Modular Program Slicing Through Ownership

ANONYMOUS AUTHOR(S)

Program slicing, or identifying the subset of a program relevant to a value, relies on understanding the dataflow of a program. In languages with functions and mutable pointers like C or Java, tracking dataflow has historically required whole-program analysis, which can be be slow and challenging to integrate in practice. Advances in type systems have shown how to modularly track dataflow through the concept of ownership. We demonstrate that ownership can modularize program slicing by using types to compute a provably sound and reasonably precise approximation of mutation. We design a modular slicing algorithm for Rust, an industrial-grade ownership-based programming language. We prove the algorithm's soundness as a form of noninteference on the Oxide formal model of Rust. Then we empirically evaluate the precision of the slicer, showing that modular slices are the same as whole-program slices in 95% of cases drawn from large Rust codebases.

Additional Key Words and Phrases: program slicing, ownership, symbolic procedure summaries, modular static analysis, rust

1 INTRODUCTION

Program slicing is the task of identifying the subset of a program relevant to computing a value of interest. The concept of slicing was introduced 40 years ago when Weiser [1982] demonstrated that programmers mentally construct slices while debugging. Since then, hundreds of papers have been published on implementing automated program slice, as surveyed by Silva [2012]; Xu et al. [2005]. Despite these efforts, a review of slicers found "slicing-based debugging techniques are rarely used in practice" [Parnin and Orso 2011].[1]

A major challenge for slicing is addressing the underlying program analysis problems. At a high level, slicing is about dataflow — if x is relevant, then any means by which data flows into x are also relevant. In today's programming languages, analyzing dataflow is difficult because of the interaction of two features: functions and pointers. For example, imagine slicing a value in a function f which calls a function g. In a language without side-effects, then the only relevance g could possibly have in f is its return value. But in a language that allows effects such as mutation on pointers, g could modify data used within f, requiring a pointer analysis. Moreover, if f is a higher-order function parameterized on g, then the slice must consider all the possible functions that g could be, i.e. control-flow analysis.

The standard solution for analyzing programs with pointers and functions is *whole-program analysis*. That is, for a given function of interest, analyze the definitions of all of the function's callers and callees in the current codebase. However, whole-program analysis suffers from a few logistical and conceptual issues:

(1) *Analysis time scales with the size of the whole program*: the time complexity of whole-program analysis scales either polynomially or exponentially with the number of call sites in the program, depending on context-sensitivity [Might et al. 2010]. In practice, this means more complex codebases can take substantially longer to analyze. For instance, the recent PSEGPT pointer analysis tool [Zhao et al. 2018] takes 1 second on a codebase of 282,000 lines of code and 3 minutes on a codebase of 2.2 million lines of code.

[1] The only open-source, functioning slicers the authors could find are Frama-C [Cuoq et al. 2012] and dg [Chalupa 2016]. Slicing tools for Java like Kaveri [Jayaraman et al. 2005] no longer work. The most industrial-strength slicing tool, CodeSurfer [Balakrishnan et al. 2005] was GrammaTech's proprietary technology and appears to no longer exist.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

(2) *Analysis requires access to source code for the whole program*: an assumption of analyzing a whole program is that a whole program is actually accessible. However, many programs use libraries that are shipped as pre-compiled objects with no source code, either for reasons of efficiency or intellectual property.
(3) *Analysis results are anti-modular*: when analyzing a particular function, relying on calling contexts to analyze the function's inputs means that any results are not universal. Calling-context-sensitive analysis determine whether two pointers alias *in the context of the broader codebase*, so alias analysis results can change due to modifications in code far away from the current module.

These issues are not new — Rountev et al. [1999] and Cousot and Cousot [2002] observed the same two decades ago when arguing for modular static analysis. The key insight arising from their research is that static analysis can be modularized by computing *symbolic procedure summaries*. For instance, Yorsh et al. [2008] show how to automatically summarize which inputs and outputs are possibly null for a given Java function. The analysis is modular because a function's summary can be computed only given the summaries, and not definitions, of callees in the function. In such prior work, the language of symbolic procedure summaries has been defined in a separate formal system from the programming language being analyzed, such as the micro-transformer framework of Yorsh et al. [2008].

Our work begins with the observation: *function type signatures are symbolic procedure summaries*. The more expressive a language's type system, the more behavior that can be summarized by a type. Nearly all work on program slicing, dataflow analysis, and procedure summaries has operated on C, Java, or equivalents. These languages have impoverished type systems, and so any interesting static analysis requires a standalone abstract interpreter. However, if a language's type system were expressive enough to encode information about dataflow, then a function's type signature could be used to reason about the aliasing and side effects needed for slicing. Moreover, a function's type signature is required information for a compiler to export when building a library. Using the type system for dataflow analysis therefore obviates the logistical challenge of integrating an external analysis tool into a complex build system.

Today, the primary technique for managing dataflow with types is *ownership*. Ownership is a concept that has emerged from several intersecting lines of research on linear logic [Girard 1987], class-based alias management [Clarke et al. 1998], and region-based memory management [Grossman et al. 2002]. Generally, ownership refers to a system where values are owned by an entity, which can temporarily or permanently transfer ownership to other entities. The type system then statically tracks the flow of ownership between entities. Ownership-based type systems enforce the invariant that values are not simultaneously aliased and mutated, either for the purposes of avoiding memory errors, data races, or abstraction violations.

Our thesis is that ownership can modularize program slicing by using types to compute a provably sound and reasonably precise approximation of the necessary dataflow information. We build this thesis in five parts:

(1) We provide an intuition for the relationship between ownership and slicing by describing how ownership works in Rust, an industrial-grade ownership-based programming language (Section 2).
(2) We formalize an algorithm for modular slicing as an extension to the type system of Oxide [Weiss et al. 2019], a formal model of Rust's static and dynamic semantics (Section 3 and Section 4).
(3) We prove the soundness of this algorithm as a form of noninterference, building on the connection between slicing and information flow established by Abadi et al. [1999] (proof

sketch in Section 4.2 and full proof in Appendix A.2, located in supplementary material for submission).

(4) We describe an implementation of the slicing algorithm for Rust, translating the core insights of the algorithm to work on a lower-level control-flow graph (Section 5).

(5) We evaluate the precision of the modular Rust slicer against a whole-program slicer on a dataset of 10 codebases with a total of 280k LOC. We find that modular slices are the same size as whole-program slices 95.4% of the time, and are on average 7.6% larger in the remaining 4.6% of cases (Section 6).

2 PRINCIPLES

A backwards static slice is the subset of a program that could influence a particular value (backwards) under any possible execution (static). A slice is defined with respect to a slicing criterion, which is a variable at a particular point in a program. In this section, we provide an intuition for how slices interact with different features of the Rust programming language, namely: places (2.1), references (2.2), function calls (2.3), and interior mutability (2.4).

2.1 Places

A place is a reference to a concrete piece of data in memory, like a variable `x` or path into a data structure `x.field`. Slices on places are defined by bindings, mutation, and control flow. For instance, the Rust snippet on the right shows the slice in orange of a place in green. The assignment `x = y` means `y` is relevant for the slice, so the statement `let y = 2` is relevant as well. Because `z` is not used in the computation of `x`, then `let z = 3` is not relevant. Additionally, because `x = y` overwrites the previous value of `x`, then the original assignment `x = 1` is not relevant either.

```
1 let mut x = 1;
2 let y = 2;
3 let z = 3;
4 x = y;
5 println!("{} {}", x, z);
```

```
1 let mut x = 1;
2 let mut y = 2;
3 if y > 0 { x = 3; }
4 else     { y = 4; }
5 println!("{} {}", x, y);
```

If a mutation is conditioned on a predicate (as in line 3 in the snippet on the left) then the predicate is relevant to the mutated place. In this example, because `x = 3` is only executed if `y > 0`, then the value of `y` (at the time-of-check) is relevant to the value of `x`.

Slices on composite data structures are defined by whether a mutation conflicts with a particular path into the data structure. For example, consider slicing on a tuple as in the three snippets below (note that `t.n` gets the n-th field of the tuple `t`):

```
1 let mut t = (0, 1, 2);
2 t = (3, 4, 5);
3 t.0 = 6;
4 t.1 = 7;
5 println!("{:?}", t);
```

```
1 let mut t = (0, 1, 2);
2 t = (3, 4, 5);
3 t.0 = 6;
4 t.1 = 7;
5 println!("{}", t.0);
```

```
1 let mut t = (0, 1, 2);
2 t = (3, 4, 5);
3 t.0 = 6;
4 t.1 = 7;
5 println!("{}", t.2);
```

In this program, when slicing on `t`, changing the value of a field of a structure changes the value of the whole structure, so `t.1 = 7` is part of the slice on `t`. However, when slicing on `t.0`, the path `t.0` is disjoint from the path `t.1`, so `t.1 = 7` is not part of the slice on `t.0`. Similarly, when slicing on `t.2`, the only relevant assignment is `t = (3, 4, 5)`. More generally, a place conflicts with another place if either's path is a prefix of the other's. For instance, `t.0` conflicts with both `t` (parent) and `t.0.1` (child) but not `t.1` (sibling). This leads to the first slicing principle:

Principle 1 (Slicing principle for places)**.** *A mutation to a place is a mutation to all conflicting places.*

This principle provides an intuition for making an algorithm that constructs slices. For instance, take the last example above on the left. On line 4, when `t.1` is mutated, that mutation is registered as part of the slice on every conflicting place, specifically `t` and `t.1`.

2.2 References

Pointers are the first major challenge for slicing. A mutation to a dereferenced pointer is a mutation to any place that is possibly pointed-to, so such places must be known to the slicer. For example:

```
1 let mut x = 1;
2 let y = &mut x;
3 *y = 2;
4 let z = &x;
5 println!("{}", *z);
```

Rust has two distinct types of pointers, which are called "references" to distinguish them from "raw pointers" with C-like behavior (discussed in Section 2.4). For a given type `T`, there are immutable references of type `&T`, and mutable references of type `&mut T` which correspond respectively to the expressions `&x` and `&mut x`. Because `y` points to `x`, then the mutation through `y` is relevant to the read of `*z`. We refer to the left-hand side of assignment statements like `*y` as "place expressions", since they could include dereferences.

The task of determining what a reference can point-to is called *pointer analysis*. While many methods exist for pointer analysis [Smaragdakis and Balatsouras 2015], our first key insight is that Rust's ownership types implicitly perform a kind of modular pointer analysis that we can leverage for slicing. To understand why, we first need to describe two ingredients: the goal, i.e. what ownership is trying to accomplish, and the mechanism, i.e. how ownership-checking is implemented in the type system.

The core goal of ownership is eliminating simultaneous aliasing and mutation. In Rust, achieving this goal enables the use of references without garbage collection while retaining memory safety. For instance, these three classes of errors are all caught at compile-time:

```
1 // Dangling reference
2 let p = {
3   let x = 1; &x
4 };
5 let y = *p;
```

```
1 // Use-after-free
2 let d = tempdir();
3 let d2 = &d;
4 d.close();
5 let p = d2.path();
```

```
1 // Iterator invalidation
2 let mut v = vec![1,2];
3 for x in v.iter() {
4   v.push(*x);
5 }
```

From left-to-right: the dangling references is caught because `x` is deallocated at the end of scope on line 4, which is a mutation, conflicting with the alias `&x`. The use-after-free is caught because `d.close()` requires ownership of `d`, which prevents an alias `d2` from being live. The iterator invalidation case is subtler: `x` is a pointer to data within `v`. However, `v.push(*x)` could resize `v` which would copy/deallocate all vector elements to a new heap location, invalidating all pointers to `v`. Hence `v.push(*x)` is a simultaneous mutation and alias of the vector.

