POPL: G: Monty PL and the Holy Grail

ACM SRC 2019 *Grand Finalist* Short Paper

Aaron Weiss
Northeastern University
weiss@ccs.neu.edu

1 THE QUEST FOR THE HOLY GRAIL

Conventional wisdom among programmers holds that one should choose "the best language for the job." Though exactly which language is "best" is often ambiguous, in practice, there has arisen a division between so-called *"systems programming"* languages that enable fine-grained control over the usage and layout of memory, and *"applications programming"* languages that provide the programmer with high-level abstractions to make correct software more quickly and with less effort. But in the lands of myths and legends, brave knights of programming languages research seek out the Holy Grail — a general-purpose programming language that enables programmers to get both the high-level abstractions of the latter and the fine-grained control of the former.

For decades, these Knights of PL have explored idea after idea in search of the grail. One of the most promising discoveries on this quest came as Ser Girard's Linear Logic [15] whose powers of reasoning about *resources* spawned a wealth of new work on high-level support for resource management in programming. Among these, the efforts of Ser Baker [3–6] on Linear Lisp are most relevant to our tale. On their quest, Ser Baker used a notion of *linearity* to enable efficient reuse of objects in memory in a functional programming language *without* garbage collection [3, 6]. Here, linearity prevented aliasing in programs, enabling the treatment of names as resources. Further, Ser Baker uncovered a relationship between linearity and stack machines that allow arbitrary manipulation on the top of the stack, capturing a *low-level* interpretation of linearity in programming [4].

Then, later quests by Sers Clarke, Potter, and Noble [11] and Sers Noble, Vitek, and Potter [25] brought us the discovery of *ownership types*, a programming discipline that empowers the programmer with the flexibility to control what kinds of aliasing patterns are allowed in their programs. While Ser Baker [3] addressed the question of how to design a high-level language without garbage collection, these next efforts address the question of how to control aliasing in programs to avoid classic systems problems like *encapsulation violations*, *use-after-free bugs*, and *data races*. Here, we see an essential insight — rather than choose one for the whole language, one can give the programmer a choice locally between uniqueness and mutability *or* sharedness and immutability (sometimes called *writing xor sharing*).

Around the same time, Sers Tofte and Talpin [29, 30] set out on their own quests to enable more fine-grained control of memory management in functional programming languages through the use of *regions*. These regions statically group objects into distinct sections of memory which are allocated and deallocated together. In the ensuing years, many worked on developing full-scale production languages that leveraged these ideas to create a *safe systems programming language* [13, 14, 16]. Yet, for a variety of reasons (not least of which a lack of large-scale engineering investment), none of these really caught on. Still, the ideas and techniques discovered on these quests moved the Knights of PL forward, helping to bring us to today.

In recent years, the Rust programming language [20] has taken off as the latest language at the intersection of low-level systems programming and high-level applications programming — bringing us the closest we have ever gotten to the Holy Grail. Its rapid growth and promising design have caught the attention of researchers [2, 7, 8, 18, 26, 31, 32] and practitioners [10, 22, 28] alike. To accomplish this, Rust integrates many of these ideas discovered on prior quests — including automatic memory management without garbage collection [3], flexible control over aliasing [25], and region-based memory management [30] — into a novel framework called *borrow checking*. But hailing from industry knights, Rust and its *borrow checker* lack the kinds of formal specifications that enable many of the reasoning techniques favored by the Knights of PL, and thus preventing their benefits.

For those uninitiated by the Knights of PL, I will try to make these benefits concrete. Historically, formal reasoning has played an essential role in developing new features for programming languages, ensuring the correctness of languages and their compilers, and enabling programmers to reason more precisely about their programs. For instance, Featherweight Java [17] and other similar efforts to formalize Java were used to experiment with designs for generics and led to Java's generics system being introduced in Java 1.5. As a squire myself, I have set out to try to bring this kind of power and interest to Rust by producing a formal account of the language, dubbed *Oxide*, on my journey to knighthood.

