Oxidize: A Step-Debugger for Static Semantics

Peter Chon
Harvard University
Cambridge, Massachusetts, USA
peterchon@college.harvard.edu

Dimi Racordon
Northeastern University
Boston, Massachusetts, USA
d.racordon@northeastern.edu

Nada Amin
Harvard University
Cambridge, Massachusetts, USA
namin@seas.harvard.edu

Abstract

Despite its success, Rust is a difficult language to learn, in particular because of its borrow checker. Part of the issue is a lack of understanding of the static semantics under which it operates. If we look at dynamic semantics, we observe that people can get this understanding using step-debugging. Therefore, we propose a static step-debugger to learn Rust's static semantics.

Keywords: pedagogy, lifetime analysis, static step-debugging

ACM Reference Format:
Peter Chon, Dimi Racordon, and Nada Amin. 2023. Oxidize: A Step-Debugger for Static Semantics. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 7 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Although it is estimated that memory safety issues are responsible for seventy percent of security vulnerabilities [4], memory-unsafe programming languages like C and C++ are still used to develop performance-critical software. Systems programmers, however, desire efficiency and control in a world where memory safety and garbage collection are colloquially (and mistakenly) synonymous and low-level systems are inherently unsafe. In response to these challenges, Rust has thrown its gauntlet: its type system. With its powerful ownership model, Rust promises memory safety without compromising execution efficiency.

Unfortunately, Rust has an infamous learning curve. A fundamental challenge is that Rust compels developers to reason statically about traditionally dynamic behavior of their programs. As Coblenz et al. notes, "[programmers] were not prepared to use a type system to address a bug that they thought of in a dynamic way" [1]. Especially for those accustomed to programming languages that automatically manage memory, Rust's arcane rules appear unmotivated.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
IWACO'23, October 22-27, 2023, Cascais, Portugal
© 2023 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

Efforts to lower this entry barrier include tutorials like Rustlings[5] which teach students about move semantics and borrow semantics by covering solutions to erroneous code examples. Furthermore, the Rust compiler will suggest fixes and link to explanations when compilation errors occur. However, there still is room for improvement. For example, the compiler, when analyzing dangling references, may suggest using static lifetimes (which denote that the reference will live forever) when the ideal solution is to rid the dangling reference. Not only will the suggested solution still raise an error, but also the users may not even know why dangling references are problematic.

Listing 1. Dangling Lifetime

```
fn f() -> &String {
    &String::from("dangling")
}

fn main() {
    let _msg = f();
}
```

Listing 2. Implemented Compiler Hints (still faulty)

```
fn f() -> &'static String {
    &String::from("temp")
}

fn main() {
    let _msg = f();
}
```

Listing 3. Correct Solution

```
fn f() -> String {
    String::from("dangling")
}

fn main() {
    let _msg = &f();
}
```

Incorrect solutions such as the ones above are particularly dangerous since the Rust compiler's capability is a double-edged sword; it is renowned for its mostly reliable enhanced compiler error messages with solutions and hints. Even in the typical case where the Rust compiler correctly identifies the solution to a bug, the solutions can still be unhelpful pedagogically. Programmers are tempted to blindly implement the suggestions [2], which makes the student "learn"

as much as they would by copying the solutions in a discrete math course. It is great that compilers do much of the thinking for programmers, but Rust's borrow and move semantics need to be understood to use the language.

With run-time behavior, step-debuggers have been shown to be useful for learning the semantics of imperative programming languages. When users are surprised by the run-time behavior of a program, step-debuggers allow users to pinpoint where and why unexpected results occur by examining the state of variables and the program flow. Dynamic step-debuggers help programmers think like the dynamic semantics of the source language. Likewise, with a static step-debugger simulating Rust's observance of the stepped-through environment, users would be well-served by being able to explicitly and more concretely reason with the compiler. Programmers need to understand how to think like the Rust compiler to write correct code productively, but this task has been found to be difficult for programmers coming from programming languages that do not make many compile-time guarantees on traditionally run-time behavior [3]. A static step-debugger would be even more beneficial for developers who haven't experienced the class of memory safety errors like segmentation faults and undefined behavior. To understand the seemingly arbitrary rules of Rust, the step-debugger would show such programmers why these memory safety errors even exist.

