Patina: A Formalization of the Rust Programming Language

Eric Reed
University of Washington

February 2015

Abstract

Rust is a new systems language that uses some advanced type system features, specifically affine types and regions, to statically guarantee memory safety and eliminate the need for a garbage collector. While each individual addition to the type system is well understood in isolation and are known to be sound, the combined system is not known to be sound. Furthermore, Rust uses a novel checking scheme for its regions, known as the Borrow Checker, that is not known to be correct. Since Rust's goal is to be a safer alternative to C/C++, we should ensure that this safety scheme actually works. We present a formal semantics that captures the key features relevant to memory safety, unique pointers and borrowed references, specifies how they guarantee memory safety, and describes the operation of the Borrow Checker. We use this model to prove the soudness of some core operations and justify the conjecture that the model, as a whole, is sound. Additionally, our model provides a syntactic version of the Borrow Checker, which may be more understandable than the non-syntactic version in Rust.

Overview

The Rust Programming Language [1] makes strong claims about ensuring memory safety without garbage collection. We would like to prove that those claims are true. To that end, we use a small model of Rust, called Patina, that characterizes the key features of Rust under the most suspicion, namely its memory management.

Rust memory management has two goals: no memory should ever become unreachable without being deallocated and no uninitialized memory should ever be read. Rust tries to achieve this by maintaining two invariants: all memory has a unique owner responsible for deallocating it and no memory is ever simultaneously aliased and mutable. These invariants simplify the situation enough so that Rust only needs to track which L-values are uninitialized and which have been borrowed. Ownership tracking of heap memory is managed by unique pointers. Safe aliasing is managed by borrowed references.

The Patina model has three layers. The innermost layer deals with L-values and ensures no uninitialized data is ever used, which includes ensuring that uninitialized pointers and null pointers are never dereferenced. It also ensures that compile-time initialization information correctly models the runtime memory. The middle layer deals with R-values. It ensures that any L-values used do not violate initialization or borrow restrictions. It also ensures that borrowed references are not created unless they are safe. The outer layer deals with statements. It mostly just chains together the guarantees of the L-value and R-value layers. However, it is also responsible for ensuring no leaks would be created when deallocating memory.

Unique Pointers in Rust

In Rust, unique pointers are the owners of heap memory. Heap memory is allocated when a new unique pointer is created. Heap memory is freed when a unique pointer falls out of scope.

```
{
  let x: ~int; // A stack variable containing a unique pointer to an integer
  x = ~3; // Allocates a heap integer, initializes it with 3, and stores the pointer in x
  // The heap memory owned by x is freed when it falls out of scope
}
```

To avoid double frees, unique pointers must remain unique. Thus, when a unqiue pointer would be copied the original must be rendered unsuable. That is, they are moved rather than copied.

```
{
  let x: ~int = ~3;
  let y: ~int = x; // x is moved into y. x is no longer usable.
  let z: ~int = x; // ERROR!
}
```

However, these deinitialized paths can be reinitialized.

```
{
  let x: ~int = ~3;
  let y: ~int = x; // x is now deinitialized
  x = ~1; // x is initialized again
}
```

Unique pointers can also be freed if they would be leaked by assignment.

```
{
  let x: ~~int = ~~3;
  *x = ~1; // The ~3 that *x points too would be leaked here. It is freed instead.
}
```

This is not a dynamic check. The compiler detects when heap memory should be freed and inserts the necessary code.

Unique Pointers in Patina

When unique pointers fall out of scope in Rust, the compiler inserts code at the end of the block that will actually free the allocated heap memory. This code will traverse the entire owned data structure and free memory without leaking anything. In the case of recursive data structures, this code will be a recursive function. Rather than model such complex behavior as a primitive in Patina, we will use only a shallow free statement. Aside from making the model simpler, this also provides us with a means of checking that the code Rust inserts to free memory is correct, i.e. by encoding it in terms of Patina's shallow frees. The only way to deallocate a unique pointer in Patina is with these explicit frees.

```
{
  let x: ~~int = ~~3;
  //free(x); // ERROR! would leak *x
  // The proper way
  free(*x);
  free(x);
}
```

However, the free statement is only valid for initialized pointers, which prevents double frees.

```
{
  let x: ~int = ~3;
  free(x); // *x is now deallocated. x is uninitialized
  free(x); // ERROR! x is not initialized
}
```

Finally, the free statement requires that the pointer is unaliased, preventing dangling pointers.

```
{
  let x: ~int = ~3;
  // Create an immutable borrowed reference to *x with &
  // Immutable borrowed references to some type A are denoted by &A
  let y: &int = &*x;
  free(x); // ERROR! would make y a dangling pointer
}
```

Borrowed References

Often, particularly in functions, we want to use some memory, but do not intend to free it. That is, we want to borrow access to memory owned by something else. This is fine for copyable values, like integers, but is tedious for noncopyable values such as unique pointers. To avoid freeing a unique pointer argument at the end of the function body, it must be returned from the function. Effectively, the programmer must thread noncopyable values through their function calls.

```
fn foo(x: ~int) -> (int, ~int) {
  return (*x + 1, x)
}

{
  let x1: ~int = ~1;
  let (y,x2): (int, ~int) = foo(x);
  // y = 2, x1 is no longer valid, x2 = ~1
}
```

