Rust Distilled: An Expressive Tower of Languages

AARON WEISS, Northeastern University and Inria Paris
DANIEL PATTERSON, Northeastern University
AMAL AHMED, Northeastern University and Inria Paris

Rust represents a major advancement in production programming languages because of its success in bridging the gap between *high-level* application programming and *low-level* systems programming. At the heart of its design lies a novel approach to *ownership* that remains highly programmable.

In this talk, we will describe our ongoing work on designing a formal semantics for Rust that captures ownership and borrowing without the details of lifetime analysis. This semantics models a high-level understanding of ownership and as a result is close to source-level Rust (but with full type annotations) which differs from the recent RustBelt effort that essentially models `MIR`, a CPS-style IR used in the Rust compiler. Further, while RustBelt aims to verify the safety of **`unsafe`** code in Rust's standard library, we model standard library APIs as primitives, which is sufficient to reason about their behavior. This yields a simpler model of Rust and its type system that we think researchers will find easier to use as a starting point for investigating Rust extensions. Unlike RustBelt, we aim to prove type soundness using *progress and preservation* instead of a Kripke logical relation. Finally, our semantics is a family of languages of increasing *expressive power*, where subsequent levels have features that are impossible to define in previous levels. Following Felleisen, expressive power is defined in terms of *observational equivalence*. Separating the language into different levels of expressive power should provide a framework for future work on Rust verification and compiler optimization.

1 INTRODUCTION

Programming languages have long been divided between "systems" languages, which enable low-level reasoning that has proven critical in writing systems software, and "high-level" languages, which empower programmers with high-level abstractions to write software more quickly and more safely. For many language researchers then, a natural goal has been to try to enable both low-level reasoning and high-level abstractions in one language. To date, the Rust programming language has been the most successful endeavour toward such a goal.

Nevertheless, Rust has also developed something of a reputation for its complexity amongst programmers. It would seem almost every new Rust programmer has their own tale of *fighting the borrow checker* with its own mess of unfamiliar type errors and associated stress. A natural question to wonder then is if this reality is inevitable. We argue it is not! The challenge of learning Rust is a familiar one—namely, learning new semantics is *hard*. While analogies by syntax make some aspects of Rust more comfortable to imperative programmers, one cannot escape having to understand the novel semantics of ownership in Rust, and for new programmers, it is tempting to get caught up in the details of lifetime inference and analysis. While these details are important for building an *efficient* analysis, we feel they are inappropriate for building a high-level mental model of the *meaning* of ownership, and hope that intuitions gleaned from our semantics can help. Of course, we don't anticipate that beginner programmers will work through the semantics directly. Instead, we believe that semanticists (in this case, ourselves) have a secondary role as teachers—to distill semantic intuitions into simple, clear explanations.

While there are some existing formalizations of Rust, we believe that none of them are sufficient for our goals of (1) understanding ownership as a seasoned Rust programmer does and (2) reasoning about how abstractions that rely on **`unsafe`** code—such as those provided by the standard library—affect the language's expressivity. The first major effort came in the form of Patina [16], a formalization of an early version of Rust with partial proofs of progress and preservation. More recently, the most well-known and complete effort in formalizing Rust is RustBelt [11] whose λ_{Rust}

has already proven useful in verifying that major pieces of `unsafe` code in the standard library do not violate Rust's safety guarantees. Nevertheless, the low-level nature of λ_{Rust} as a language in continuation-passing style makes it harder to use for source-level reasoning. Also, RustBelt's goal of *verifying* the `unsafe` code in Rust's standard library means that λ_{Rust} has a much more complex type system and lifetime logic than is necessary for *understanding* ownership and borrowing.

2 FORMALIZING RUST

In our talk, we will describe work in progress on developing `Oxide`, a formal semantics that aims to capture the essence of Rust with inspiration from linear capabilities [8] and fractional permissions [3]. To understand the core principles of how we model ownership and borrowing in our semantics, it is helpful to look at a simple example in Rust with its corresponding form in `Oxide`. This example declares a binding, and then *immutably borrows* it.

```
let x = 5;
let y = &x;
```

In `Oxide`, our code remains largely the same, but we make stack allocation explicit via the `alloc` operator, and insert the usage of `drop` that Rust would ordinarily infer. We also include annotations naming the *regions* that are being created (when we `alloc` or `borrow`) and destroyed (when we `drop`). To aid in comprehension, we also include comments describing the state of an important static context as it changes during type checking.

```
1  // P = {}
2  let imm x = alloc 'x 5;
3  // P = { 'x ↦ (u32, 1, {}) }
4  let imm y = borrow imm 'y x;
5  // P = { 'x ↦ (u32, 1/2, {}), 'y ↦ (u32, 1/2, { ϵ ↦ 'x }) }
6  drop 'y;
7  // P = { 'x ↦ (u32, 1, {}) }
8  drop 'x;
9  // P = {}
```