While there are some existing formalizations of Rust [8, 18, 26, 34], none capture a high-level understanding of Rust's essence (namely *ownership* and *borrowing*). In particular, the

first major effort, Patina [26], formalized an early version of Rust which predates much of the work to simplify and streamline the language, and was ultimately left unfinished. The next effort, known as Rusty Types [8], set out to characterize Rust-like type systems, developing a formal calculus, *Metal*, which relies on an algorithmic formulation of borrow checking that is less expressive than both Rust and Oxide. RustBelt [18], the most complete effort to date, formalized a *low-level*, intermediate language in continuation-passing style, making it difficult to reason about ownership as a *source-level* concept. Finally, an early version of Oxide [34] oversimplified some parts of the language and overcomplicated others. We have since revised and simplified Oxide greatly to get at its *essence*.

In this work, I present the key intuitions behind Oxide (Section 2), wade a bit into some of the essential formal elements (Section 3), and finally explore some of the ramifications of this newfound understanding (Section 4). Weiss et al. [35] features a larger, and more complete account of Oxide.

2 UNDERSTANDING THE BORROW CHECKER

As alluded to already, the essence of Rust lies in its *borrow checker*, a novel approach to *ownership*, which accounts for the most interesting aspects of the language's type system (or *static semantics*) and provides justification for its claims to *memory safety* and *data race freedom*. In this section, we work through a number of examples on our quest to understand ownership and borrowing, and how they are ultimately captured in Oxide.

Ownership for Great Good

The first element of Rust's *borrow checker* is a notion of *ownership* that builds on the quests of Ser Baker [3–6], which developed support for efficient reuse of objects in memory using *linearity*. In fact, there is a strong resemblance between this half of Rust's borrow checker and Ser Baker's 'use-once' variables [6]. We can view these ideas at work in Rust in the following example:

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

In this example, we declare a type `Point` consisting of a pair of unsigned 32-bit integers (denoted `u32`). Then, on line 3, we create a new `Point` named `pt`. We use `mut` to mark that the binding for `pt` can be reassigned, and we say that this value is *owned* by its identifier `pt`. Then, on line 4, we transfer ownership to `x` by *moving* the value from `pt`. After moving the value out of `pt`, we invalidate the old name, and thus, in the subsequent attempt to use it on line 5, we encounter an error as `pt` was already moved in the previous line. With the exception of required type annotations, this program is identical in Oxide, and produces the same error.

Borrowing for Flexibility

The second element of Rust's *borrow checker* is known as *borrowing* and represents the language's main departure from ideas like 'use-once' variables [6]. Rather than say that *everything* must be unique, Rust allows the programmer to locally make a decision between using unique references [23] with unguarded mutation *or* using shared references without such mutation.[1] This flexibility takes inspiration from the quests of Sers Noble, Clarke, Vitek, and Potter toward flexible alias protection and ownership types [11, 25]. We can once again see this at work 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 create shared references to `pt`. As noted in the comment, this program does not produce an error because the references specifically *allow* this kind of sharing. However, unlike with plain variable bindings (as in the last example), we cannot mutate through these references, and attempts to do so would result in an error at compile-time. However, the behavior changes when we try to mix shared and unique references to the same place:

```
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
8  //        a mutable loan to pt is live
9  ... // additional code that uses x and y
```

In this case, we've changed the borrow expression on line 4 to create a unique reference, and added explicit type annotations to our bindings on lines 3–5. This 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

[1] 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.

of a reference. Regions[2] in Rust (denoted `'a`, `'b`, etc.) can be understood as collections of these loans which together statically approximate which pointers could be used dynamically at a particular reference type. This is the sense in which Rust's regions are distinct from the existing literature on region-based memory management [1, 16, 29, 30].

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*. Rust employs a fine-grained notion of ownership allowing 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.

