rust), these runtime checks exist to cause programs which would violate the language's guarantees about uniqueness of `uniq`-references to become stuck in the operationl semantics. So, at some level, this can be thought of as a proof aide that allows us to say, through syntactic type safety, that we maintain this guarantee. However, this runtime check appears to have an added use when considering an Oxide extension for raw pointers, the essential feature of **`unsafe`** in Rust.

For our purposes, the essential difference between a reference and a raw pointer is that the latter is not subject to Oxide's ownership restrictions (in the form of the static ω-safety judgment). However, we could imagine taking advantage of exactly these dynamic checks when converting (via `mem::transmute`) from a raw pointer to a reference. This would allow us to maintain soundness in the presence of some kinds of **`unsafe`** code by producing runtime errors when we would otherwise have violated our static guarantees. It is possible that similar efforts would allow us to enforce more broadly that all **`unsafe`** code in a program is extensionally safe, perhaps relying on techniques like those used in Valgrind [Nethercote and Seward 2007]. We view such an effort as a potential contribution to the ongoing conversation about unsafe code guidelines for Rust [2019].

6.3 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:

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

Without two-phase borrows, this example behaves like one of our early examples in Section 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. Of course, we could try to resolve the problem by changing the desugaring. For example, we could try to desugar to the following instead:

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

Unfortunately, the desugaring is subtle and getting it to work out properly in all cases is difficult. Still, the *idea* behind this reordering suggests a possible weakening of the type system to make the first expansion equivalent to the second in the eyes of the borrow checker. That weakening is precisely two-phase borrows. The idea is that when a mutable loan is first introduced, it is marked as *reserved*. Mutable loans marked as reserved act as if they are shared *until* they reach a point in the program where they must be uniquely (e.g. because the program attempts to use it in assignment).

While Oxide does not currently support two-phase borrows, we could imagine extending our grammar for ownership quantifiers ω with a new form `rsvd`, 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 ω-safety judgment ($\Gamma \vdash_{\omega} \pi : \tau$) to indicate which loans need to be changed in Γ, and thus would complicate our type system.

6.4 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.

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].

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].

6.5 Mechanized Metatheory for Oxide

Though we have paper proofs in our technical appendix for all the theorems presented here, we have begun an effort to mechanize the semantics in Coq. This has a number of advantages in this case. 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 be able to know that their changes have not broken the Oxide's formal guarantees.

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 Oxide sound using syntactic techniques with an instrumented semantics (Section 3.4), and demonstrated that it is safe to erase this instrumentation (Section 3.5). 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, Matthew Fluet, and Greg Morrisett. 2005. A Step-Indexed Model of Substructural State. In *International Conference on Functional Programming (ICFP), Tallinn, Estonia.*

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.*

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), Vancouver, British Columbia.*

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.

J. Roger Hindley. 1969. The Principal Type-Scheme of an Object in Combinatory Logic. *Trans. Amer. Math. Soc.* (1969).

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, 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.*

Shuanglong Kan, David Sanán, Shang-Wei Lin, and Yang Liu. 2018. K-Rust: An Executable Formal Semantics for Rust. *CoRR* abs/1804.07608 (2018). arXiv:1804.07608 http://arxiv.org/abs/1804.07608

Steve Klabnik. 2015. The Language Strangeness Budget. https://words.steveklabnik.com/the-language-strangeness-budget. Accessed: 2019-02-28.

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

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/. Accessed: 2019-02-17.

Nicholas D. Matsakis and Contributors. 2018. Type Inference - `rustc` Guide. https://rust-lang.github.io/rustc-guide/type-inference.html. Accessed: 2019-02-28.

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

Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), San Diego, California*.

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*.

Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. 2015. Safe & Efficient Gradual Typing for TypeScript. In *ACM Symposium on Principles of Programming Languages (POPL), Mumbai, India*.

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*.

Grigore Rosu and Traian Florin Şerbănuţă. 2010. An Overview of the K Semantic Framework. *Journal of Logic and Algebraic Programming* (2010).

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.

Aaron Turon and Michael Hicks. 2015. Interview with Mozilla's Aaron Turon. http://www.pl-enthusiast.net/2015/06/09/interview-with-mozillas-aaron-turon/. 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)*.

Feng Wang, Fu Song, Min Zhang, Xiaoran Zhu, and Jun Zhang. 2018. KRust: A Formal Executable Semantics of Rust. *CoRR* abs/1804.10806 (2018). arXiv:1804.10806 http://arxiv.org/abs/1804.10806

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).