Catching these errors requires understanding which places are pointed by which references. For instance, knowing that `x` points to an element of `v` and not just any arbitrary `i32`. The key mechanism behind these ownership checks is *lifetimes*.

```
1 let mut x: i32 = 1;
2 let y: &'1 i32 = &'0 mut x;
3 *y = 2;
4 let z: &'3 i32 = &'2 x;
5 println!("{}", *z);
```

Each reference expression and type has a corresponding lifetime, written explicitly in the syntax `'n` on the left, where `n` is an arbitrary and unique number. The name "lifetime" implies a model of lifetimes as the live range of the reference. Prior work on region-based memory management like Tofte and Talpin [1997] and Grossman et al. [2002] use this model.

However, recent work from Matsakis [2018] and Weiss et al. [2019] have devised an alternative model of lifetimes as "provenances" or "origins" that more directly correspond to a pointer analysis. In essence, a lifetime is the set of places that a reference could point-to. For the above example,

that would be `'n = {x}` for all `n`, because each reference points to `x`. As a more interesting example, consider the code on the left.

```
1 let mut x = 1;
2 let mut y = 2;
3 let z: &'2 mut i32 = if true {
4   &'0 mut x
5 } else {
6   &'1 mut y
7 };
8 let w: &'4 mut i32 = &'3 mut *z;
9 *w = 1;
```

There, lifetimes for borrow expressions are assigned to the place being borrowed, so `'0 = {x}` and `'1 = {y}`. Because `z` could be assigned to either reference, then `'2 = '0 ∪ '1 = {x, y}`. An expression of the form `&*p` is called a "reborrow", as the underlying address is being passed from one reference to another. To register that a reference is reborrowed, the reborrowed place is also added to the lifetime, so `'3 = '4 = {x, y, *z}`. More generally:

Principle 2 (Slicing principle for references)**.** *The lifetime of a reference contains all potential aliases of what the reference points-to.*

In the context of slicing, then to determine which places could be modified by a particular assignment, one only needs to look up the aliases in the lifetime of references. For instance, `*w = 1` would be part of a slice on `*z`, because `*z` is in the lifetime `'4` of `w`.

2.3 Function calls

The other major challenge for slicing is function calls. For instance, consider slicing a call to an arbitrary function `f` with various kinds of inputs:[2]

```
1 let x = String::from("x");
2 let y = String::from("y");
3 let mut z = String::from("z");
4 let w = f(x, &y, &mut z);
5 println!("{} {} {}", y, z, w);
```

The standard approach to slicing `f` would be to inspect the definition of `f`, and recursively slice it by translating the slicing criteria from caller to callee (e.g. see Weiser [1982] for an example). However, our goal is to avoid using the definition of `f` (i.e. a whole-program analysis) for the reasons described in Section 1.

To modularly slice through function calls, we need to approximate the effects of `f` in a manner that is sound, but also as precise as possible. Put another way, what mutations could possibly occur as a result of calling `f`? Consider the three cases that arise in the code above.

- Passing a value `x` of type `String` (or generally of type `T`) moves the value into `f`. Therefore it is an ownership error to refer to `x` after calling `f` and we do not need to consider slices on `x` after `f`.
- Passing a value `y` of type `&String` (or `&T`) passes an immutable reference. Immutable references cannot be mutated, therefore `y` cannot change in `f`.[3]
- Passing a value `z` of type `&mut String` (or `&mut T`) passes a mutable reference, which could possibly be mutated. This case is therefore the only observable of effect `f` apart from its return value.

Without inspecting `f`, we cannot know how a mutable reference is modified, so we have to conservatively assume that every argument was used as input to a mutation. Therefore the modular slice of each variable looks as in the snippets below:

[2] Why is `String::from` needed? The literal `"Hello world"` has type `&'static str`, meaning an immutable reference to the binary's string pool which lives forever. The function `String::from` converts the immutable reference into a value of type `String`, which stores its contents on the heap and allows the string to be mutated.

[3] A notable detail to the safety of immutable references is that immutability is transitive. For instance, if `b = &mut a` and `c = &b`, then `a` is guaranteed not to be mutated through `c`. This stands in contrast to other languages with pointers like C and C++ where the `const` keyword only protects values from mutation at the top-level, and not into the interior fields.

```
let x = String::from("x");
let y = String::from("y");
let mut z = String::from("z");
let w = f(x, &y, &mut z);
println!("{} {} {}", y, z, w);
```

```
let x = String::from("x");
let y = String::from("y");
let mut z = String::from("z");
let w = f(x, &y, &mut z);
println!("{} {} {}", y, z, w);
```

```
let x = String::from("x");
let y = String::from("y");
let mut z = String::from("z");
let w = f(x, &y, &mut z);
println!("{} {} {}", y, z, w);
```

Note that like `z` (above), the return value `w` (left) is also assumed to be influenced by every input to `f`. Implicit in these slices are additional assumptions about the limitations of `f`. For example, in C, a function could manufacture a pointer to the stack frame above it and mutate the values, meaning `f` could mutate `y` (even if `y` was not an input!). Similarly, functions could potentially read arbitrary data (e.g. global variables) that would influence mutations apart from just the arguments.

However, allowing such pointer manipulation would easily break ownership safety, since fundamentally it permits unchecked aliasing. Hence, our principle:

Principle 3 (Slicing principle for function calls)**.** *When calling a function, (a) only mutable references in the arguments can be mutated, and (b) the mutations and return value are only influenced by the arguments.*

This principle is essentially a worst-case approximation to the function's effects. It is the core of how we can modularly slice programs, because a function's definition does not have to be inspected to analyze what it can mutate.

A caveat to this principle is global variables: (Principle 3-a) is not true with mutable globals, and (Principle 3-b) is not true with read-only globals. Mutable globals are disallowed by the rules of ownership, as they are implicitly aliased and hence disallowed from being mutable. However, read-only globals are ownership-safe (and hence permitted in Rust). For simplicity we do not consider read-only globals in this work.

Another notable detail is the interaction of function calls and lifetimes. Pointer analysis, like slicing, has historically been done via whole-program analysis for maximum precision. However, Rust can analyze lifetimes (and subsequently what references point-to) modularly just by looking at the type signature of a called function using *lifetime parameters*. Consider the function `Vec::get_mut` that returns a mutable reference to an element of a vector. For instance, `vec![5, 6].get_mut(0)` returns a mutable reference to the value 5. This function has the type signature:

```
Vec::get_mut  :  ∀ 'a, T . (&'a mut Vec<T>, usize) -> &'a mut T
```

Because this type signature is parametric in the lifetime `'a`, it can express the constraint that the output reference `&'a mut T` must have the same lifetime as the input reference `&'a mut Vec<T>`. Therefore the returned pointer is known to point to the same data as the input pointer, but without inspecting the definition of `get_mut`.

2.4 Interior mutability

The previous sections describe a slicing strategy for the subset of Rust known as "safe Rust", that is programs which strictly adhere to the rules of ownership. Importantly, Rust also has the **`unsafe`** feature that gives users access to raw pointers, or pointers with similar unchecked behavior to C. Most commonly, **`unsafe`** code is used to implement APIs that satisfy ownership, but not in a manner that is deducible by the type system. For example, shared mutable state between threads:

```
let value = Arc::new(Mutex::new(0));

let value_ref = value.clone();
thread::spawn(move || {
  *value_ref.lock().unwrap() += 1;
}).join().unwrap();

assert!(*value.lock().unwrap() == 1);
```

In this snippet, two threads have ownership over two values of type `Arc<Mutex<i32>>` which internally point to the same number. Both threads can call `Mutex::lock` which takes an immutable reference to an `&Mutex<i32>` and returns a mutable reference `&mut i32` to the data inside.[4] This nominally violates ownership, as the data is aliased (shared by two threads) and mutable (both can mutate).

The mutex is ownership-safe only because its implementation ensures that both threads cannot *simultaneously* access the underlying value in accordance with the system mutex's semantics. For our purposes, the aliasing between `value` and `value_ref` is not possible to observe using the type system alone. For example, in our algorithm, slicing on `value` would *not* include mutations to `value_ref`. This is because the data inside the mutex has type `*mut i32` (a raw pointer), and without a lifetime attached, our algorithm has no way to determine whether `value` and `value_ref` are aliases just by inspecting their types.

More broadly, modular slicing is only sound for safe Rust. The point of this work is to say: when a program can be statically determined to satisfy the rules of ownership, then modular slicing is sound. The principles above help clarify the specific assumptions made possible by ownership, which are otherwise impossible to make in languages like C or Java. Astrauskas et al. [2020] found that 76.4% of published Rust projects contain no unsafe code, suggesting that safe Rust is more common than not. However, their study does not account for safe Rust built on internally-unsafe abstractions like `Mutex`, so it is difficult to estimate the true likelihood of soundness in practice. We discuss the issue of slicing with unsafe code further in Section 6.4.1.

3 FORMAL MODEL

To build an algorithm from these principles, we first need a formal model to describe and reason about the underlying language. Rather than devise our own, we build on the work of Weiss et al. [2019]: Oxide is a model of (safe) Rust's surface language with a formal static and dynamic semantics, along with a proof of syntactic type soundness. Importantly, Oxide uses a provenance model of lifetimes which we leverage for our slicing algorithm.

We will incrementally introduce the aspects of Oxide's syntax and semantics as necessary to understand our principles and algorithm. We describe Oxide's syntax (3.1), static semantics (3.2) and dynamic semantics (3.3), and then apply these concepts to formalize the slicing principles of the previous section (3.4).

3.1 Syntax

Figure 1 shows a subset of Oxide's syntax along with a labeled example. An Oxide program consists of a set of functions Σ (the "global environment"), where each function body is an expression e. The syntax is largely the same as Rust's with a few exceptions:

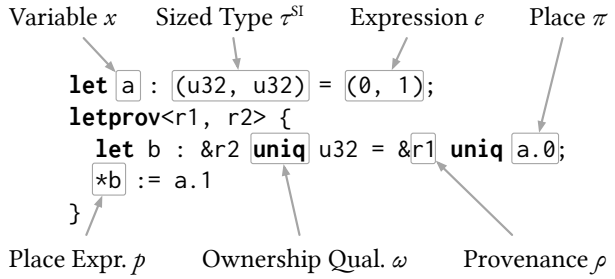
- Lifetimes are called "provenances", and they are both explicit in expressions and types throughout the program, and initially bound via letprov expressions or as function parameters.
- Rather than having immutable references `&'a` τ and mutable references `&'a mut` τ, Oxide calls them "shared" references $\&\rho$ shrd τ and "unique" references $\&\rho$ uniq τ.