In particular, these comments describe the state of our region context (denoted P) after type checking each expression. This context contains a mapping from region names `'r` to a triple of the region's type, its fractional capability, and some additional metadata. We can see on line 3 that when allocating a new region `'x` for a numeric constant, we associate it with its type `u32` (an unsigned 32-bit integer), a whole capability (denoted **1**), and no additional metadata. Then, when we borrow immutably from `x` on line 4, we create a new region `'y` that takes half of the capability and records that it is aliased from the region `'x`. This metadata about aliasing is then used on line 6 to return the half-capability to `'x` when we drop `'y`. This sort of automatic management is a departure from typical presentations of linear capabilities—where they are instead first-class values which are threaded manually through the program—but more closely resembles the programming style of Rust. Finally, note that dropping `'x` on line 8 corresponds to different operational behavior than dropping `'y` on line 6. Since we have a full capability for `'x` on line 8 and since there is no metadata indicating that we must return the capability to some other region, operationally this situation corresponds to freeing the data on the stack.

It is also important to note the departures from Rust in the wild. Specifically, to have a capability guard the use of each value, it must be associated with a region (since capabilities are always tied to regions). Thus, in `Oxide`, all values are used under references. One view of this model is that the mandatory reference makes explicit the notion that the value is placed somewhere on the stack. Further, this decision enables us to simplify our model by treating *moves* as *mutable borrows* since both require full ownership represented by a whole capability.

In the rest of this section, we discuss our plans for formalizing Oxide as a tower of languages and then give more detail about our current model.

2.1 A Tower of Languages

Though we introduced it as a single language, Oxide is actually a *family* of languages that capture increasing levels of expressive power [7]. The language we've already seen above represents "safe Rust" without any features from the standard library—we call this $\texttt{Oxide}_0$. Subsequent language levels $\texttt{Oxide}_{n+1}$ are achieved by extending each language $\texttt{Oxide}_n$ with abstractions (functionality) implemented using **unsafe** code. We move up a language level, saying that $\texttt{Oxide}_{n+1}$ is more expressive than $\texttt{Oxide}_n$, when there exist observationally equivalent programs in $\texttt{Oxide}_n$, that are *not* observationally equivalent in $\texttt{Oxide}_{n+1}$.[1] We say $\texttt{Oxide}_{n+1}$ is "more expressive" than $\texttt{Oxide}_n$ since $\texttt{Oxide}_{n+1}$ has contexts with greater power that allows them to tell apart programs that cannot be distinguished by contexts in $\texttt{Oxide}_n$.

This model of Rust as a family of languages at different levels of expressive power gives us a way of precisely talking about what code refactoring, compiler optimization, and program reasoning is justified given our codebase and assumptions about the language level of code we link with. In particular, we can say for real Rust code—which might contain some **unsafe** blocks—precisely what **unsafe** abstractions have been considered, giving us a way to reason about observational equivalence of Rust programs.

Allocation on the Heap. In $\texttt{Oxide}_1$, we extend $\texttt{Oxide}_0$ with Vec<T> which increases the expressivity of our language by giving us access to the heap. Readers familiar with Rust might note that Box<T> is typically thought of as the "heap-allocated type", but we chose Vec<T> because it is more general (a Box is a Vec of length 1). Further, in principle, Vec alone is sufficient to write interfaces observationally equivalent to data structures from std::collections like HashMap, BTreeMap, and BinaryHeap—assuming, as is typical, that performance is not included in our notion of observation.

Shared Memory with Rc. For $\texttt{Oxide}_2$, we include Rc<T> which provides reference-counted pointers. Like immutable references, these pointers can be used to share memory between different parts of the program, but unlike immutable references, the information is tracked *dynamically*. This enables programs to recover mutable references at runtime when they know that there are no additional aliases. It is this ability to recover mutable references that raises the language's expressive power.

RefCells for Interior Mutability. In $\texttt{Oxide}_3$, we include RefCell<T> which provides a way for shared data to be mutated. This capability is known in the Rust community as *interior mutability* because it is often used to hide manipulations of internal state to ultimately present an immutable interface. Like with Rc, RefCell works by deferring the necessary safety checks around mutation to runtime. Though not restricted to the heap, RefCell is analogous to ref in the ML tradition.

Growing Further. Our current goal is to formalize $\texttt{Oxide}_0$ through $\texttt{Oxide}_3$. In the future, we could extend our family of languages further, adding the ability to spawn threads ($\texttt{Oxide}_4$), communicate between them ($\texttt{Oxide}_5$), and so on. Like the early levels, these extensions add further complexity and increase language expressivity.