To develop a static-semantics step-debugger, we built Oxidize, a programming language that features a subset of the Rust programming language, along with a command-line tool to step through its static semantics. Oxidize features immutable and mutable assignments, references, tuples, and conditional expressions. Tuples can be used to demonstrate partial borrows and moves. Conditional expressions can be used to demonstrate why the Rust compiler will conservatively flag memory safety errors in cases that will never be reached at run-time. Oxidize also features dangling lifetimes through lexical scopes. We note that Oxidize cannot work on the examples found in Listings 1-3, as functions were not added yet.

The main novelty of our tool is that it should be usable by a beginner without external assistance, which is unlike RustViz which requires a person with familiarity with Rust [6]. RustViz can generate visualizations of ownership and borrowing events by requiring teachers to manually annotate code examples and write visual specifications for RustViz to analyze static semantics. While teachers can be expected to be familiar with Rust and its borrow checker, students need to already know how to explicitly reason about borrow events to use such visualization tools by themselves. On the other hand, users of Oxidize should be able to write some code and step through it. If their code is erroneous, the step-debugger will walk them through the error. Even if their code type checks, the step-debugger will show them how to know that their code satisfied the borrow checker.

A natural code example that users will step-through may naturally include terms of primitive types. Because primitive types are copy-able (implicitly cloned rather than moved) in Rust, users may not be able to observe a move event. Because we are trying to assume as little background as possible for our target user base, Oxidize does not feature the copy-able trait for pedagogical clarity. It contains what is necessary to learn how a borrow checker works. We also leave out magical convenience features like type-coercion, which would allow implicit dereferencing, reborrows, and reference downgrade coercions.

While this is not representative of code one would likely write, this motivating example uses crucial features like dereferencing. It also features the lack of copyable types.

```
let mut x = 3 in
    (let y = &mut x in
        *y = 5; x = 1);
    (let z = x in 1)
```

When the user runs the step-debugger on this code, they see a display of the context and which part of the code they are currently stepping through. Gamma stores the types, which includes mutability, for variable names. Mu maps variable names to memory locations. Eta tracks the moves and borrows.

```
Type Context (Gamma):
  x: mutable Int

Memory Map (Mu):
  x -> 0

Borrow Layout (Eta) :

--------------------------
let mut x = 3 in
```

Memory locations are generated as the successor to the maximal memory location, a natural number, that is currently in scope.

```
Type Context (Gamma):
  x: mutable Int
  y: immutable &mut Int

Memory Map (Mu):
  x -> 0
  y -> 1

Borrow Layout (Eta) :
  the values at addresses...
  0 is mutably referenced by address 1
--------------------------
let mut x = 3 in
    (let y = &mut x in
```

y mutably borrows the value at x. Thus, Eta notes that the value at address 0 is mutably referenced by address 1.

```
Type Context (Gamma):
  x: mutable Int
  y: immutable &mut Int

Memory Map (Mu):
  x -> 0
  y -> 1

Borrow Layout (Eta) :
  the values at addresses...
  0 is mutably referenced by address 1
--------------------------
let mut x = 3 in
    (let y = &mut x in
        *y = 5
```

Assignment to y is not allowed since y is not mutable. However, dereferencing y and changing the value at the address referred to by y is allowed.

```
Type Context (Gamma):
  x: mutable Int
  y: immutable &mut Int

Memory Map (Mu):
  x -> 0
  y -> 1

Borrow Layout (Eta) :
  the values at addresses...
  0 is mutably referenced by address 1
--------------------------
let mut x = 3 in
    (let y = &mut x in
        *y = 5; x = 1
--------------------------
At 56..61, TypeCheckError was raised: You can't
    assign to an address that is currently
    referred to.

Code snippet:
```
x = 1
```
```

While x is mutable, so mutating x would normally be allowed, x is mutably borrowed by y. Thus, the static semantics blocks any mutation directly through x. Any mutation must be through dereferencing of y.

2 Language Presentation

We now formally present Oxidize. First, we introduce the syntax, and then we introduce the static semantics.