[4] Technically the returned type is a `LockResult<MutexGuard<'a, i32>>` but the distinction isn't relevant here.

$$\text{Path } q ::= \varepsilon \mid n.q \qquad \text{Provenances } \rho ::= \varrho \mid r$$
$$\text{Places } \pi ::= x.q \qquad \text{Ownership Qualifiers } \omega ::= \text{shrd} \mid \text{uniq}$$
$$\text{Place Expressions } p ::= x \mid *p \mid p.n \qquad \text{Base Types } \tau^{\text{B}} ::= \text{unit} \mid \text{u32} \mid \text{bool}$$
$$\text{Constants } c ::= () \mid n \mid \text{true} \mid \text{false} \qquad \text{Sized Types } \tau^{\text{SI}} ::= \tau^{\text{B}} \mid \&\rho\ \omega\ \tau^{\text{XI}} \mid (\tau_1^{\text{SI}}, \ldots, \tau_n^{\text{SI}})$$

$$\text{Expressions } e ::= c \mid p \mid \&r\ \omega\ p \mid \texttt{letprov}\langle r \rangle\ \{\ e\ \} \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$$
$$\mid \texttt{let}\ x : \tau_a^{\text{SI}}\ =\ e_1;\ e_2 \mid \pi := e \mid p := e \mid e_1;\ e_2 \mid f\langle \overline{\Phi}, \overline{\rho}, \overline{\tau} \rangle(\pi)$$
$$\text{Global Entries } \varepsilon ::= \texttt{fn}\ f\langle \overline{\psi}, \overline{\varrho}, \overline{\alpha} \rangle(x : \tau_a^{\text{SI}}) \to \tau_r^{\text{SI}}\ \texttt{where}\ \overline{\varrho_1 : \varrho_2}\ \{\ e\ \}$$
$$\text{Global Environment } \Sigma ::= \bullet \mid \Sigma, \varepsilon$$

(a) Subset of Oxide syntax, reproduced from Weiss et al. [2019, p. 8]. The only difference in this subset is that closures are eliminated and functions are simplified to take one argument.



(b) Syntactic forms and corresponding metavariables labeled in context of an example

Fig. 1. Formal elements of Oxide and their explanation (excerpts).

- Provenances are divided into "concrete" (r) and "abstract" (ϱ). Concrete provenances are used by borrow expressions, and abstract provenances are function parameters used for inputs with reference type.

3.2 Static semantics

Expressions are typechecked via the judgment $\Sigma; \Delta; \Gamma \vdash e : \tau \Rightarrow \Gamma'$, read as: "$e$ has type τ under contexts Σ, Δ, Γ producing new context Γ'." Δ contains function-level type and provenance variables. Γ maps variables to types and provenances to pointed-to place expressions with ownership qualifiers ("loans" in a "loan set" of the form $\ell ::= {}^{\omega}p$). For instance, when type checking `*b := a.1` in Figure 1b, the inputs would be $\Delta = \bullet$ (empty) and $\Gamma = \{a \mapsto (\text{u32}, \text{u32}),\ b \mapsto \&r_2\ \text{uniq}\ \text{u32},\ r_1 \mapsto \{\ {}^{\text{uniq}}a.0\},\ r_2 \mapsto \{\ {}^{\text{uniq}}a.0\}\}$.

Typechecking relies on a number of auxiliary judgments, such as subtyping ($\tau_1 \lesssim \tau_2$) and ownership-safety ($\Delta; \Gamma \vdash_{\omega} p \Rightarrow \{\overline{\ell}\}$, read as "$p$ has ω-loans $\{\overline{\ell}\}$ in the contexts Δ, Γ"). As an

example, consider T-Assign [2019, p. 11] for the assignment expression $\pi := e$:

$$\textsc{T-Assign}\quad \frac{\Sigma;\Delta;\Gamma \vdash e : \tau^{\mathrm{SI}} \Rightarrow \Gamma_1 \qquad \Gamma_1(\pi) = \tau^{\mathrm{SX}} \qquad (\tau^{\mathrm{SX}} = \tau^{\mathrm{SD}} \vee \Delta;\Gamma_1 \vdash_{\mathsf{uniq}} \pi \Rightarrow \{\,^{\mathsf{uniq}}\pi\}) \qquad \Delta;\Gamma_1 \vdash \tau^{\mathrm{SI}} \lesssim \tau^{\mathrm{SX}} \Rightarrow \Gamma'}{\Sigma;\Delta;\Gamma \vdash \pi := e : \mathsf{unit} \Rightarrow \Gamma'[\pi \mapsto \tau^{\mathrm{SI}}] \rhd \pi}$$

A valid assignment must be type-safe and ownership-safe. To be type-safe, the type of the expression τ^{SI} must be a subtype of the place's type $\Gamma_1(\pi)$. To be ownership-safe, the type must either be dead[5], or π must have unique ownership over itself, i.e. there should be no live references to π. If so, then the type of π is updated to τ^{SI}.

3.3 Dynamic semantics

Expressions are executed via a small-step operational semantics, and the program state is a pair of a stack and an expression. A single step is represented by the judgment $\Sigma \vdash (\sigma; e) \rightarrow (\vec{\sigma}; e')$. A stack σ is a list of stack frames $\varsigma ::= \{\overline{x \mapsto v}\}$ that map variables to values. For example, consider E-Assign [2019, p. 16] that covers $\pi := e$ and $p := e$ expressions:[6]

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

This rule introduces several new shorthands and administrative forms:

- The syntax $\pi^{\square}[x]$ means the decomposition of a place π into a root variable x and context $\pi^{\square}$. For example, if $\pi = a.0$ then $\pi^{\square} = \square.0$ and $x = a$.
- A value context $\mathcal{V}$ is a form to handle mutation of compound objects. For instance, if $a = (0, 1)$, when evaluating $a.0 := 2$, then $\mathcal{V} = (\square, 1)$. $\mathcal{V}$ copies all the old values, leaving a hole for the one value to be updated. Then the syntax $\mathcal{V}[v]$ means plugging v into the hole. Hence, mutating a place is represented as $\sigma[x \mapsto \mathcal{V}[v]]$.
- The judgment $\sigma \vdash p \Downarrow \pi^{\square}[x] \times \mathcal{V}$ evaluates a place expression under the current stack into a place π and value context $\mathcal{V}$. For instance, if p is a dereference of a reference, then this judgment resolves p to the concrete memory location π it points-to under σ.

3.4 Formalized principles

Now, we have enough of the language formalized to give a precise statement of each slicing principle from Section 2. Each principle will be presented with the corresponding theorem, using underlining in color to highlight correspondences.

In the principles and corresponding algorithm/proofs, there are many concepts which we distinguish by notational convention. We denote objects by their metavariable, e.g. p or σ, and add a sans-serif subscript for distinct roles where needed, e.g. π_{mut} for a mutated place and π_{any} for an arbitrary place. We generally use a superscript i for an object that varies between two executions

[5] Oxide uses the metavariables τ^{SD} to mean "dead types" and τ^{SX} to mean "possibly dead types". A place becomes dead when it is moved, e.g. see T-Move in [2019, p. 11]. T-Assign allows a dead place to be revived. For instance, consider the program:

$$\textbf{let}\ x : \mathsf{String} = \text{“}a\text{”};\ \mathsf{print}(x);\ x := \text{“}b\text{”}$$

When $\mathsf{print}(x)$ moves x, its type is updated to $\mathrm{String}^{\dagger}$ in Γ. Then the T-Assign rule permits x to be assigned again to "revive" that place, setting its type back to String.

[6] This E-Assign rule is not the exact same rule that appears in Weiss et al. [2019, p. 16], as the published version is incorrect. In correspondence with the authors, we determined that the rule presented here has the intended semantics. Additionally, we do not use the referent $\mathcal{R}$ construct of Oxide since we do not consider arrays in this paper, so we use π anywhere $\mathcal{R}$ would otherwise appear.

of a program, like σ^i or v^i. And we use right arrows to indicate changes to an object after stepping (instead of primes, to avoid polluting the superscript), e.g. σ^i versus $\vec{\sigma}^i$.

Principle 1 (Slicing principle for places)
A mutation to a place is a mutation to all conflicting places.

Theorem 3.1. *Let:*
- $\pi_{\text{mut}} = \pi^{\square}_{\text{mut}}[x]$, σ *where* $\sigma \vdash \pi^{\square}_{\text{mut}} \times x \Downarrow \mathcal{V}$
- $v, \vec{\sigma} = \sigma[x \mapsto \mathcal{V}[v]]$
- π_{any} *be any place*

Then $\sigma(\pi_{\text{any}}) \neq \vec{\sigma}(\pi_{\text{any}}) \implies \pi_{\text{any}} \sqcap \pi_{\text{mut}}$

As described in Section 3.3, a mutation to a place is represented by updating a variable x in a stack σ by plugging a value v into a value context $\mathcal{V}$. To denote a conflict, we reuse the notation from Oxide that $\pi_1 \# \pi_2$ means "π_1 and π_2 do not conflict", or more formally:

$$x_1.q_1 \# x_2.q_2 \stackrel{\text{def}}{=} x_1 \neq x_2 \vee ((q_1 \text{ is not a prefix of } q_2) \wedge (q_2 \text{ is not a prefix of } q_1))$$

Conversely, we use the shorthand $\pi_1 \sqcap \pi_2 \stackrel{\text{def}}{=} \neg(\pi_1 \# \pi_2)$. So if a place π_{any} is changed when π_{mut} is mutated, then it must be that $\pi_{\text{any}} \sqcap \pi_{\text{mut}}$.

Principle 2 (Slicing principle for references)
The lifetime of a reference contains all potential aliases of what the reference points-to.

Theorem 3.2. *Let:*
- σ *where* $\Sigma \vdash \sigma : \Gamma$
- p_{mut} *where* $\bullet; \Gamma \vdash_{\text{uniq}} p_{\text{mut}} \Rightarrow \{\bar{\ell}\}$ *and* $\sigma \vdash p_{\text{mut}} \Downarrow \pi_{\text{mut}}$
- p_{any} *where* $\sigma \vdash p_{\text{any}} \Downarrow \pi_{\text{any}}$

Then $\pi_{\text{any}} \sqcap \pi_{\text{mut}} \implies \exists^{\text{uniq}} p_{\text{loan}} \in \{\bar{\ell}\} \,.\, p_{\text{any}} \sqcap p_{\text{loan}}$

Rather than referring to a lifetime directly, we instead use Oxide's ownership safety judgment described in Section 3.2 to get the corresponding loan set for a mutated place expression p_{mut}. If p_{mut} includes a dereference, then the loan set should include potential aliases.

A notable detail is that we do not compare the loan sets of p_{mut} and p_{any} to see if they contain conflicting places, but rather compare p_{any} just against the loan set of p_{mut}. This works because the loan set contains not just the set of places π that p_{mut} could point-to, but also the set of other references to the places p_{mut} points-to (via reborrows).