Formalizing Rust

These programs remain largely the same in Oxide, though I don't have the space to reproduce them all here. There are three main differences from Rust. First, we explicitly annotate the types of every binding. Second, acknowedgling the two distinct roles **`mut`** plays in Rust, I draw attention to its use as a qualifier for the uniqueness of references (renaming it to `uniq`). Third, I shift the language we use to discuss regions or lifetimes. In particular, since regions capture approximations of a reference's origin, I use the more precise term *approximate provenances*, and refer to their variable form (`'a`, `'b`, etc.) as *provenance variables*. With these differences in mind, let's revisit our last example after translating it into Oxide:

```
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
```

[2]Previously, Rust used the term *lifetime*, but recent efforts on a borrow checker rewrite called Polonius have been using the term region [21].

As already noted, the type annotations on lines 3–5 (i.e. for each let binidng) are now required, and we replaced **`mut`** with the qualifier `uniq`, denoting that the references on lines 4 and 5 are unique. The program is otherwise unchanged from the Rust version. Then, as in the original, the Oxide version of the program type checks successfully because the origins of the loans created on lines 4 and 5 do not overlap. That is, we know that `x` can only have originated from `pt.0`, `y` only from `pt.1`, and that `pt.0` and `pt.1` refer to disjoint portions of memory.

How did the type-checker determine this? First, while type-checking the let bindings, Oxide computes the concrete values for any provenance variables that appear in their types (such as `'x` and `'y`). Here, it determines that `'x` will be mapped to { $^{\texttt{uniq}}$`pt.0` } and `'y` to { $^{\texttt{uniq}}$`pt.1` }. These mappings tell us the potential provenance of each reference, where in this simple case, there is exactly one for each. Then, when type-checking the borrow expressions, Oxide looks at its place environment Γ to determine that there are no live loans to any place that overlaps with the place we're borrowing from. In this case, on line 5, the type-checker looks at the current environment and finds that the only live loan is $^{\texttt{uniq}}$`pt.0`, and since `pt.0` and `pt.1` do not overlap, the second borrow also type checks.

Information Loss. Though the example we just saw has a precise origin for each reference, provenances are, in general, approximate because of join points in the program. For example, in an if expression, we might create a new set of loans in one branch, and a different set of loans in the other branch. To keep the type-checker sound, we need to be conservative and act as if *both* sets of loans are live, and so, we combine the return types and environments from each branch.

3 OXIDE, MORE FORMALLY

We've now seen roughly enough to describe Oxide in more formal detail. First, note that loans are created and destroyed over the course of the program. This means that our type system has to somehow track the flow of this information as it type-checks each expression. As such, we use an environment-passing typing judgment where the output of the judgment includes an updated environment that may have added or removed some bindings (and thus created or destroyed some loans). We write this judgment as $\Sigma; \Delta; \Gamma \vdash e : \tau \Rightarrow \Gamma'$, which can be read as: under the global environment Σ (containing top-level program definitions including both function and type definitions), the type variable environment Δ (tracking in-scope type and provenance variables), and the place environment Γ (mapping places to their respective types), the expression e has type τ and produces an updated output environment Γ', which contains all of the remaining places *after* type-checking the expression e.

$$\textsc{T-Move}\quad \frac{\Gamma \vdash_{\texttt{uniq}} \pi \ : \ \tau \qquad \texttt{noncopyable}\ \tau}{\Sigma;\ \Delta;\ \Gamma \vdash \boxed{\pi} : \tau \Rightarrow \Gamma - \pi}$$

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

Figure 1: The essence of Oxide.

In Figure 1, we see two typing rules that capture the essence of how Oxide models the behavior of Rust's borrow checker. Following the traditions of the Knights of PL, these rules are written in the style of *natural deduction* where each rule can be read as "if we have a proof of the statements above the horizontal line, then we can combine them to construct a proof of the statement below the line." As such, to understand our two rules, it is necessary to understand the meaning of the judgement in their premise ($\Gamma \vdash_{\omega} \pi \ : \ \tau$).

This judgment, called ω-safety, says "in the place environment Γ, it is safe to use the place 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 this judgment is the one 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.

For both more examples and more details about the Oxide formalism, interested readers should look at the full article by Weiss, Patterson, Matsakis, and Ahmed [35].