Borrowed references allow programmers to have temporary access to a value, but they do not confer ownership so this tedium is not necessary.

```
fn foo(x: &int) -> int {
  return (*x + 1)
}

{
  let x: ~int = ~1;
  let y: int = foo(&*x); // Rust lets us write foo(x) here and will insert the &* for us
  // x = ~1, y = 2
}
```

However, unrestricted borrowed references would open a hole in our memory safety by enabling aliased, mutable memory.

```
{
  let x: ~int = ~1;
  let y: &~int = &x; // x and *y are the same memory location
  let z: ~int = x; // x is no longer usable - effectively x is uninitialized memory
  let w: int = **y; // ERROR! *y is uninitialized now, so **y dereferences a null pointer!
}

{
  let x: &int;
  {
```

```
    let y: int = 3;
    y = &x;
  } // x would be freed here
  let z: int = *x // ERROR *y would point to unallocated memory
}

{
  let x: Option<int> = Some(3);
  let y: &int = match x {
    Some(ref z) => z,
    None => fail!(), // Impossible
  }
  x = None // payload is now uninitialized
  let z: int = *y; // ERROR! *y points to uninitialized memory
}
```

Aliased, mutable memory can be avoided in two ways: forbidding mutability or forbidding aliasing. Correspondingly, Rust has two kinds of borrowed references. Immutable borrows, denoted `&`, are aliasable, but require all memory reachable through them to be immutable for the duration of the borrow.

```
{
  let x: ~int = ~1;
  let y: &~int = &x; // this immutable borrow prevents x from being changed
  let z: ~int = x; // Rust will not let this typecheck because it would change x
}
```

Conversely, mutable borrows, denoted `&mut`, are mutable, but require all memory reachable through them to be *only* accessible through the borrow for the duration. That is, they guarantee unique access.

```
{
  let x: ~int = ~1;
  let y: &mut ~int = &mut x; // this mutable borrow prevents others from using x
  let z: ~int = x; // Rust will not let this typecheck because it would access x
}
```

Finally, both kinds of borrowed reference require that the referent outlive the reference, which prevents dangling references.

```
fn foo() -> &int {
  let x: int = 1;
  return &x; // Rust will not let this typecheck because it would be a dangling pointer
}
```

Types

Values in Patina can be described by the following types:

Lifetime	ℓ		
Type Variable	X		
Qualifier	q	::=	$\mathsf{imm} \mid \mathsf{mut}$
Type	τ	::=	$\mathsf{int} \mid \sim\tau \mid \& \ell\ q\ \tau \mid \langle\tau_i\rangle_{i\in\{1\ldots n\}} \mid [\tau_i]_{i\in\{1\ldots n\}} \mid \mu X.\tau \mid X$

$\sim\tau$ is a unique pointer to a τ. $\& \ell\ q\ \tau$ is a borrowed reference to a τ providing mutability guarantee q for lifetime ℓ, which is Rust's name for regions associated with stack frames. $\langle\tau_i\rangle_{i\in\{1\ldots n\}}$ is a discriminated union or a variant type. $[\tau_i]_{i\in\{1\ldots n\}}$ is a tuple type. Finally, $\mu X.\tau$ is a recursive type.

L-values

L-values in Patina are a combination of a variable and a path. Paths are relative and specify subections of memory reachable from a L-value. Projection ($p \cdot i$) deconstructs tuples. Unrolling ($\mathsf{unroll}\ [\tau]\ p$) deconstructs recursive types. Dereference ($*p$) deconstructs pointers and references. The base path ($\bullet$) does not deconstruct anything.

$$\begin{array}{lll} \text{Variable} & x & \\ \text{Path} & p & ::= \quad \bullet \mid *p \mid p \cdot i \mid \mathsf{unroll}\ [\tau]\ p \end{array}$$

The L-value layer of Patina has two core operations: evaluating a path to a memory location and reading the value at that location. We need to characterize the circumstances under which these operations will progress, and we need to show that these operations preserve certain information, specifically type and initialization data. Higher layers of Patina will build on these properties. We shall start with typing paths and then move into defining evaluation for paths.

The typing judgment for paths does not present any surprises. We use a partial map for the typing context and the type substitution operation is the standard capture-avoiding substition.

$$\Gamma : \text{Variable} \to \text{Type}$$

$\boxed{\Gamma \vdash x@p : \tau}$

$$\text{PT-BASE}\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x@\bullet : \tau} \qquad \text{PT-DEOWN}\ \frac{\Gamma \vdash x@p : \sim \tau}{\Gamma \vdash x@*p : \tau} \qquad \text{PT-DEREF}\ \frac{\Gamma \vdash x@p : \&\ \ell\ q\ \tau}{\Gamma \vdash x@*p : \tau} \qquad \text{PT-PROJ}\ \frac{\Gamma \vdash x@p : [\tau_i]_{i \in \{1 \ldots n\}}}{\Gamma \vdash x@p \cdot i : \tau_i}$$

$$\text{PT-UNROLL}\ \frac{\Gamma \vdash x@p : \mu X.\tau}{\Gamma \vdash x@\mathsf{unroll}\ [\mu X.\tau]\ p : [X \mapsto \mu X.\tau]\tau}$$

Runtime Memory

Representation and Addressing

To accurately model Rust's memory usage, Patina restricts the contents of a memory cell to void data, an integer, or a pointer to another cell. Tuples and recursive types have no physical memory presence (beyond contiguity for tuples). The memory representation of a variant is a pair of a cell for the discriminant and whatever memory is necessary to store the payload.