Principle 3 (Slicing principle for function calls))
When calling a function:
(a) only mutable references in the arguments can be mutated, and …

Theorem 3.A. *Let:*
- $\Gamma, \pi_{\text{arg}}, \sigma$ *where* $\Gamma(\pi_{\text{arg}}) = \tau^{SI}$ *and* $\Sigma \vdash \sigma : \Gamma$
- f *where* $\Sigma \vdash (\sigma; f(\pi_{\text{arg}})) \xrightarrow{*} (\vec{\sigma}; \_)$
- $\vec{\sigma}' = \vec{\sigma}[\forall p_{\text{ref}} \in \text{uniq-refs}(\pi_{\text{arg}}, \tau^{SI}) \,.\, p_{\text{ref}} \mapsto \sigma(p_{\text{ref}})]$

Then $\sigma = \vec{\sigma}'$.

First, we define "mutable references in the arguments" as ω-refs(p, τ) that returns the ω-safe place expressions of references inside of p of type τ. For instance, if $x = 0$ and $y = (0, \&r\ \text{uniq}\ x)$

then uniq-refs$(y, (\text{u32}, \&r\ \text{uniq u32})) = \{*y.1\}$. The definition is:

$$\omega\text{-refs}(p, \tau^{\text{B}}) = \varnothing \qquad \omega\text{-refs}(p, (\tau_1^{\text{SI}}, \ldots, \tau_n^{\text{SI}})) = \bigcup_i \omega\text{-refs}(p.i, \tau_i^{\text{SI}})$$

$$\omega\text{-refs}(p, \&\rho\ \omega'\ \tau^{\text{XI}}) = \begin{cases} \{*p\} \cup \omega\text{-refs}(*p, \tau^{\text{XI}}) & \text{if } \omega' \lesssim \omega^{(7)} \\ \varnothing & \text{otherwise} \end{cases}$$

Then we define Theorem 3.A in the theme of a transaction: if all the changes to unique references in π_{arg} are rolled back, then the new stack is the same as the one before the function call. This means implicitly that no other values could have been mutated.

Principle 3 (Slicing principle for function calls)
When calling a function:
(b) …the mutations and return value are only influenced by the arguments.

Theorem 3.B. *Let:*
- $\Gamma, \pi_{\text{arg}}, \sigma^i$ *where* $\Gamma(\pi_{\text{arg}}) = \tau^{SI}$ *and* $i \in \{1, 2\}$ *and* $\Sigma \vdash \sigma^i : \Gamma$
- f *where* $\Sigma \vdash (\sigma^i; f(\pi_{\text{arg}})) \xrightarrow{*} (\vec{\sigma}^i; v^i)$
- $P = \text{all-places}(\pi_{\text{arg}}, \tau^{SI})$

Then $\sigma^1 \sim_P \sigma^2 \implies \vec{\sigma}^1 \sim_P \vec{\sigma}^2 \wedge v^1 = v^2$

The idea behind Theorem 3.B is that "influence" is translated into a form of noninterference: if the input to a function is the same under any two stacks σ^1 and σ^2, then the mutations to that input must be the same. The rest of the stack is allowed to vary, but because the function f cannot read it, that variation cannot influence the final value.

To formalize "the input being the same", we introduce another auxiliary function for transitive equality. For instance, if we only required that $\sigma^1(\pi_{\text{arg}}) = \sigma^2(\pi_{\text{arg}})$ where $\pi_{\text{arg}} = \text{ptr } x$, then if $\sigma^1(x) \neq \sigma^2(x)$ the theorem would not be true. Hence, transitive equality is defined as equality including all pointed places. We define this concept through two pieces: a function for generating the set of places (denoted by P), and a relation defining the equivalence of stacks for a set of places.

$$\text{all-places}(p, \tau^{\text{SI}}) \stackrel{\text{def}}{=} \{p\} \cup \text{shrd-refs}(p, \tau^{\text{SI}})$$

$$\sigma^1 \sim_P \sigma^2 \stackrel{\text{def}}{=} \forall p \in P\ .\ \sigma^1(p) = \sigma^2(p)$$

Therefore Theorem 3.B states that if π_{arg} is transitively equal under two otherwise arbitrary stacks, then π_{arg} is still transitively equal after evaluating $f(\pi_{\text{arg}})$, and the output of $f(\pi_{\text{arg}})$ is also equal.

4 ALGORITHM

Now, we use the formalized principles to build an algorithm for slicing Oxide. A slicing algorithm canonically has the form of a function that takes as input a program and a slicing criterion, and outputs a subset of the program [Weiser 1984]. However, Abadi et al. [1999] observed that a slice is a particular perspective on a term's dependencies, and so an alternative view on slicing is as a system for tracking dependencies. The slicing calculus of Abadi et al. has a type judgment of the form $\Gamma \vdash e : (\tau, \kappa)$ where expressions have a type τ and set of dependencies κ. Then, type-checking a program involves computing the dependencies of all terms, and computing a slice is simply looking up the dependencies in the term's type.

Inspired by Abadi et al., we formulate our algorithm as an extension to the Oxide type system that calculates dependencies for all expressions and places in a program. First, we assume that all

[7] The relation $\omega' \lesssim \omega$ (again re-used from Oxide) is defined as uniq $\not\lesssim$ shrd and otherwise $\omega' \lesssim \omega$. This relation captures the concept that a unique reference can be used as a shared reference, but not the other way around.

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

T-U32

$$\frac{}{\Sigma; \Delta; \Gamma; \Theta \vdash n_l : \mathsf{u32} \bullet \{l\} \Rightarrow \Gamma; \Theta}$$

T-Let

$$\frac{\begin{array}{c}\Sigma; \Delta; \Gamma; \Theta \vdash e_1 : \tau_1^{\mathrm{SI}} \bullet \kappa_1 \Rightarrow \Gamma_1; \Theta_1 \qquad \Gamma; \Delta_1 \vdash \tau_1^{\mathrm{SI}} \lesssim \tau_a^{\mathrm{SI}} \Rightarrow \Gamma_1' \\ \Theta_1' = \Theta_1[\forall \pi^{\square}[x] \,.\, \pi^{\square}[x] \mapsto \kappa_1] \qquad \Sigma; \Delta; \text{gc-loans}(\Gamma_1', x : \tau_a^{\mathrm{SI}}); \Theta_1' \vdash e_2 : \tau_2^{\mathrm{SI}} \bullet \kappa_2 \Rightarrow \Gamma_2, x : \tau^{\mathrm{SD}}; \Theta_2\end{array}}{\Sigma; \Delta; \Gamma; \Theta \vdash \mathbf{let}\ x : \tau_a^{\mathrm{SI}} = e_1;\ e_2 : \tau_2^{\mathrm{SI}} \bullet \kappa_2 \Rightarrow \Gamma_2; \Theta_2}$$

T-Move

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

T-Assign

$$\frac{\begin{array}{c}\Sigma; \Delta; \Gamma; \Theta \vdash e : \tau^{\mathrm{SI}} \bullet \kappa_e \Rightarrow \Gamma_1; \Theta_1 \qquad \Gamma_1(\pi) = \tau^{\mathrm{SX}} \qquad (\tau^{\mathrm{SX}} = \tau^{\mathrm{SD}} \vee \Delta; \Gamma_1 \vdash_{\mathsf{uniq}} \pi \Rightarrow \{ {}^{\mathsf{uniq}}\pi\}) \\ \Delta; \Gamma_1 \vdash \tau^{\mathrm{SI}} \lesssim \tau^{\mathrm{SX}} \Rightarrow \Gamma' \qquad \Theta_1' = \Theta_1[\text{update-conflicts}\ (\Theta_1, \pi, \kappa_e)]\end{array}}{\Sigma; \Delta; \Gamma; \Theta \vdash \pi := e : \mathsf{unit} \bullet \varnothing \Rightarrow \Gamma'[\pi \mapsto \tau^{\mathrm{SI}}] \rhd \pi; \Theta_1'}$$

T-AssignDeref

$$\frac{\begin{array}{c}\Sigma; \Delta; \Gamma; \Theta \vdash e : \tau_n^{\mathrm{SI}} \bullet \kappa_e \Rightarrow \Gamma_1; \Theta_1 \qquad \Delta; \Gamma_1 \vdash_{\mathsf{uniq}} p : \tau_o^{\mathrm{SI}} \qquad \Delta; \Gamma_1 \vdash_{\mathsf{uniq}} p \Rightarrow \{\bar{\ell}\} \\ \Delta; \Gamma_1 \vdash \tau_n^{\mathrm{SI}} \lesssim \tau_o^{\mathrm{SI}} \Rightarrow \Gamma' \qquad \Theta_1' = \Theta_1[\forall\, {}^{\omega}p' \in \{\bar{\ell}\} \,.\, \text{update-conflicts}\ (\Theta_1, p', \kappa_e)]\end{array}}{\Sigma; \Delta; \Gamma; \Theta \vdash p := e : \mathsf{unit} \bullet \varnothing \Rightarrow \Gamma' \rhd p; \Theta_1'}$$

T-App

$$\frac{\begin{array}{c}\overline{\Sigma; \Delta; \Gamma \vdash \Phi} \qquad \overline{\Delta; \Gamma \vdash \rho} \qquad \overline{\Sigma; \Delta; \Gamma \vdash \tau^{\mathrm{SI}}} \\ \Sigma(f) = \mathbf{fn}\ f\langle \overline{\psi}, \overline{\varrho}, \overline{\alpha}\rangle(x : \tau_a^{\mathrm{SI}}) \to \tau_r^{\mathrm{SI}}\ \mathbf{where}\ \overline{\varrho_1 : \varrho_2}\ \{\ e\ \} \\ \Sigma; \Delta; \Gamma; \Theta \vdash \pi : \tau_a^{\mathrm{SI}}[\Phi/\varphi][\rho/\varrho][\tau^{\mathrm{SI}}/\alpha] \bullet \kappa \Rightarrow \Gamma_1; \Theta_1 \qquad \Delta; \Gamma_1 \vdash \overline{\varrho_2[\rho/\varrho] :> \varrho_1[\rho/\varrho]} \Rightarrow \Gamma_2 \\ \Theta_2 = \Theta_1[\forall p \in \text{uniq-refs}(\pi, \tau_a^{\mathrm{SI}}) \,.\, \Delta; \Gamma_2 \vdash_{\mathsf{uniq}} p \Rightarrow \{\bar{\ell}\}, \forall\, {}^{\omega}p' \in \{\bar{\ell}\} \,.\, \text{update-conflicts}\ (\Theta_1, p', \kappa)]\end{array}}{\Sigma; \Delta; \Gamma; \Theta \vdash f\langle \overline{\Phi}, \overline{\rho}, \overline{\tau^{\mathrm{SI}}}\rangle(\pi) : \tau_r^{\mathrm{SI}}[\overline{\Phi/\varphi}][\rho/\varrho][\overline{\tau^{\mathrm{SI}}/\alpha}] \bullet \kappa \Rightarrow \Gamma_2; \Theta_2}$$

Fig. 2. Modular slicing algorithm formulated as an extension to Oxide's static semantics. Dependencies for expressions are represented as $\bullet\, \kappa$, and dependencies for places are contained in Θ. Remaining rules are given in Appendix A.1.