2.1 Syntax

⟨*lhs*⟩ ::= ⟨*ident*⟩
  | *⟨*lhs*⟩
  | lhs.⟨*nat*⟩

⟨*atom*⟩ ::= '`unit`'
  | '`true`' | '`false`'
  | ⟨*nat*⟩
  | ⟨*lhs*⟩
  | '`(`'⟨*expr*⟩'`)`'

⟨*unary*⟩ ::= &⟨*lhs*⟩
  | &mut ⟨*lhs*⟩
  | [-]⟨*atom*⟩

⟨*product*⟩ ::= ⟨*unary*⟩ ('*' ⟨*unary*⟩)*
  | ⟨*unary*⟩ ('/' ⟨*unary*⟩)*

⟨*sum*⟩ ::= ⟨*product*⟩ ('+' ⟨*product*⟩)*
  | ⟨*product*⟩ ('-' ⟨*product*⟩)*

⟨*cmp*⟩ ::= ⟨*sum*⟩ ('<' ⟨*sum*⟩)*
  | ⟨*sum*⟩ ('>' ⟨*sum*⟩)*

⟨*cond*⟩ ::= '`if`' ⟨*cmp*⟩ '`then`' ⟨*cmp*⟩ '`else`' ⟨*cmp*⟩

⟨*tuple*⟩ ::= '`[`' ⟨*cmp*⟩ ('`,`' ⟨*cmp*⟩)* '`]`'

⟨*normal*⟩ ::= ⟨*cond*⟩
  | ⟨*tuple*⟩
  | ⟨*cmp*⟩

⟨*assign*⟩ ::= (⟨*lhs*⟩ '=')+ ⟨*normal*⟩

⟨*seq*⟩ ::= (⟨*assign*⟩ | ⟨*normal*⟩) ('`;`' ⟨*assign*⟩ | ⟨*normal*⟩)*

⟨*lets*⟩ ::= '`let`' ⟨*ident*⟩ '=' ⟨*normal*⟩ '`in`' ⟨*expr*⟩
  | '`let`' '`mut`' ⟨*ident*⟩ '=' ⟨*normal*⟩ '`in`' ⟨*expr*⟩

⟨*expr*⟩ ::= ⟨*lets*⟩
  | ⟨*seq*⟩

2.2 Semantics

We have the integer type $\mathbb{Z}$, the Boolean type bool, the unit type unit, and the product type $[\tau_i]_{i=1}^{l}$. We also have reference type Ref ⟨bool, τ⟩, which stores both mutability and the referent type.

A value at location $\mathcal{L}$ can be moved to another location, be borrowed by a reference at a location, or be dropped. A drop can be thought of as a move to an inaccessible location. There is no longer a way to use the value, and we now consider it "dropped." Because the borrow-checker is part of the static semantics, we define the following borrow states (**State$^{\mathbf{B}}$**s) to track moves and borrows.

Ref$^{\mathbf{B}}$ $(\{(\text{bool}_i, \ell_i)\}_i)$	the value at $\mathcal{L}$ is referenced by location ℓ_i through which it can be mutated if bool_i = true
Moved$^{\mathbf{B}}$ (ℓ)	the value at $\mathcal{L}$ is moved to location ℓ
Dropped$^{\mathbf{B}}$	the value at $\mathcal{L}$ is dropped
Union$^{\mathbf{B}}$ $([\textbf{State}^{\mathbf{B}}]_i)_i)$	all possible states any $s_i \neq$ **Union$^{\mathbf{B}}$**$(\{s_i\}_{i=1}^{n}$
None$^{\mathbf{B}}$	none of the above

For simplicity, we also define **State**, a limited version of $\textbf{State}^{\textbf{B}}$, which is used to propagate borrow-check information in our type system's rules. Note that the propagation makes reasoning about **State** better as a "mirrored" version of $\textbf{State}^{\textbf{B}}$ (see 2.2).

$\textbf{MutRef}(\ell)$	a mutable reference to ℓ
$\textbf{ImmutRef}(\ell)$	an immutable reference to ℓ
$\textbf{Moved}(\ell)$	value at location ℓ moved
$\textbf{Union}([\textbf{State}_i]_i)$	all possible states
None	none of the above

Our type rules are in the following form, where a type and a list of states are propagated.

Expr-*

$$\frac{\textbf{predicate(s)}}{\Gamma, \eta, \mu \vdash \textbf{term} : (\textbf{type},\ \textbf{S}),\ \Gamma, \eta, \mu}$$

Lhs-*

$$\frac{\textbf{predicate(s)}}{\Gamma, \eta, \mu \vdash_\ell \textbf{lhs} : (\textbf{type},\ \textbf{address}),\ \Gamma, \eta, \mu}$$

where

- $\Gamma : x \to (\text{bool}, \tau)$ maps variable names to their mutability and type,
- $\mu : x \to \ell$ maps variable names to the addresses they point to, and
- $\eta : \ell \to \textbf{State}^{\textbf{B}}$ maps memory addresses to their borrow state. For example, if $\eta(2) = \textbf{Ref}^{\textbf{B}}(\text{true}, 3)$, then the value at address 2 is mutably referenced by a reference at location 3.