4 OXIDE IN CONTEXT

As with prior quests directed at understanding real world languages, Oxide provides a formal framework for reasoning about the behavior of programs in one such language — i.e. programs in surface-level Rust. One prominent example is Featherweight Java [17] which supported a wealth of new research on Java. And as with prior quests such as this, a number of promising avenues for future work on Rust open up that leverage our newfound reasoning ability with 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 [2, 7, 31, 32], but without a semantics the possibilities are limited. For example, the work by Astrauskas et al. [2] builds verification support for Rust into Viper [24], but uses an ad-hoc subset of the language without support for shared references. We believe that our work on Oxide can help extend such work to support more Rust code, and will enable further verification techniques like those seen in $F^{\star}$ [27] and Liquid Haskell [33].

Verified Compilation

Given the prevalence of memory safety issues in security vulnerabilities, Rust's guarantees of memory safety lend themselves well to building security-critical applications. However, many security-critical applications (like cryptography code) require that these guarantees still hold after compilation, and further require a guarantee that timing of the program is preserved under compilation (to avoid the introduction of side-channel attacks). Rust's existing compiler toolchain — which is built using LLVM [19] — is unable to formally prove preservation of these kinds of guarantees, and indeed, for timing, optimizations are designed to do almost entirely the opposite. As such, another avenue for future work would be to develop a verified compiler for Rust that preserves program semantics and timing behavior, perhaps by compilation to Vellvm [37] or CompCert Clight [9].

Language-based 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. Similarly, Oxide can support building extensions for data-oblivious computing (as in work by Zahur and Evans [36]) and relaxed noninterference (as in recent work by Cruz et al. [12]).

5 THE QUEST MARCHES ON...

As mentioned in Section 1, Rust is the closest the Knights of PL have gotten to the Holy Grail of Programming — the language that provides programmers all the benefits of high-level abstractions and all the fine-grained control of systems programming. With Oxide, we now have the tools to interact with and build on Rust more formally. This work too will help bring us closer to the Grail. Yet, we also remain so far! Still, I — forever the fool — continue on my own search.

REFERENCES

[1] Amal Ahmed, Matthew Fluet, and Greg Morrisett. 2005. A Step-Indexed Model of Substructural State. In *International Conference on Functional Programming (ICFP), Tallinn, Estonia.*

[2] 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.

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

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

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

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

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

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

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

[10] Chucklefish. 2018. *Rust Case Study: Chucklefish Taps Rust to Bring Safe Concurrency to Video Games.* Technical Report. The Rust Project Developers.

[11] 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).*

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

[13] Robert DeLine and Manuel Fähndrich. 2001. Enforcing High-Level Protocols in Low-Level Software. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).*

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

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

[16] 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.*

[17] 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).

[18] 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.*

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

[20] Nicholas Matsakis and Felix S. Klock II. 2014. The Rust Language. In *ACM SIGAda Annual Conference on High Integrity Language Technology (HILT).*

[21] 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/.

[22] Cade Metz. 2016. The Epic Story of Dropbox's Exodus From the Amazon Cloud Empire. https://www.wired.com/2016/03/epic-story-dropboxs-exodus-amazon-cloud-empire/.

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

[24] 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).*

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

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

[27] 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.*

[28] The Rust Project Developers. 2019. Production Users of Rust. https://www.rust-lang.org/production/users. Accessed: 2019-04-01.

[29] 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.*

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

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

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

[33] 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) (ICFP '14).* ACM, New York, NY, USA, 269–282. https://doi.org/10.1145/2628136.2628161

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

[35] Aaron Weiss, Daniel Patterson, Nicholas D. Matsakis, and Amal Ahmed. 2019. Oxide: The Essence of Rust. *arXiv e-prints*, Article arXiv:1903.00982 (Mar 2019), arXiv:1903.00982 pages. arXiv:cs.PL/1903.00982

[36] Samee Zahur and David Evans. 2015. Obliv-C: A Language for Extensible Data-Oblivious Computation. *Cryptology ePrint Archive* (2015).

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