Oxide expressions e are uniquely labeled with their location l (written as e_l). Second, we modify the type-checking judgment to include two new features: a dependency set $\kappa = \{\bar{l}\}$ for the value of an expression, and a dependency context $\Theta = \{\overline{\pi \mapsto \kappa}\}$ for the dependencies of places in memory. As a convention, we will highlight in red our additions to the rules, and leave in black the original logic from Weiss et al. [2019]. So the new judgment has the form:

$$\Sigma; \Delta; \Gamma; \Theta \vdash e : \tau \bullet \kappa \Rightarrow \Gamma'; \Theta'$$

This judgment reads as: "under global context Σ, function context Δ, type context Γ, and dependency context Θ, expression e has type τ and dependencies κ and produces a new type context Γ' and dependency context Θ'."

4.1 Rules

Figure 2 introduces a subset of the modified static semantics for dependency tracking. The general goal is that for an expression e, κ contains the set of dependencies needed to determine the value of just e. So for instance, if e has type unit, then $\kappa = \varnothing$ because it has no distinguishable values. Then for Θ we have a corresponding entry for every place π in Γ. As places are mutated, additional dependencies are added to $\Theta(\pi)$ based on the dependencies of the mutation's inputs.

The base case is that constructors initialize the slice set with their location. For example, with rule T-u32, a labeled number n_l has the single dependency $\{l\}$, and the dependency context Θ is unchanged. Appendix A.1 shows additional constructors like T-Tuple and T-Borrow.

Places are introduced in T-Let with two notable aspects: first, dependency contexts are threaded through expressions in the order of evaluation. So e_1 produces Θ_1 which is then passed to checking of e_2. Second, x is bound to e_1 with dependencies κ_1, so that fact is saved in the updated dependency context Θ_1' as $x \mapsto \kappa_1$. For compound objects, an entry in the dependency context is created for every path into x, represented as $\pi^{\square}[x]$. Then a read of a place π has the dependencies $\Theta(\pi)$ as in T-Move

T-Assign handles assignments to places π, i.e. without dereferences. The overall expression has unit value and hence has an empty dependency set. The main step is to update Θ to reflect the dependencies κ_e of e. The naive implementation $\Theta[\pi \mapsto \kappa_e]$ doesn't work because of Principle 1: all conflicting places are also mutated, where conflicts are formally represented as $p_1 \sqcap p_2$. Hence, we define a function update-conflicts that updates all the conflicts of a place expression p with κ in Θ:

$$\text{update-conflicts}\,(\Theta, p, \kappa) \stackrel{\text{def}}{=} \forall \pi \mapsto \kappa_\pi \in \Theta\,.\;(p \sqcap \pi \implies \pi \mapsto \kappa_\pi \cup \kappa)$$

T-AssignDeref generalizes to assignments with dereferences. As per Principle 2, any place pointed by p is contained in the loan set $\{\overline{\ell}\}$. Therefore this rule propagated the dependencies κ_e of e to every such place π in the dependency context Θ. Additionally, $\{\overline{\ell}\}$ contains reborrowed pointers of the form $*\pi$ which also get their dependencies updated.

Finally, function calls are handled by T-App.[8] Each place in the loan set of each unique reference in $\text{uniq-refs}(\pi, \tau^{\text{sl}})$ is updated with the dependencies of the argument κ. For example, in the program:

$$\texttt{let}\ x : \text{u32} = 0;\ \texttt{let}\ y : \&r\ \text{uniq}\ \text{u32} = \&r\ \text{uniq}\ x;\ \texttt{let}\ z : (\text{u32}, \&r\ \text{uniq}\ \text{u32}) = (1_l, y);\ f(z)$$

The location l of the number 1 in z should be included in the dependencies of x, since f could use $z.0$ in mutating $*z.1$. This happens because $\text{uniq-refs}(z, (\text{u32}, \&r\ \text{uniq}\ \text{u32})) = \{*z.1\}$, and $\bullet; \Gamma \vdash_{\text{uniq}} *z.1 \Rightarrow \{x\}$. Then because $\kappa = \Theta(z)$ and $l \in \Theta(z)$, the T-App rule ensures that l is added to $\Theta(x)$.

4.2 Soundness

To characterize the correctness of our algorithm, we again draw on the insight of Abadi et al. [1999]: our algorithm is sound if it satisfies a form of noninterference. At a high-level, if an expression e has some dependencies κ, then for two stacks σ^1, σ^2, if those stacks are the same for at least κ, then e should evaluate to the same value whether it is run with σ^1 or σ^2. To define "same for at least κ", we first map from locations in κ to places in Θ as follows:

$$\text{deps}(\Theta, \kappa) \stackrel{\text{def}}{=} \{\pi \mid \pi \mapsto \kappa_\pi \in \Theta \wedge \kappa_\pi \subseteq \kappa\}$$

[8] Besides E-Assign, T-App is the only other rule that is significantly modified from the published version of Oxide. Originally, T-App permitted a list of argument expressions. For the sake of simplifying the slicing algorithm and proof, we restricted functions to only take a single argument which must be a place, not an expression. We also restricted functions to only be top-level, and not closures.

Then if $P = \text{deps}(\Theta, \kappa)$, two stacks are equivalent under P if $\sigma^1 \sim_P \sigma^2$. We will use the shorthand $\sigma^1 \sim_\kappa^\Theta \stackrel{\text{def}}{=} \sigma^1 \sim_{\text{deps}(\Theta,\kappa)} \sigma^2$.

Because our language has effects, noninterference must also account for them: for all places π in the stack, if the stacks are equivalent for the place's dependencies κ_π, then π should be the same in the output stacks after executing e. We formalize these statements as a noninterference theorem:

Theorem 4.1 (Noninterference). *Let e such that $\Sigma; \bullet; \Gamma; \Theta \vdash e : \tau \bullet \kappa \Rightarrow \Gamma'; \Theta'$. For $i \in \{1, 2\}$, let σ_i such that $\Sigma \vdash \sigma_i : \Gamma$ and $\Sigma \vdash (\sigma_i; e) \xrightarrow{*} (\vec{\sigma}^i; v^i)$. Then:*

(a) $\sigma^1 \sim_\kappa^\Theta \sigma^2 \implies v^1 = v^2$
(b) $\forall \pi \mapsto \kappa_\pi \in \Theta' \,.\, \sigma^1 \sim_{\kappa_\pi}^\Theta \sigma^2 \implies \vec{\sigma}^1(\pi) = \vec{\sigma}^2(\pi)$

Note that the context Δ is required to be empty because an expression e can only evaluate if it does not contain abstract type or provenance variables. The judgment $\Sigma \vdash \sigma^i : \Gamma$ means "the stack σ^i is well-typed under Σ and Γ". That is, all places π in Γ must exist and have values of the corresponding type in σ.

Additionally, we ignore the issue of non-termination in this definition. Distinguishing terminating from non-terminating programs is important in the context of security, but less so for slicing. So we make the simplifying assumption that the theorem only applies to terminating programs.

Proof sketch. The proof proceeds by induction on the derivation of the typing judgment. The full proof is in Appendix A.2, and here we will sketch the proof of part (b) on the key cases. Most of the novel pieces relate to effects on the stack, so the proof of (a) is not as interesting.

Let $\pi_{\text{any}} \mapsto \kappa_{\text{any}} \in \Theta'$. Assume the initial heaps from the two runs are equivalent under κ_{any}, i.e. $\sigma^1 \sim_{\kappa_{\text{any}}}^\Theta \sigma^2$. Then we want to show $\vec{\sigma}^1(\pi_{\text{any}}) = \vec{\sigma}^2(\pi_{\text{any}})$.

T-Assign: $e =$ "$\pi_{\text{mut}} := e$". First, $\sigma^1 \sim_{\kappa_{\text{any}}}^\Theta \sigma^2$ implies π_{any} is initially equal between the two runs, i.e. $\sigma^1(\pi_{\text{any}}) = \sigma^2(\pi_{\text{any}})$. Then by the inductive hypothesis on e, π_{any} is still equal after stepping in both runs (σ^1, e) to $(\vec{\sigma}_e^1, v_e^1)$ and (σ^2, e) to $(\vec{\sigma}_e^2, v_e^2)$. Finally, the runs conclude by updating the place π_{mut} to v_e^i. If π_{mut} has root variable x, i.e. $\pi_{\text{mut}} = \pi^\square[x]$, then updating π_{mut} is done by computing the value of x without π_{mut} as $\mathcal{V}^i$: $\vec{\sigma}_e^i \vdash \pi_{\text{mut}} \Downarrow \mathcal{V}^i$. Then the stack is updated by plugging v_e^i into the context: $\vec{\sigma}^i = \vec{\sigma}_e^i[x \mapsto \mathcal{V}^i[v_e^i]]$.

By Theorem 3.1, the only way π_{any} could have changed is if π_{mut} and π_{any} conflict, i.e. $\pi_{\text{mut}} \sqcap \pi_{\text{any}}$ By the premises of T-Assign, then $\Theta'(\pi_{\text{any}}) = \Theta(\pi_{\text{any}}) \cup \kappa_e$ where κ_e are the dependencies of e. Because $\kappa_e \subseteq \Theta'(\pi_{\text{any}})$ then $\sigma^1 \sim_{\kappa_e}^\Theta \sigma^2$, which by the inductive hypothesis (a) on e means $v_e^1 = v_e^2$, and subsequently $\vec{\sigma}^1(\pi_{\text{any}}) = \vec{\sigma}^2(\pi_{\text{any}})$.

T-AssignDeref: $e =$ "$p_{\text{mut}} := e$". This proof is similar to T-Assign, except we must account for whether p_{mut} could point to a conflicting place with π_{any}. That is, where $\vec{\sigma}_e^i \vdash p_{\text{mut}} \Downarrow \pi_{\text{mut}}^i$ and $\pi_{\text{any}} \sqcap \pi_{\text{mut}}^i$. By Theorem 3.2, because π_{mut}^i and π_{any} conflict, then π_{any} must conflict with a place in the loan set of p_{mut}. Therefore by T-AssignDeref, $\kappa_e \subseteq \kappa_{\text{any}}$, and $\vec{\sigma}^1(\pi_{\text{any}}) = \vec{\sigma}^2(\pi_{\text{any}})$.

T-App: $e =$ "$f(\pi_{\text{arg}})$". Then $(\sigma^i, f(\pi_{\text{arg}})) \xrightarrow{*} (\vec{\sigma}^i, v^i)$. By Theorem 3.A, if $\sigma^i(\pi_{\text{any}}) \neq \vec{\sigma}^i(\pi_{\text{any}})$, then π_{any} conflicts with a unique reference p_{ref} projected from π_{arg}. Then by Theorem 3.2, π_{any} is in the loan set of p_{arg}, and by T-App then π_{arg} is registered as a dependency of π_{any}, i.e. $\kappa_{\text{arg}} \subseteq \kappa_{\text{any}}$. Because σ^1 and σ^2 are then equivalent up to κ_{arg}, then π_{arg} is initially equal between stacks. By Theorem 3.B, π_{arg} is also equal after executing f. Therefore no mutations could have occurred, and $\vec{\sigma}^1(\pi_{\text{any}}) = \vec{\sigma}^2(\pi_{\text{any}})$ □

5 IMPLEMENTATION

We have established that our modular slicing algorithm is sound, but it is still an approximation to the true slice. For example, a function could take a unique reference as input, but then not mutate it. Hence, our next goal is to evaluate the *precision* of our algorithm: how much larger are its slices than optimal?