2.2 A Further Look at $\texttt{Oxide}_0$

With a high-level understanding of Oxide in place, we can now take a closer look at the core language, $\texttt{Oxide}_0$. It includes allocation on the stack (denoted $\texttt{alloc}_\rho\ e$), copying ($\texttt{copy}_\rho\ e$) and

[1] Observational equivalence for each level, $\texttt{Oxide}_n$, is defined as the standard notion of contextual equivalence for that language.

borrowing mutably ($\texttt{borrow}_\rho$ `mut` x) and immutably ($\texttt{borrow}_\rho$ `imm` x), all of which create a fresh region bound to ρ. Note that in our formal syntax, we write ρ for `'r` seen in the earlier example and use μ to collectively refer to mutability quantifiers `mut` and `imm`. $\texttt{Oxide}_0$ also features let bindings, assignment, branching, and pattern matching. However, pattern matching is restricted to allow only simple patterns—i.e. those without nesting and **`ref`** patterns. $\texttt{Oxide}_0$ also includes structs, tuples, enumerations, and fixed-sized arrays with borrowing inside each data structure ($\texttt{borrow}_\rho\ \mu\ x.\pi$). Here, π denotes the path through the data structure, e.g. borrowing the first field of a tuple x is written $\texttt{borrow}_\rho\ \mu\ x.0$. Finally, $\texttt{Oxide}_0$ requires all drops to be explicit (denoted `drop` ρ). Hence, with different strategies for placing these drop expressions when we compile—in essence, *elaborate*—from Rust to `Oxide`, we can model Rust both with and without the upcoming non-lexical lifetimes feature.

Type system. We make use of a type-and-effect system where effects keep track of changes to the region context (written P) caused by the given expression. This takes the shape of a judgment of the form $\Sigma;\ \Delta;\ P;\ \Gamma \vdash e : \tau \Rightarrow \varepsilon$, read "in the global context Σ (which contains structure definitions), type variable context Δ, region context P, and variable context Γ, e has type τ with effect ε." Our available effects include creating new regions (`newrgn` $(\tau, f, \mathcal{M})$ `as` ρ where τ is the type of the value stored in the new region ρ, f denotes the fractional capability for the region, and $\mathcal{M}$ denotes the metadata seen in the example), borrowing from one region into a new region (`borrow` $\mu\ \rho_1$ `as` ρ_2), deleting a region (`delrgn` ρ), and updating (sub)regions during assignment (`update` $\rho_1.x$ `to` ρ_2).

In order to prove type soundness using progress and preservation, we rely on the usual trick of using an *instrumented semantics*. In our model, the instrumented semantics maintains information about regions in order to track aliasing/borrowing relationships in memory during runtime (instead of maintaining a more traditional flat memory structure that only maps locations to values). Intuitively, the way to think about this is that, upon allocation (or copy), we create a new region that essentially corresponds to what would normally be a physical location, but when we borrow, we create fresh regions akin to ghost locations that exist for the purpose of keeping track of the aliasing that results from borrowing. Nonetheless, both the physical and the ghost locations are modeled as regions `'r`.

Concretely, our operational semantics is a relation on machine configurations $(\sigma,\ \mathcal{R},\ e)$ where σ is a store that maps variables to regions, $\mathcal{R}$ is a region store that maps regions to the values stored there—similar to the static environment P discussed earlier but storing values instead of types—and e is the expression being evaluated. The instrumented semantics contains enough information to enable a proof via progress and preservation (though the proof is still in progress). In the future, we will provide an erasure procedure that, in essence, removes information about ghost locations from our machine configurations, and prove an operational correspondence between the instrumented semantics and the post-erasure semantics.

Current Status. At the time of this writing, we have specified $\texttt{Oxide}_0$ and $\texttt{Oxide}_1$ and are working on the progress and preservation proofs for $\texttt{Oxide}_0$. We have a prototype type checker implemented in Scala for experimentation, and the beginnings of a Coq formalization. We plan to expand the Scala prototype further to include a compiler that elaborates Rust programs into Oxide, as well as an interpreter for Oxide. This will allow us to test our semantics against Rust for accuracy (along the lines of Guha et al. [10]'s testing of their core calculus for JavaScript).

3 A RUSTY FUTURE

With a precise framework for reasoning about source-level Rust programs, we hope that we, as a community, can build great things around Rust! We already have a number of ideas ourselves, many of which we are only just beginning to explore.