We have the trivial type rules for such as the ones for 'true' and '$e_1 > e_2$.' We leave out other "obvious" rules for terseness.

E-True

$$\frac{}{\Gamma, \eta, \mu \vdash \text{ true} : (\text{bool}, \textbf{None}),\ \Gamma, \eta, \mu}$$

E-Gt

$$\frac{\Gamma, \eta, \mu \vdash e_1 : (\mathbb{Z}, _),\ \Gamma, \eta', \mu \qquad \Gamma, \eta', \mu \vdash e_2 : (\mathbb{Z}, _),\ \Gamma, \eta'', \mu}{\Gamma, \eta, \mu \vdash e_1 > e_2 : (\text{bool}, \textbf{None}),\ \Gamma, \eta'', \mu}$$

Lhs-Var

$$\frac{\Gamma(x) = (c,\ \tau) \qquad \ell = \mu(x)}{\Gamma, \eta, \mu \vdash_\ell x : (\text{Ref}\ \langle c, \tau \rangle, \ell),\ \Gamma, \eta, \mu}$$

Lhs-DeRef

$$\frac{\Gamma, \eta, \mu \vdash lhs : (\text{Ref}\ \langle _, \text{Ref}\ \langle c, \tau \rangle \rangle, \ell),\ \Gamma, \eta, \mu}{\Gamma, \eta, \mu \vdash_\ell {}^*lhs : (\text{Ref}\ \langle c, \tau \rangle, \ell),\ \Gamma, \eta, \mu}$$

Tuples are 0-indexed since we access elements starting from index 0, but the memory layouts are 1-indexed for representation purposes.

E-Tup

$$\frac{\eta'_0 = \eta \qquad \forall i \in [0, l) : \Gamma, \eta'_i, \mu \vdash e_i : (\tau_{i+1}, s_{i+1}),\ \Gamma, \eta'_{i+1}, \mu}{\Gamma, \eta, \mu \vdash [e_i]_{i=0}^{l-1} : \left([\tau_i]_{i=1}^{l}, [\textbf{None}, s_1, s_2, \ldots, s_l]\right),\ \Gamma, \eta', \mu}$$

Lhs-Index

$$\frac{\Gamma, \eta, \mu \vdash x : (\text{Ref}\left\langle c, [\tau_i]_{i=1}^{l} \right\rangle, \ell),\ \Gamma, \eta, \mu \qquad n \in [1, l]}{\Gamma, \eta, \mu \vdash_\ell x.n : (\text{Ref}\ \langle c, \tau_i \rangle, \ell + n),\ \Gamma, \eta, \mu}$$

Whenever an lvalue is evaluated, the borrow checker should flag its corresponding memory address as "dropped." Thus, we must first check whether the lvalue can indeed be dropped. Then, we actually drop it. For clarity's sake, we define the following helper functions.

(a) **maydrop: $\eta \to \ell \to \tau \to$ bool**
This helper function returns whether a value of type τ at address ℓ can be dropped. If τ is a product type, **maydrop** is recursively called on its elements. Let state $= \eta(\ell)$. If the state is $\textbf{Moved}^{\textbf{B}}$, $\textbf{Dropped}^{\textbf{B}}$, or $\textbf{Ref}^{\textbf{B}}$, the value at location ℓ cannot be dropped. If the state is $\textbf{Union}^{\textbf{B}}\textbf{(states)}$, every element of **states** must be droppable.

(b) **strip: $\textbf{State}^{\textbf{B}} \to \ell \to \textbf{State}^{\textbf{B}}$**
$b' = \textbf{strip}(b, \ell)$ returns a new b' that is a b with all occurrences of ℓ removed. $\textbf{Moved}^{\textbf{B}}(\ell)$ would become $\textbf{Dropped}^{\textbf{B}}$ because ℓ is to be inaccessible. If $b = \textbf{Ref}^{\textbf{B}}$(refs) where refs is a now an empty set, $\textbf{None}^{\textbf{B}}$ is returned.