We chose to address this question empirically, by generating slices on real-world codebases and comparing the slice sizes. To some extent, the practical precision of the slicer depends upon the frequency with which programmers write code where the slicer's modular assumptions are too strong. Therefore we needed an implementation of the slicing algorithm that works on real code. Using Oxide to verify the correctness of our algorithm, we then translated the key concepts into a slicer that works for Rust.

Rust imposes a number of logistical constraints that required us to adapt the ideas above such that they could be integrated with the Rust compiler. Below, we discuss the implementation details needed to meet these constraints. First, Rust computes lifetime information not on the surface language like Oxide, but on an LLVM-like intermediate representation called MIR (5.1). Second, Rust does not use the same loan-set model of lifetimes as Oxide, but rather a line-set model (5.2). Finally we describe a few details necessary to execute the slicer on Rust programs (5.3).

5.1 MIR

The Rust compiler lowers programs into a "mid-level representation", or MIR, that represents programs as a control-flow graph. Essentially, expressions are flattened into sequences of instructions (basic blocks) which terminate in instructions that can jump to other blocks, like a branch or function call.

To implement the modular slicing algorithm for MIR, we largely reused standard techniques for analyzing CFGs. Specifically, we implemented a dataflow analysis that takes as input a slicing criterion, and finds all relevant instructions in the CFG to the criterion by maintaining a set of relevant places. The analysis is:

- *Backwards,* e.g. if p is determined to be relevant after $p := e$, then e is registered as relevant before $p := e$.
- *Flow-sensitive,* i.e. a different set of relevant places is maintained at each instruction.
- *Gen-kill,* i.e. each instruction is processed by a transfer function that either adds or removes places in the relevant place set.

The analysis is iterated to a fixpoint over the set of relevant places. The transfer function largely implements the same transformations as the algorithm in Section 4. For instance, the transfer function for assignment instructions, shown here in pseudocode, looks similar to T-AssignDeref.

```
let instruction = (p := e);
if ∃ p1 ∈ loans(p), p2 ∈ relevant set . p1 ⊓ p2 {
  relevant set ∪= places(e)
}
```

To handle conditional effects as described in Section 2.1, we compute the control-dependence between instructions. If a relevant instruction is control-dependent on a branch, then the branch is also relevant. We define control-dependence following Ferrante et al. [1987]: an instruction I_X is control-dependent on I_Y if there exists a directed path P from I_X to I_Y such that any I_Z in P is post-dominated by I_Y, and I_X is not post-dominated by I_Y. An instruction I_X is post-dominated by I_Y if I_Y is on every path from I_X to a `return` node. We compute control-dependencies by generating the post-dominator tree and frontier of the CFG using the algorithms of Cooper et al. [2001] and Cytron et al. [1989], respectively.

```
1  fn get_count(
2    h: &mut HashMap<String, u32>,
3    k: String
4  ) -> u32 {
5    if !h.contains_key(&k) {
6      h.insert(k, 0);
7      0
8    } else {
9      *h.get(&k).unwrap()
10   }
11 }
```

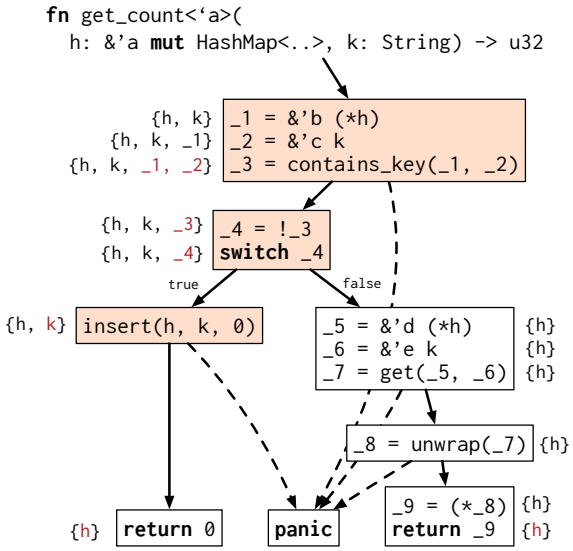


Fig. 3. Example of a Rust function (left) and corresponding MIR graph (right) sliced on the variable h at the end of the function. The MIR graph shows the result of the slicing dataflow analysis after reaching a fixpoint. Relevant MIR instructions are highlighted in orange, and the relevant place sets are adjacent to instructions with additions highlighted in red.

Figure 3 shows an example of the Rust slicer in action. First, it demonstrates how the function `get_count` is lowered from Rust to MIR. For instance, the condition on `contains_key` is lowered to a `switch` statement. Second, it shows how a relevant place set is maintained at each node, and that instructions which modify the place set are considered relevant.

5.2 Lifetimes

Because Rust does not compute loans for lifetimes, we had to regenerate the loans ourselves. The key observation is that both Oxide's borrow checker and Rust's borrow checker share a common ingredient: subtyping in the form of outlives-constraints. An outlives-constraint is a relation between lifetimes. Constraints are generally created either implicitly via subtyping, or explicitly via user-providec constraints in a function's signature. For instance, consider an assignment like `let x: &'a i32 = &'b y`. In both languages, type-checking this assignment requires `&'b i32` $\lesssim$ `&'a i32` which in turn requires `'b : 'a`.

In Oxide, the act of proving subtyping between lifetimes causes the loan sets to change. Specifically, Rule OL-LocalProvenances [2019, p. 14] states that proving r_1 :> r_2 updates $\Gamma(r_2) = \Gamma(r_1) \cup \Gamma(r_2)$. From this rule, we can generalize to the principle: if `loans('b)` $\subseteq$ `loans('a)` then `'b : 'a`. Hence, we modified the Rust compiler to export the set of outlives-constraints generated by a program, and computed the reflexive-transitive closure to generate a complete list of outlives-constraints. Then, we compute the loans for a lifetime as follows:

```
loans('a) = {p} if expression "&'a p" exists in the program
loans('a) = ⋃'b : 'a loans('b)
```

Fig. 4. Screenshot of our VS Code extension that uses the modular slicer to highlight relevant lines of code for a given criterion. The variable `input` is sliced on line 14.

5.3 Running the slicer

To slice a given Rust program, we take as input a slicing criterion, then run the Rust compiler to generate MIR and lifetime constraints for the function containing the criterion. We then run the slicing dataflow analysis seeded by the criterion.

For the evaluation, we simply count the number of relevant instructions for a given slice. For usage of the tool beyond the context of the paper, we have also developed an IDE extension for VSCode that source-maps relevant instructions back to spans of code in the surface program. Figure 4 shows an example program sliced within the GUI.

Our fork of the Rust compiler (to export lifetime outlives-constraints) and our modular slicer are both free and open-source. A GitHub link will be provided upon publication, and for double-blind review, an anonymized copy of the source code is provided in the supplementary materials.

6 EVALUATION

To evaluate the precision of modular slicing, we ran the modular Rust slicer over a dataset of Rust codebases. We then compared the size of slices for a given criterion against a series of alternatives. Specifically, we evaluate in two directions:

(1) What if the algorithm had *more* information? If the slicer had access to the definition of called functions, how much more precise are the whole-program slices vs. an equivalent modular slice?
(2) What if the algorithm had *less* information? If Rust's type system was more like C, i.e. it lacked ownership, then how much less precise do the modular slices become?

To answer these questions, we modified the slicer described in Section 5 in three ways:

- WHOLE-PROGRAM: the slicer will recursively slice function calls. For example, if calling a function `f(&mut x)` where `f` does not actually modify `x`, then the recursive slicer will recognize `x` is not mutated while the modular slicer will assume that `x` is mutated.

- Mut-blind: the slicer does not distinguish between mutable and immutable references. For example, if calling a function `f(&x)`, then the slicer assumes that `x` can be modified.
- Reference-blind: the slicer does not use lifetimes to reason about references, and rather assumes all references of the same type can alias. For example, if a function takes as input `f(x: &mut i32, y: &mut i32)` then `x` and `y` are assumed to be aliases.

The Mut-blind and Reference-blind modifications represent an ablation of the precision provided by ownership types. The Whole-program modification represents the most precise slicer we can feasibly implement. Each modification can be combined with the others, representing $2^3 = 8$ possible conditions for evaluation.

Whole-program requires some clarification on its implementation: it works by transferring slicing criteria from the call site to the function definition. For example, if a slice in function `f` calls function `g(&mut y)` where `y` is relevant, then the slicer checks whether `g` ever modifies its first input, and consider the call relevant if so.

Due to the architecture of the Rust compiler, it is difficult to analyze more than one package, or "crate", at a time. Hence Whole-program can only recurse *within a given package*, and not into the standard library or other dependencies. If Whole-program reaches a function call that cannot be recursed, then it falls back onto the modular slicer. This also includes situations like higher-order functions where no definition is available.

With these three modifications, we compare the size of slices (6.1) on a dataset of Rust projects (6.2) to quantatively (6.3) and qualitatively (6.4) evaluate the precision of the modular slicer.

6.1 Metric

To compare the slices generated in different conditions, we need a metric for precision. As an example, consider this slice on a vector:

```
1 // Modular slice
2 let mut v = vec![1, 2];
3 let x = v.get_mut(0).unwrap();
4 println!("{:?} {}", v, *x);
```

```
1 // Whole-program slice
2 let mut v = vec![1, 2];
3 let x = v.get_mut(0).unwrap();
4 println!("{:?} {}", v, *x);
```

The function `Vec::get_mut` returns a mutable reference to an element of the vector. Because it takes as input `&mut self`, the modular slicer assumes that `get_mut` mutates `v`, even though it is simply passing the mutable permission to an inner element. Therefore the modular slice (left) is larger than the whole-program slice (right).

To measure the difference between these slices, we need to compute and compare their sizes. The modular slice has 2 lines and the true slice has 1 line. To control for the size of the program being sliced, we use the percentage difference between the slice sizes: here, the modular slice is 100% larger. For Rust slices generated on the MIR, we compare the number of instructions within each slice.

6.2 Dataset

To create a dataset of slices, we first needed a dataset of Rust crates to slice. We had two main selection criteria: first, due to the single-crate limitation of Whole-program, we preferred large crates so as to see a greater impact from Whole-program, specifically crates with over 10,000 lines of code as measured by the `cloc` utility [Danial 2021]. Second, to control for code styles specific to individual applications, we wanted crates from a wide range of domains.

After a manual review of large crates in the Rust ecosystem, we selected 10 crates, shown in Table 1. We built each crate with as many feature flags enabled as would work on our Ubuntu 16.04 machine. Additional details like the specific flags and exact commit hashes used can be found in

Project	Crate	Purpose	LOC	# Slices	Avg. Slices/Func
rayon		Data parallelism library	15,524	10,535	9.9
Rocket	core/lib	Web backend framework	10,688	12,615	17.0
rustls	rustls	TLS implementation	16,866	23,658	27.3
sccache		Distributed build cache	23,202	24,673	38.3
nalgebra		Numerics library	31,951	37,143	20.8
image		Image processing library	19,718	38,165	36.4
hyper		HTTP server	15,082	39,981	50.6
rg3d		3D game engine	54,426	60,798	17.6
rav1e		Video encoder	50,294	74,667	86.4
RustPython	vm	Python interpreter	48,124	94,722	28.9
		Total:	285,975	416,957	

Table 1. Dataset of crates used to evaluate slicer precision, ordered in increasing number of slices. Each project often contains many crates, so a sub-crate is specified where applicable, and the root crate is analyzed otherwise.