Language Extensions. With semantics in hand, the eager programming language researcher can jump at the opportunity to build nice, well-behaved extensions to Rust. This can be useful in trying to evolve the language through its RFC process [4] where informal formalisms have already begun to crop up [17]. Meanwhile, `Oxide` can also form the basis of domain-specific extensions. For example, we are designing extensions for secure multiparty computation [6, 19] in the style of Obliv-C [20]. Further, extensions can be built with the particular focus of enabling Rust programmers to write more reliable and correct software. This can include anything from verification-oriented language features as in Liquid Haskell [18] to tools for symbolic execution [12] and beyond.

Safe Interoperability. The Rust community has already begun to recognize the importance of building higher-level interfaces for interoperability with other programming languages [5, 9]. We hope to use `Oxide` to expand what is possible for these interoperability frameworks. In particular, we want to build on prior work on multi-language compilers [2, 15] and linking types [14] to support *provably safe* interoperation between languages.

Unsafe Code Guidelines. Finally, a pressing issue in the Rust community remains the open question of what **`unsafe`** code is safe to write [1]. With `Oxide`, we believe we are laying a foundation for answering such a question [13]. Going forward, we hope to use the intuitions from our work to contribute to the effort to develop unsafe code guidelines.

ACKNOWLEDGMENTS

We wish to thank Niko Matsakis for his invaluable feedback, discussions, and blogging. We would also like to thank Denis Merigoux for his feedback on a draft of this paper. This work was done at Inria Paris during Fall 2017 and Spring 2018 while Amal Ahmed and Aaron Weiss were visiting the Prosecco team. 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) for Aaron Weiss. This work is also supported in part by the European Research Council under ERC Starting Grant SECOMP (715753).

REFERENCES

[1] actix-web Contributors. 2018. Unsound uses of unsafe in API. https://github.com/actix/actix-web/issues/289. Accessed: 2018-07-20.

[2] Amal Ahmed. 2015. Verified Compilers for a Multi-Language World. In *1st Summit on Advances in Programming Languages (SNAPL 2015) (Leibniz International Proceedings in Informatics (LIPIcs))*, Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.), Vol. 32. 15–31.

[3] John Boyland. 2003. Checking interference with fractional permissions. In *Static Analysis: 10th International Symposium*. Springer, 55–72.

[4] The Rust Community. 2018. Rust RFCs. http://rust-lang.github.io/rfcs/. Accessed: 2018-06-01.

[5] Neon Contributors. 2017. Neon Bindings. https://www.neon-bindings.com/. Accessed: 2018-06-01.

[6] Jack Doerner and Abhi Shelat. 2017. Scaling ORAM for Secure Computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM, New York, NY, USA, 523–535. https://doi.org/10.1145/3133956.3133967

[7] Matthias Felleisen. 1990. On the Expressive Power of Programming Languages. In *Science of Computer Programming*. Springer-Verlag, 134–151.

[8] Matthew Fluet, Greg Morrisett, and Amal Ahmed. 2006. Linear Regions Are All You Need. In *European Symposium on Programming (ESOP)*. 7–21.

[9] Michael Gattozzi. 2016. curryrs. https://github.com/mgattozzi/curryrs. Accessed: 2018-06-01.

[10] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. 2010. The Essence of JavaScript, In ecoop. *European Conference on Object-Oriented Programming*.

[11] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: Securing the Foundations of the Rust Programming Language. In *Proceedings of the 45th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2018, Los Angeles, California, January 7-13, 2018.*

[12] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (July 1976), 385–394. https://doi.org/10.1145/360248.360252

[13] Niko Matsakis. 2016. Observational equivalence and unsafe code. http://smallcultfollowing.com/babysteps/blog/2016/10/02/observational-equivalence-and-unsafe-code/. Accessed: 2017-11-15.

[14] Daniel Patterson and Amal Ahmed. 2017. Linking Types for Multi-Language Software: Have Your Cake and Eat It Too. In *2nd Summit on Advances in Programming Languages (SNAPL 2017) (Leibniz International Proceedings in Informatics (LIPIcs))*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.), Vol. 71. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 12:1–12:15. https://doi.org/10.4230/LIPIcs.SNAPL.2017.12

[15] James T. Perconti and Amal Ahmed. 2014. Verifying an Open Compiler Using Multi-Language Semantics. In *European Symposium on Programming (ESOP)*.

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

[17] ticki. 2017. The pi type trilogy. https://github.com/rust-lang/rfcs/issues/1930. Accessed: 2018-06-01.

[18] 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

[19] Andrew Chi-Chih Yao. 1986. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*. 162–167. https://doi.org/10.1109/SFCS.1986.25

[20] Samee Zahur and David Evans. 2015. Obliv-C: A Language for Extensible Data-Oblivious Computation. Cryptology ePrint Archive, Report 2015/1153. https://eprint.iacr.org/2015/1153.