(c) **drop: $\eta \to \ell \to \tau \to \eta$**
$\eta' = \textbf{drop}(\eta, \ell, \tau)$ returns a new map η' such that

$$\eta'(x) = \begin{cases} \textbf{Dropped}^{\textbf{B}} & x = \ell \\ \textbf{strip}(\eta(x)) & \text{otherwise} \end{cases}.$$

The point of disagreement between η and η' is the *drop*. The helper function **strip** is called to ensure that the dropped value is no longer referred to by any borrow states in η'. While ℓ is given to drop the correct value, τ is given because dropping tuples requires dropping every element of the tuple.

E-Lvalue

$$\frac{\Gamma, \eta, \mu \vdash_\ell \text{ lhs} : (\text{Ref}\ \langle _, \tau \rangle, \ell),\ \Gamma, \eta, \mu \qquad \textbf{maydrop}(\eta, \ell, \tau) = \text{true} \qquad \eta' = \textbf{drop}(\eta, \ell, \tau)}{\Gamma, \eta, \mu \vdash \text{ lhs} : (\tau, \textbf{Moved}(\ell)),\ \Gamma, \eta', \mu}$$

Whenever we assign to an lhs, we must first judge the expression of the assigned value. Then, we check for any live references to the memory address referred to by lhs, raising an error if we do find any. If there aren't any, we drop all borrow-information corresponding to lhs, as the other borrow states are irrelevant, and we now update η with new borrow states. To do this, we define a helper function.

(a) **mayoverwrite: $\eta \to \ell \to \tau \to$ bool**
This helper function works like **maydrop** with how it handles tuples and **Union^B**. However, **Moved^B** and **Dropped^B** are permissible. Only **Ref^B** and unions containing it are blocked.

(b) **insert: $\eta \to \ell \to$ State $\to \eta$**
$\eta' = \textbf{insert}(\eta, \ell, [s_i]_{i=0}^{l})$ returns a new map η' that is identical to η with the following modifications:
- If $s_i = \textbf{None}$, then $\eta(\ell + i) = \textbf{None}^B$.
- If $s_i = \textbf{Moved}(\ell')$, then $\eta(\ell') = \textbf{Moved}^B(\ell + i)$.
- If $s_i = \textbf{MutRef}(\ell')$, then $\eta(\ell') = \textbf{Ref}^B(\{(\text{true}, \ell + i)\})$.
- If $s_i = \textbf{ImmutRef}(\ell')$, then $\eta(\ell') = \textbf{Ref}^B(\{(\text{false}, \ell + i)\})$.
- If $s_i = \textbf{Union}([s_j]_n)$, then we follow a similar algorithm for each s_j, but instead of overriding the previous s, we join s and s_j into a **Union^B**.

E-Assign

$$\frac{\Gamma, \eta, \mu \vdash e : (\tau, s),\ \Gamma, \eta', \mu \qquad \Gamma, \eta, \mu \vdash_{\ell} \text{lhs} : \text{Ref}\,\langle \text{true}, \tau \rangle,\ \Gamma, \eta, \mu \qquad \textbf{mayoverwrite}(\eta, \ell, \tau) = \text{true} \qquad \eta' = \textbf{insert}(\textbf{drop}(\eta, \ell, \tau), \ell, s)}{\Gamma, \eta, \mu \vdash \text{lhs} = e : (\text{unit}, \textbf{None}),\ \Gamma, \eta''', \mu}$$

Whenever we make a mutable reference, we first check the mutability of lhs. If it's immutable, then an error is raised. Then, we generate the addresess controlled by ℓ and make sure each corresponding borrow state is **None^B** (or a **Union^B** that only contains **None^B**). This rule is simpler to reason with than thinking about how we cannot mutably borrow something that is dropped, moved, or immutably borrowed.

(a) **getaddresses: $\ell \to \tau \to [\ell_i]_i$**
The helper function returns a list of memory addresses that are under ℓ. We use τ to determine whether ℓ is a single-value or a head of a region of memory.