Appendix A.3. For each crate, we ran the slicer on every function. For a given function with n local variables, we generated n slicing criteria for the value of each variable at all return points of the function. For each criterion, we ran the modular slicer under each of the eight conditions. Finally, for each condition, we computed the size of the computed slice in the number of MIR instructions.

Given the crate-level limitation of Whole-program discussed at the beginning of Section 6, we also wanted to estimate the impact of this limitation. That is, if a hypothetical tool *could* recurse into every dependency, would precision be substantively improved? We cannot answer the question directly due to the Rust compiler. However as a proxy, for each slice with the Whole-program modification, we computed whether or not the slicer ever reached a crate boundary. Then we compared the slice size differences between slices that reached boundaries and those that did not.

Efficiency was not a primary concern for our evaluation because Whole-program is not a truly fair comparison. Whole-program is the most context-sensitive possible slicer, whereas many interprocedural slicing algorithms improve efficiency by using context-insensitive approximations that e.g. only analyze a function once by grouping together multiple call-sites to the same function. But the execution times were nonetheless small — the median execution time for the unmodified slicer was 109μsec (max 1.1sec), and for the fully modified slicer was 194μsec (max 2.9sec).

6.3 Quantitative results

We observed no meaningful patterns from the interaction of modifications, e.g. Mut-blind and Reference-blind being active simultaneously versus separately. So to simplify our presentation of results, we focus only on four conditions: three for each modification individually active with the rest disabled, and one for all modifications disabled, referred to as Modular.

6.3.1 Whole-program. Our first evaluation question is: how does Modular compare against a slicer with more information, i.e. Whole-program? To answer this, Figure 5 shows the slice size differences between Modular and Whole-program. Figure 5-left shows that the vast majority of differences are close to 0. Using a log-log scale to magnify the countours of the distribution, Figure 5-center shows that the data breaks down into two categories: % diff = 0, where the slices are exactly the same, and % diff > 0, where Modular is larger by some amount. A significant majority of slices are exactly the same: 95.4% of slices have % diff = 0.

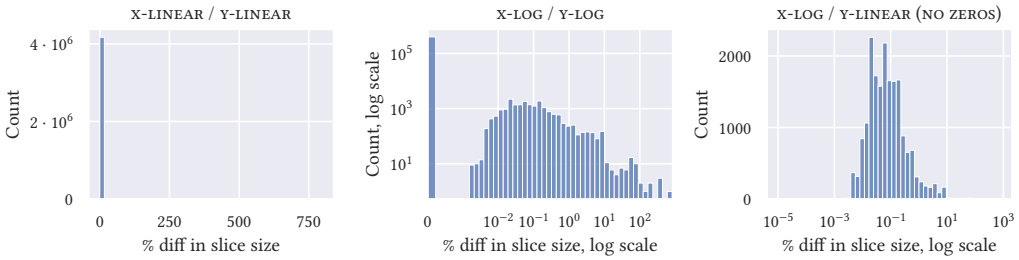


Fig. 5. Histogram of slice size differences between Whole-program and Modular. Left: linear/linear axes, plot is mostly empty because values close to 0 dominate. Middle: log/log axes, with a bar just for zero values at the far left. Right: log/linear axes, but zero-values removed from the plotted distribution.

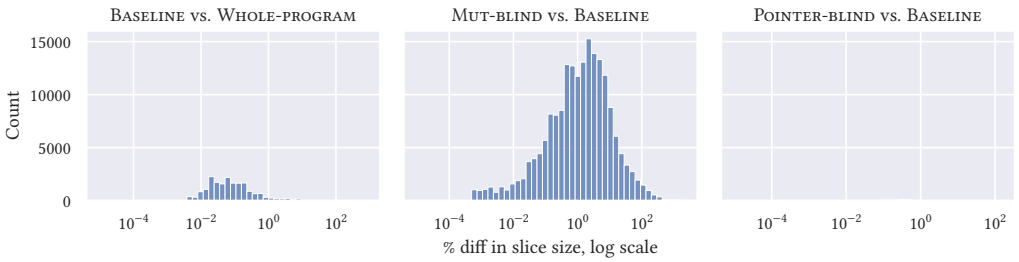


Fig. 6. Distribution of non-zero slice size differences between each modification condition and Modular. Plots share axes to compare relative frequency and magnitude of non-zero slices. Mut-blind slices very often differ from Modular, Whole-program significantly less so, and Reference-blind almost never (infrequently enough that they do not show up on for this shared y-axis).

Focusing on the non-zero cases, Figure 5-right shows the distribution of non-zero data on a linear y-axis. These 4.6% of slices are log-normally distributed with a geometric mean of 7.9%, lying around 10^{-1} on the log scale. These statistics indicate that the modular approximation is equally precise as inspecting definitions in nearly every slice. And when the approximation is less precise, the difference is relatively small – only 7.9% on average.

In the Whole-program condition, 66.3% of slices reached a crate boundary, suggesting that the limitation does frequently affect slices. However, for boundary slices, 93.1% of them were the same size as the corresponding Modular slice, while for non-boundary slices, 99.5% of them were the same as Modular. This result suggests that exhaustive recursion didn't necessarily produce more precise slices in general compared to limited recursion.

6.3.2 Mut-blind and Reference-blind. Next, we answer the question: how does Modular compare against a slicer with less information, i.e. Mut-blind and Reference-blind? The slice size difference distributions exhibit the same zero vs. non-zero breakdown as with Whole-program, so we similarly describe them by the fraction of zero cases along with the distributio of non-zero cases shown in Figure 6. Mut-blind increased slice sizes in 45.8% of cases for a geometric mean of 100.3% (mean around 10^0 in the Figure 6-center) in such cases. By contrast, Reference-blind only increased slice sizes in 0.2% of cases by a geometric mean of 25.3%. Reference-blind made so

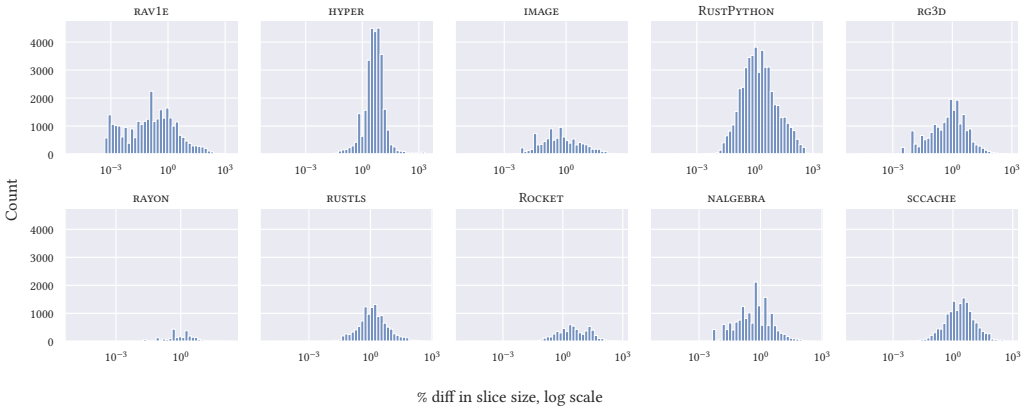


Fig. 7. Distribution of non-zero slice size differences between Mut-blind and Modular faceted by crate.

little impact that on a shared axis with the other conditions, its data is not visible in a histogram (Figure 6-right).

These results suggest that the distinction between immutable and mutable references is essential for the precision of the modular approximation. Ignoring this distinction increases slice sizes 2× in nearly half of all slices! The use of lifetimes for pointer analysis was comparatively unimportant for precision, as a naive aliasing assumption only changed a fraction of a percent of slices.

6.3.3 Variation between crates. To understand the variance in slice size differences between different crates, Figure 7 shows the distribution of non-zero data for Mut-blind vs. Modular per-crate. Some crates have substantively more non-zero slices than others. To some extent, this disparity is expected because each crate has a different number of slices, e.g. `RustPython` has 94k slices while `rayon` has 10k slices.

However, `hyper` and `image` have roughly the same number of slices, but `hyper` (72.4% of slices are non-zero) is significantly more affected by Mut-blind than `image` (27.2% non-zero). Based on manual inspection of functions with non-zero slices in each crate, one explanation is simply the differing code styles between application domains. For instance, many large `image` functions are format encoder/decoders that have primarily either mutable references to buffers or owned values copied out of those buffers, but no immutable references. By contrast, many large `hyper` functions involve e.g. validating HTTP requests that primarily use shared references. Therefore Mut-blind would be more likely to affect `hyper` functions than `image` functions.

6.4 Qualitative analysis

The statistics convey a sense of how often each condition influences slice sizes. But it is equally important to understand the kind of code that leads to such differences. For each condition, we manually inspected a sample of cases with non-zero slice size differences vs. Modular.

6.4.1 Whole-program vs. modular slices. As highlighted at the beginning on Section 6, one common difference between whole-program and modular slices is functions which take a mutable pointer as input for the purposes of passing the permission off to an element of the input.

```
fn crop<I: GenericImageView>(
  image: &mut I, x: u32, y: u32,
  width: u32, height: u32
) -> SubImage<&mut I> {
  let (x, y, width, height) =
    crop_dimms(image, x, y, width, height);
  SubImage::new(image, x, y, width, height)
}
```

For example, the function `crop` from the `image` crate returns a mutable view on a sub-image of a bigger image. No data is mutated, but rather the mutable permission is passed from whole image to sub-image. However, a modular slice on the `image` input would assume that `image` is mutated by `crop`.

Another difference arises when a value only depends on a subset of a function's inputs. A modular slice assumes all inputs are relevant to all mutations, but this is naturally not always the case.

For example, this function from `nalgebra` returns a boolean whose value solely depends on the argument `diag`. However, as per the modular approximation in Principle 3, a modular slice on a call to this function would assume that all inputs are relevant to the boolean

```
fn solve_lower_triangular_with_diag_mut<R2, C2, S2>(
  &self, b: &mut Matrix<N, R2, C2, S2>, diag: N,
) -> bool {
  if diag.is_zero() {
    return false;
  }
  // function logic with no returns...
  true
}
```

The final common case arises from unsafe code. As discussed in Section 2.4, the pattern of interior mutability uses unsafe code to hide mutations behind an immutable reference. But another kind of unsafe code came from API designers using mutable references to indicate that mutations were occurring in external C libraries.

```
fn draw(&mut self, state: &mut State, ...) {
  let fbo: GLuint = self.id;
  state.set_framebuffer(fbo);
  unsafe { gl::DrawElements(...); }
  // many other statements elided...
}
```

For example, consider this heavily simplified function from the `rg3d` game engine. This engine uses OpenGL to draw to the screen, and OpenGL uses a implicitly stateful API. Here, a frame buffer carries an identifier that is passed to OpenGL via `set_framebuffer`. Then `DrawElements` draws into the previously bound framebuffer. Importantly, `draw` does *not* have to be `&mut self`, because the only ownership-relevant action is a read to `self.id`. However, the API designer chose to use `&mut self` because it represents the fact that the OpenGL framebuffer is being mutated.

