Oxide: The Essence of Rust

POPL 2019 *Student Research Competition* Extended Abstract

Aaron Weiss
Northeastern University
weiss@ccs.neu.edu

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 safer and quicker to produce. To accomplish this, Rust integrates decades of programming languages research into a production system, in particular, linear and ownership types [2, 4, 7, 9] and region-based memory management [3, 5]. Yet, Rust goes beyond much of this 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. To that end, we are designing a formal semantics called Oxide to capture *the essence of Rust*.

While there are some existing formalizations [6, 8, 10], none capture a high-level understanding of Rust's essence (namely *ownership* and *borrowing*). Patina [8], the first major effort, formalized an early version of Rust which predates much of the work to simplify and streamline the language. RustBelt [6], the most complete effort to date, formalized a *low-level*, intermediate language in continuation-passing style, which makes it difficult to reason about ownership as a *source-level* concept. Finally, an early version of Oxide [10] oversimplified some parts of the language and overcomplicated others. We describe the changes from this early version in more detail later.

In this work, we present key pieces of the latest version of Oxide, and in particular, our core language Oxide_0 which omits **`unsafe`**-implemented abstractions from the standard library. In future work, we will describe further extensions Oxide_1, Oxide_2, and so forth that add essential abstractions from the standard library as described in our prior work [10].

2 OWNERSHIP AND BORROWING

The essence of Rust lies in its novel approach to *ownership* and *borrowing*, which we explain intuitively before diving into their formal presentation. To start, we consider Fig. 1.

```
1 struct Point(u32, u32);
2
3 let p: Point = Point(3, 2);
4 let pt: Point = p;
5 // At this point, we can no longer refer to p.
6 let x: &'x u32 = &'a imm pt.0;
7 // At this point, we can no longer mutate pt.
8 let y: &'y u32 = &'b imm pt.1;
```

Figure 1: A little example program in Oxide_0.

First, on line 1, we declare a type `Point` that consists of a pair of unsigned 32-bit integers (`u32`). Then, on line 3, we create a new `Point` bound to `p`. In Rust, each value is *owned* by the identifier to which it's bound. So, we can then say that `p` owns the value `Point(3, 2)`. Then, on line 4, we *move* the value from `p` to the new identifer `pt`. This is the first operation that affects ownership. After moving the value out of `p`, we invalidate the old name and thus further attempts to use it will result in an error. Then, on line 6, we create a reference *borrowing* from the place `pt.0` — the first projection of `pt`. This borrowing operation also affects ownership, but in a more subtle way as we explain below.

There are two distinct kinds of borrows — immutable and mutable — though it might be more apt to refer to them as shared and unique respectively. During an immutable borrow (like the one on line 6), the value can actually be *shared* between other references, but only as long as they are *all* immutable. During a mutable borrow, the new reference must be *unique* which means that the old identifer is temporarily invalidated while this reference is alive. This mutual exclusivity between mutation and sharing is how Rust can rule out the possibility of data races in concurrent programs, and is the site of the aforementioned subtlety.

Formally, we model ownership and borrowing with linear capabilities [3] and fractional permissions [1]. We've developed a novel type-and-effect system that automatically tracks fractional capabilities representing ownership by recording and applying effects that modify these capabilities. This system takes the form of the judgment:

$$\Sigma; \Delta; \Gamma; \mathscr{L} \vdash e : \tau \Rightarrow \varepsilon$$

In this judgment, we have four environments. Our first environment Σ (called the global environment) records top-level definitions of functions and types like the one we saw on line 1 of our example (**`struct Point(u32, u32)`**). Our second environment Δ (called the type variable environment) records in-scope type variables and their respective kinds κ

$$\begin{array}{llll} \text{Places} & \pi & ::= & x \mid *\pi \\ & & \mid & \pi.x \mid \pi.n \\ & & \mid & \pi[n] \\ & & \mid & \pi[n_1..n_2] \end{array}$$

Figure 2: The grammar for places in Oxide.

— since we have both type variables α and region variables ρ. Our third environment Γ (called the variable environment) tracks in-scope places π (described below) and their types τ with the fractional capability f guarding their use. Entries in Γ are written $\pi :_f \tau$. Our final environment $\mathscr{L}$ (called the loan environment) tracks valid loans ℓ with a fractional capability f and the place π of their provenance. These loans ℓ (like `'a` and `'b` in Fig 1) give names to specific borrow sites in the program that are used in our types to track the possible provenance of references.

Places. Places are expressions that represent a location in memory. In Oxide, we use these expressions themselves to capture the shape of memory in order to keep our memory model abstract and easier to reason about. In Fig 2, we have the grammar of places including identifiers, dereferencing, projection, indexing, and slicing. Places appear in the syntax of expressions that affect ownership like *moves* and *borrows*.

Transferring Ownership. In T-Move (Fig. 3), we give a static semantics to places π as expressions which dynamically *move* values out of the places in memory which *own* them. As the earlier example (Fig. 1) captured, this is only safe if π has a whole capability (written 1) associated with it — since otherwise, moving the value would invalidate some existing references. Further, T-Move must have the effect of `drop`ping π from the environments to prevent further use.

Borrowing Ownership. In T-BorrowImm (Fig. 3), we give a static semantics to immutable borrows which require that the place π being borrowed from has a non-zero capability associated with it and yields a `borrow` effect that, when applied, creates a new entry in the loan environment $\mathscr{L}$ with the given data. Additionally, the type we produce records a singleton loan set $\{\ \ell\ \}$ specifying that the reference came from that particular borrow site. T-BorrowMut (Fig. 3) is analogous, but requires a whole capability, rather than simply a non-zero one.