E-MutRef

$$\frac{\Gamma, \eta, \mu \vdash_{\ell} \text{lhs} : (\text{Ref}\,\langle \text{true}, \tau \rangle, \ell) \qquad \forall \ell' \in \textbf{getaddresses}(\ell, \tau) : \eta(\ell') = \textbf{None}^B}{\Gamma, \eta, \mu \vdash \&\textbf{mut}\ \text{lhs} : (\text{Ref}\,\langle \text{true}, \tau \rangle, \textbf{Ref}^B((\text{true}, \ell))),\ \Gamma, \eta, \mu}$$

3 Tool Presentation

We now present our static step-debugger through an example. The following code will demonstrate most of what Oxidize's step-debugger will do with the exception of lifetimes. While the value of **a.2** will try to escape out of the lexical scope that introduced **a**, the step-debugger correctly identifies an error that should be raised before.

```
let mut a = [8, -23, 5] in
    let b = if true then ([a.1])
    else ((let c = &a.1 in unit);
          (let e = a in [0])) in
        a.1
```

3.1 Steps

As mentioned before, tuples have an atypical memory representation in Oxidize. The tuple aggregate lives at address 0, and the integer -23, the second element of the tuple, lives at address 2.

```
Type Context (Gamma):
  a: mutable Int * Int * Int

Memory Map (Mu):
  a -> 0
    a.0 -> 1
    a.1 -> 2
    a.2 -> 3

Borrow Layout (Eta) :

-------------------------
let mut a = [8, -23, 5] in
```

While we can clearly see that only the true-branch of the conditional expression will be evaluated at run-time, Oxidize does not do any clever compiler optimizations with tautologies. Like Rust, Oxidize type checks both branches.For the user's convenience, we allow the user to select one branch to step-through. If they wish to step-through the other branch, they can rerun the debugger.

```
Type Context (Gamma):
  ...
Memory Map (Mu):
  ...
Borrow Layout (Eta) :
-------------------------
let mut a = [8, -23, 5] in
  let mut b = if true

Select which branch you want to go to.
Input `true` or `false`.
```

True Branch Note that the borrow layout does not immediately show that the value at address 2 is dropped. Because **a.1** is a (possible) value of the conditional expression, we must wait until the conditional expression is all type-checked to update the borrow layout with the unification of the branches.

```
Type Context (Gamma):
  a: mutable Int * Int * Int
Memory Map (Mu):
  a -> 0
    ...
Borrow Layout (Eta) :
-------------------------
let mut a = [8, -23, 5] in
  let mut b = if true then ([a.1])
```

False Branch While this immutable borrowing of address 0 has consequences on values at addresses 1, 2, and 3, we do not directly store like so in the borrow state. Because

the tuple elements are accessed as an lvalue, we can always check the tuple which an indexing expressions is indexing.

```
Type Context (Gamma):
  a: mutable Int * Int * Int
  c: immutable &Int

Memory Map (Mu):
  a -> 0
    ...
  c -> 4

Borrow Layout (Eta) :
  the values at addresses...
    0 is partially immutably_referenced
    2 is immutably referenced by addresses [4]
-------------------------
let mut a = [8, -23, 5] in
    let b = if true then ([a.1])
    else ((let c = &a.1 in
```

When tuples are moved, their elements are also moved accordingly.

```
Type Context (Gamma):
  a: mutable Int * Int * Int
  e: immutable Int * Int * Int

Memory Map (Mu):
  a -> 0
    ...
  e -> 4
    e.0 -> 5
    e.1 -> 6
    e.2 -> 7

Borrow Layout (Eta) :
  the values at addresses...
   0 is moved to address 4
      1 is moved to address 5
      2 is moved to address 6
      3 is moved to address 7
-------------------------
let mut a = [8, -23, 5] in
    let b = if true then ([a.1])
    else ((let c = &a.1 in unit);
          (let e = a in
```

Branches unified Unification of a conditional expression's branches are done immediately after both branches are type-checked. Unification checks with the assumption that the borrow states resulting from both branches are possible.

```
Type Context (Gamma):
  a: mutable Int * Int * Int

Memory Map (Mu):
  a -> 0
    ...

Borrow Layout (Eta) freshly unified:
  the values at addresses...
   0 is dropped
      1 is dropped
      2 is dropped
      3 is dropped
-------------------------
let mut a = [8, -23, 5] in
    let b = if true then ([a.1])
    else ((let c = &a.1 in unit);
          (let e = a in [0]))
```