Therefore, a whole-program slice on `framebuffer.draw(..)` would consider `framebuffer` not to be mutated, and `draw` would not be part of the slice. By contrast, the modular slice would include `draw` in the slice because of the `&mut self` input type. From a strict perspective, the Whole-program slice is sound and more precise, because no value *defined within Rust* is changed as a result of `draw`. However, from a system-level perspective, the Modular slice is correct and the Whole-program slice is unsound because the value *represented* by `framebuffer`, the screen, is being mutated.

6.4.2 Mutability. The reason Mut-blind produces many larger slices than Modular is quite simple — many functions take immutable references as inputs, and so many more mutations have to be assumed. Importantly, due to higher-order functions, this issue could not be solved just with a whole-program analysis.

```rust
fn read_until<R, F>(io: &mut R, func: F)
  -> io::Result<Vec<u8>>
  where R: Read, F: Fn(&[u8]) -> bool
{
  let mut buf = vec![0; 8192];
  let mut pos = 0;
  loop {
    let n = io.read(&mut buf[pos..])?;
    pos += n;
    if func(&buf[..pos]) { break; }
    // ..
  }
}
```

For instance, this function from `hyper` repeatedly calls an input function `func` with segments of an input buffer. Without a control-flow analysis, it is impossible to know what functions `read_until` will be called with. And so Mut-blind must always assume that `func` could mutate `buf`. However, Modular can rely on the immutablity of shared references and deduce that `func` could not mutate `buf`.

6.4.3 Lifetimes. Surprisingly, information about aliases from lifetimes seemed to make almost no impact on slice sizes. One hypothesis is that it is simply uncommon to have multiple mutable references of the same type within a given function, and so the lifetime-based pointer analysis is not useful most of the time.

However, such cases do nonetheless arise. For instance, in `rg3d` this function takes two arguments of type `&mut FbxComponent`. Most of the code modifies the `parent` argument. However, a naive pointer analysis must assume that `parent` and `child` could alias, and therefore mutating `parent` also mutates `child`. Therefore a Reference-blind slice on `child` would unnecessarily include these mutations that would be avoided in Modular.

```rust
fn link_child_with_parent_component(
  parent: &mut FbxComponent,
  child: &mut FbxComponent,
  child_handle: Handle<FbxComponent>,
) { match parent {
  FbxComponent::Model(model) => {
    model.geoms.push(child_handle),
  },
  // ..
}}
```

7 RELATED WORK

Historically, the primary technique for backwards static slicing has been analysis of system dependence graphs (SDG), introduced by Horwitz et al. [1988]. SDGs encode the call graph of a program to enable efficient interprocedural (whole-program) static analysis. SDGs are combined with a pointer analysis like the methods of Andersen [1994] or Steensgaard [1996] to handle indirect mutations and function calls. For instance, the recent `dg` slicer for LLVM uses these techniques [Chalupa 2016].

Fundamentally, prior work on slicing has been limited by the requirement to analyze existing languages like C and Java. By focusing on newer languages with more expressive type systems, we are able to build on three lines of research: modular static analysis, ownership types, and information flow.

7.1 Modular static analysis

The key technique to modularizing static analysis is the ability to symbolically summarize a function's behavior. Rountev et al. [1999] and Cousot and Cousot [2002] pioneered this technique of symbolic procedure summaries. Later work built on this framework for a variety of static analyses. Gulwani and Tiwari [2007] create symbolic summaries for the domains of unary uninterpreted functions and linear arithmetic. Yorsh et al. [2008] designed a framework of micro-transformers which they applied to null-checking and typestate. Dillig et al. [2011] apply procedure summaries to perform a modular pointer analysis on C and C++ programs.

From this perspective, ownership types and their corresponding inference/checking are just another system of symbolic procedure summaries. As discussed in Section 2.3, lifetime parameters

are the key mechanism that enable symbolic expression of dataflow within a function's type signature.

7.2 Ownership types and Rust

Rust and Oxide's conceptions of ownership derive from Clarke et al. [1998] and Grossman et al. [2002]. For instance, the Cyclone language of Grossman et al. uses regions to restrict where a pointer can point-to, and region variables to express relationships between regions in a function's input and output. A lifetime is similar in that it annotates the types of pointers, but differs in how it is analyzed.

Matsakis [2018] first observed that Rust's lifetimes could be analyzed and interpreted as a set of loans rather than a set of lines. Ongoing efforts in the Rust community seek to integrate this perspective into the Rust compiler to reduce the number of valid-but-rejected programs. Oxide [Weiss et al. 2019] recently formalized this set-of-loans model, providing the basis for our work.

Recent work have demonstrated innovative applications of Rust's type system for modular program analysis. Astrauskas et al. [2019] embed Rust programs into a separation logic to verify pre/post conditions about functions. Jung et al. [2020] use Rust's ownership-based guarantees to implement provably correct program optimizations.

7.3 Information flow

Information flow has been historically studied in the context of security, such as ensuring low-security users of a program cannot infer anything about its high-security internals. However, information flow is fundamentally a generalization of dataflow to consider control-flow and effects, both relevant to slicing. The connection between information flow and slicing was first established in Abadi et al. [1999], where they showed that a slicing calculus could be embedded in a more general dependency calculus (because dependency sets, like security labels, form a lattice).

Hence, our work bears some resemblance to prior systems for information flow inference in the presence of effects, such as Pottier and Simonet [2003]. One notable difference is that our slicing algorithm has to do more work to track updates to dependency sets across effects. An information flow system needs only to ensure e.g. for $p := e$ that p cannot be low-security when e is high-security. By contrast, our system must propagate the dependencies of e to all possible pointees of p.

8 CONCLUSION

We have demonstrated that ownership can be leveraged to build a modular slicing algorithm that is both provably sound and reasonably precise in practice: equivalent to a whole-program slicer in 95.4% of cases, and only 7.6% larger in the remaining cases. In the short-term, these results suggest that our modular slicer could be a practical program comprehension tool for Rust programmers. To that end, we hope that our interactive slicer shown in Figure 4 can help users understand Rust code.

More broadly, we are excited by the prospects of repurposing ownership for other forms of static analysis. For instance, by formalizing our slicer as a general algorithm for tracking dependencies, our work could be easily applied to other domains of information flow such as security types. Prior work on information flow often needed to restrict the underlying language — for instance, the MLIF language of Pottier and Simonet [2003] could not be a drop-in replacement for all existing OCaml programs, since critical features like security-level polymorphism must be integrated manually by the programmer.

However, as discussed in Section 2.3, Rust programs already use lifetime-polymorphism to propagate dataflow across function boundaries for modular ownership analysis. We were able

to implement our modular slicer without any changes to the Rust language or to existing Rust programs. And unlike analyzing C or C++, there are no unpredictable expressiveness cliffs (as the cliffs are explicitly demarcated by `unsafe`). By restricting the space of permissible programs with ownership-safety, we simultaneously expand the space of program analyses that were previously intractable in more expressive languages.

REFERENCES

Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G Riecke. 1999. A core calculus of dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 147–160.

Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language*. Ph.D. Dissertation. Citeseer.

Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J Summers. 2020. How do programmers use unsafe rust? *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–27.

Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J Summers. 2019. Leveraging Rust types for modular specification and verification. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–30.

Gogul Balakrishnan, Radu Gruian, Thomas Reps, and Tim Teitelbaum. 2005. Codesurfer/x86—a platform for analyzing x86 executables. In *International Conference on Compiler Construction*. Springer, 250–254.

Marek Chalupa. 2016. *Slicing of LLVM bitcode*. Master's thesis. Masaryk University.

David G Clarke, John M Potter, and James Noble. 1998. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 48–64.

Keith D Cooper, Timothy J Harvey, and Ken Kennedy. 2001. A simple, fast dominance algorithm. *Software Practice & Experience* 4, 1-10 (2001), 1–8.

Patrick Cousot and Radhia Cousot. 2002. Modular static program analysis. In *International Conference on Compiler Construction*. Springer, 159–179.

Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2012. Frama-c. In *International conference on software engineering and formal methods*. Springer, 233–247.

Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1989. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 25–35.

Al Danial. 2021. cloc: Count Lines of Code. https://github.com/AlDanial/cloc

Isil Dillig, Thomas Dillig, Alex Aiken, and Mooly Sagiv. 2011. Precise and compact modular procedure summaries for heap manipulating programs. *ACM SIGPLAN Notices* 46, 6 (2011), 567–577.

Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, 3 (1987), 319–349.

Jean-Yves Girard. 1987. Linear logic. *Theoretical computer science* 50, 1 (1987), 1–101.

Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. 2002. Region-based memory management in Cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*. 282–293.

Sumit Gulwani and Ashish Tiwari. 2007. Computing procedure summaries for interprocedural analysis. In *European Symposium on Programming*. Springer, 253–267.

Susan Horwitz, Thomas Reps, and David Binkley. 1988. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*. 35–46.

Ganeshan Jayaraman, Venkatesh Prasad Ranganath, and John Hatcliff. 2005. Kaveri: Delivering the indus java program slicer to eclipse. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 269–272.

Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2020. Stacked borrows: an aliasing model for Rust. *Proceedings of the ACM on Programming Languages* 4, POPL (2020), 1–32.

Niko 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

Matthew Might, Yannis Smaragdakis, and David Van Horn. 2010. Resolving and exploiting the k-CFA paradox: illuminating functional vs. object-oriented program analysis. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 305–315.

Chris Parnin and Alessandro Orso. 2011. Are automated debugging techniques actually helping programmers?. In *Proceedings of the 2011 international symposium on software testing and analysis*. 199–209.

François Pottier and Vincent Simonet. 2003. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 25, 1 (2003), 117–158.

Atanas Rountev, Barbara G Ryder, and William Landi. 1999. Data-flow analysis of program fragments. In *Software Engineering—ESEC/FSE'99*. Springer, 235–252.

Josep Silva. 2012. A vocabulary of program slicing-based techniques. *ACM computing surveys (CSUR)* 44, 3 (2012), 1–41.

Yannis Smaragdakis and George Balatsouras. 2015. Pointer analysis. *Foundations and Trends in Programming Languages* 2, 1 (2015), 1–69.

Bjarne Steensgaard. 1996. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 32–41.

Mads Tofte and Jean-Pierre Talpin. 1997. Region-based memory management. *Information and computation* 132, 2 (1997), 109–176.

Mark Weiser. 1982. Programmers use slices when debugging. *Commun. ACM* 25, 7 (1982), 446–452.

Mark Weiser. 1984. Program slicing. *IEEE Transactions on software engineering* 4 (1984), 352–357.

Aaron Weiss, Olek Gierczak, Daniel Patterson, Nicholas D Matsakis, and Amal Ahmed. 2019. Oxide: The essence of rust. *arXiv preprint arXiv:1903.00982* (2019).

Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. 2005. A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes* 30, 2 (2005), 1–36.

Greta Yorsh, Eran Yahav, and Satish Chandra. 2008. Generating precise and concise procedure summaries. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 221–234.

Jisheng Zhao, Michael G Burke, and Vivek Sarkar. 2018. Parallel sparse flow-sensitive points-to analysis. In *Proceedings of the 27th International Conference on Compiler Construction*. 59–70.