Ownership and Branching. Branching in programs plays a central role in tracking ownership since it introduces a point where precise aliasing information is *lost*. T-Branch (Fig. 3) captures how we handle this loss of precision. In particular, we typecheck each side of the branch in environments Γ and $\mathscr{L}$ after applying the effect ε_1, and then unify their types to get a combined type τ (denoted $\tau_1 \sim \tau_2 \Rightarrow \tau_3$).

T-Move
$$\frac{\Gamma \vdash \pi :_1 \tau}{\Sigma; \Delta; \Gamma; \mathscr{L} \vdash \pi : \tau \Rightarrow \texttt{drop}\ \pi}$$

T-BorrowImm
$$\frac{\ell \notin \mathscr{L} \qquad \Gamma \vdash \pi :_f \tau \qquad f \neq 0}{\Sigma; \Delta; \Gamma; \mathscr{L} \vdash \&\ell\ \texttt{imm}\ \pi : \&\{\ \ell\ \}\ \texttt{imm}\ \tau \Rightarrow \texttt{borrow}\ \texttt{imm}\ \pi\ \texttt{as}\ \ell}$$

T-BorrowMut
$$\frac{\ell \notin \mathscr{L} \qquad \Gamma \vdash \pi :_1 \tau}{\Sigma; \Delta; \Gamma; \mathscr{L} \vdash \&\ell\ \texttt{mut}\ \pi : \&\{\ \ell\ \}\ \texttt{mut}\ \tau \Rightarrow \texttt{borrow}\ \texttt{mut}\ \pi\ \texttt{as}\ \ell}$$

T-Branch
$$\frac{\begin{array}{c} \Sigma; \Delta; \Gamma; \mathscr{L} \vdash e_1 : \texttt{bool} \Rightarrow \varepsilon_1 \\ \Sigma; \Delta; \varepsilon_1(\Gamma; \mathscr{L}) \vdash e_2 : \tau_2 \Rightarrow \varepsilon_2 \\ \Sigma; \Delta; \varepsilon_1(\Gamma; \mathscr{L}) \vdash e_3 : \tau_3 \Rightarrow \varepsilon_3 \\ \tau_2 \sim \tau_3 \Rightarrow \tau \end{array}}{\Sigma; \Delta; \Gamma; \mathscr{L} \vdash \texttt{if}\ e_1\ \{\ e_2\ \}\ \texttt{else}\ \{\ e_3\ \}\ : \tau \Rightarrow \varepsilon_1, \texttt{merge}(\varepsilon_2,\ \varepsilon_3)}$$

Figure 3: A selection of important typing rules in Oxide.

In this unification, we union the sets of loans that appear in the reference types (introduced by T-BorrowImm and T-BorrowMut) which tell us that values at that type could come from any of the loans in the set.

3 COMPARISON TO EARLIER OXIDE

Compared to our earlier version [10], the primary difference is that we now model *moves* and *mutable borrows* as separate operations, rather than modelling the former as the latter. In doing so, we recognize that the distinction between a *move* and a *mutable borrow* is essential to Rust's semantics. We then simplified Oxide by removing the existence of `alloc` as a syntactic form (which was never present in Rust). The result is a semantics more faithful to Rust.

4 METATHEORY

We prove type safety for Oxide_0 using progress and preservation [11]. The proofs of both lemmas are standard, but rely on an instrumented dynamic semantics where we maintain the loan environment $\mathscr{L}$ at runtime. It is straightforward to define a subsequent erasure translation that removes this instrumentation.

Lemma (Progress). *If* $\Sigma; \Delta; \Gamma; \mathscr{L} \vdash e : \tau \Rightarrow \varepsilon$ *and* $\Gamma \vdash \mathscr{L}$ *and* $\Sigma; \Gamma; \mathscr{L} \vdash \sigma$ *, then either* e *is a value or* $\exists\, \sigma', \mathscr{L}', e'.\ (\sigma;\ \mathscr{L};\ e) \rightarrow (\sigma';\ \mathscr{L}';\ e')$.

Lemma (Preservation). *If* $\Sigma; \Delta; \Gamma; \mathscr{L} \vdash e : \tau \Rightarrow \varepsilon$ *and* $\Gamma \vdash \mathscr{L}$ *and* $\Sigma; \Gamma; \mathscr{L} \vdash \sigma$ *and* $(\sigma;\ \mathscr{L};\ e) \rightarrow (\sigma';\ \mathscr{L}_1;\ e')$ *then* $\exists\, \varepsilon_1, \varepsilon_2.\ \varepsilon_1, \varepsilon_2 \leq \varepsilon\ \wedge\ \exists\, \Gamma',\ \mathscr{L}'.\ \varepsilon_1(\Gamma'; \mathscr{L}') = \Gamma_1; \mathscr{L}_1\ \wedge\ \Sigma; \Delta; \Gamma_1; \mathscr{L}_1 \vdash e' : \tau \Rightarrow \varepsilon_2\ \wedge\ \Gamma_1 \vdash \mathscr{L}_1\ \wedge\ \Sigma; \Gamma_1; \mathscr{L}_1 \vdash \sigma'$.

ACKNOWLEDGMENTS

This work has been done in collaboration with Amal Ahmed, Niko Matsakis, and Daniel Patterson. However, the material and presentation here is the work of Aaron Weiss. This material is based upon work supported in part by the National Science Foundation under grants CCF-1453796 and CCF-1618732, and an NSF Graduate Research Fellowship (GRFP). It was also supported in part by the European Research Council under ERC Starting Grant SECOMP (715753).

REFERENCES

[1] John Boyland. 2003. Checking Interference with Fractional Permissions. *International Static Analysis Symposium* (2003).

[2] 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*.

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

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

[5] 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*.

[6] 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*.

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

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

[9] Philip Wadler. 1990. Linear types can change the world! *Programming Concepts and Methods* (1990).

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

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