Now that both branches are unified, we can show that the value at address 2 (**a.1**) was moved to address 5 (**b.0**). While this move is a *possible* move, we can conservatively and safely treat this possibility as certain since the other branch does not directly propagate a borrow state.

```
Type Context (Gamma):
  a: mutable Int * Int * Int
  b: immutable Int

Memory Map (Mu):
  a -> 0
    ...
  b -> 4
    b.0 -> 5

Borrow Layout (Eta) :
  the values at addresses...
   0 is dropped
      1 is dropped
      2 is moved to address 5
      3 is dropped
-------------------------------------------------------------

let mut a = [8, -23, 5] in
    let b = if true then ([a.1])
    else ((let c = &a.1 in unit);
          (let e = a in [0])) in
```

We observe that the value at address 3 was dropped (somewhere in the false branch). Thus, using that moved value will raise an error.

```
Type Context (Gamma):
  a: mutable Int * Int * Int
  b: immutable Int

Memory Map (Mu):
  a -> 0
    ...
  b -> 4
    ...

Borrow Layout (Eta) :
  the values at addresses...
   0 is dropped
      1 is dropped
      2 is moved to address 5
      3 is dropped
-------------------------------------------------------------

let mut a = [8, -23, 5] in
    let b = if true then ([a.1])
```

```
    else ((let c = &a.1 in unit);
          (let e = a in [0])) in
        a.1
-----------------------------------------------------


At 135..138, TypeCheckError was raised: You're
    trying to use a value that was already moved
     to address 5.

Code snippet:
```
a.1
```
```

4 Future Work

When building a pedagogical tool, a natural question to ask is about the efficacy of the tool. Therefore, a natural future step of our research would be to conduct a user study. Beforehand, we would also like to improve the UI. One possible improvement would be to write a VSCode extension allowing users to step-through an Oxidize file, similar to how the Lean VSCode extension allows users to step-through the type-checking of a Lean file.

Another extension to this project would be to provide examples of illegal programs to students, and let them use static step-debugging to spot the memory violation. We would replace any condition with true or false to produce a code path that necessarily triggers the violation. For example, the below code would not type-check because the value of **a** could be moved and then used. Thus, we would replace **COND** with **true**.

```
let a = 3 in
    let b = if COND then a else 4 in
        a
```

More ambitious future directions include support for contracts, which are annotations that describe expected behavior by specifying the behavior of functions and methods. While Rust conservatively infers some local invariants, Oxidize prefers explicitness. This causes Oxidize's semantics to be stricter than Rust's. With an annotation system that is an extension of Hoare logic, users could explore lifetimes with Oxidize. With functions added to Oxidize, users could write post-conditions for functions that mutably borrow entire structs while only possibly mutating parts of it. In the Rust code below, a mutable borrow to the immutably borrowed **self** triggers a memory violation even though we can clearly see the borrows are disjoint. While this can be resolved by passing mutable references of the attribute **self**.**val** to the write function, this solution is less elegant. If programmers could use contracts to guarantee that the write function does not mutate the **color** attribute, then the borrow checker would be satisfied as is.

```
struct Node {val : String, color: bool}
impl Node {
    fn whoops(&mut self) {
        let c = &self.color;
        self.write("value".into());
        let _ = format!("{}", c);
    }
    fn write(&mut self, v: String) {
         self.val = v;
    }
}
```

References

[1] Michael Coblenz, Gauri Kambhatla, Paulette Koronkevich, Jenna L. Wise, Celeste Barnaby, Joshua Sunshine, Jonathan Aldrich, and Brad A. Myers. 2020. PLIERS: A Process that Integrates User-Centered Methods into Programming Language Design. arXiv:1912.04719 [cs.HC]

[2] Michael Coblenz, Michelle L. Mazurek, and Michael Hicks. 2022. Garbage Collection Makes Rust Easier to Use: A Randomized Controlled Trial of the Bronze Garbage Collector. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1021–1032. https://doi.org/10.1145/3510003.3510107

[3] Will Crichton. 2021. The Usability of Ownership. arXiv:2011.06171 [cs.PL]

[4] Summers et. al. 2022. *2022 CWE Top 25 Most Dangerous Software Weaknesses.* MITRE. Retrieved June 17, 2023 from https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html

[5] Rust Foundation. 2023. Rustlings. https://github.com/rust-lang/rustlings.

[6] Gongming, Luo, Vishnu Reddy, Marcelo Almeida, Yingying Zhu, Ke Du, and Cyrus Omar. 2020. RustViz: Interactively Visualizing Ownership and Borrowing. arXiv:2011.09012 [cs.PL]