We could model this by a map from addresses to memory cell values, but two issues make this inconvenient: the need for address arithmetic and non-unique typing. However, we can add a little extra structure to our model and eliminate these two issues. We wrap the cells inside layouts describing the type structure overlaying memory.

We also separate plain pointer cells into owned cells and reference cells. This is mostly useful for providing just enough type information about the pointer for memory operation purposes. We will explain how we address memory in a moment, so we use a placeholder for now.

$$\begin{array}{lllll} \text{Integer} & z & & & \\ \text{Address} & \rho & & & \text{a placeholder} \\ \text{Cell} & c & ::= & \mathsf{void} \mid z \mid \sim \rho \mid \&\ q\ \rho & \\ \text{Layout} & l & ::= & c \mid \langle c : l \rangle \mid [l_i]_{i \in \{1 \ldots n\}} \mid \mathsf{roll}\ [\tau]\ l & \end{array}$$

Due to the extra structure from layouts, addressing memory now requires more than simple labels. Instead we use a runtime analogue of variables and paths. Allocations are chunks of memory allocated and

freed atomically. They correspond to either variables or heap allocations. Routes are like paths except for two key differences.

$$\begin{array}{lll} \text{Allocation} & \alpha & \\ \text{Route} & r & ::= \quad \bullet \mid r \cdot i \mid \mathsf{pay}\ r \mid \mathsf{unroll}\ [\tau]\ r \end{array}$$

The $\bullet$ route, projection, and unrolling are effectively identical to their path equivalents. The $\mathsf{pay}\ r$ route refers to the payload l in a variant $\langle c : l \rangle$. This is primarily used for match-by-reference. Unlike paths, there is no dereference route. This forces any pointer following into path evaluation (into routes) rather than doing so while reading memory at a route. The address placeholder from before is simply a pair of an allocation label and a route. The exception is that unique pointers need only an allocation since a unique pointer always points to a complete heap allocation. That is:

$$\begin{array}{lll} \text{Integer} & z & \\ \text{Allocation} & \alpha & \\ \text{Route} & r & ::= \quad \bullet \mid r \cdot i \mid \mathsf{pay}\ r \mid \mathsf{unroll}\ [\tau]\ r \\ \text{Cell} & c & ::= \quad \mathsf{void} \mid z \mid \sim \alpha \mid \&\ q\ \alpha@r \\ \text{Layout} & l & ::= \quad c \mid \langle c : l \rangle \mid [l_i]_{i \in \{1 \ldots n\}} \mid \mathsf{roll}\ [\tau]\ l \end{array}$$

Reading

Reading a layout from a route in memory is a straightforward operation. We model the heap as a partial map from allocations to layouts. Routes and layouts interact as you would expect.

$$H : \text{Allocation} \to \text{Layout}$$

$\boxed{\textsc{read}(H, \alpha@r, l)}$

$$\text{RD-BASE}\ \frac{H(\alpha) = l}{\textsc{read}(H, \alpha@\bullet, l)} \qquad \text{RD-PROJ}\ \frac{\textsc{read}(H, \alpha@r, [l_i]_{i \in \{1 \ldots n\}})}{\textsc{read}(H, \alpha@r \cdot i, l_i)} \qquad \text{RD-PAY}\ \frac{\textsc{read}(H, \alpha@r, \langle c : l \rangle)}{\textsc{read}(H, \alpha@\mathsf{pay}\ r, l)} \qquad \text{RD-UNROLL}\ \frac{\textsc{read}(H, \alpha@r, \mathsf{roll}\ [\tau']\ l)}{\textsc{read}(H, \alpha@\mathsf{unroll}\ [\tau]\ r, l)}$$

As one would expect of a read operation, it's result is unique.

Lemma (Read Uniqueness)**.** *If* $\textsc{read}(H, \alpha@r, l)$ *and* $\textsc{read}(H, \alpha@r, l')$*, then* $l = l'$*.*

Read Uniqueness. We will induct on the derivation of $\textsc{read}(H, \alpha@r, l)$.

RD-BASE Then $r = \bullet$.
By inversion, $H(\alpha) = l$.
There is only one rule (RD-BASE) for deriving $\textsc{read}(H, \alpha@\bullet, l')$.
Thus, by inversion, $H(\alpha) = l'$.
Ergo, $l = l'$.

RD-PROJ Then $r = r' \cdot i$ and $l = l_i$.
By inversion, $\textsc{read}(H, \alpha@r', [l_i]_{i \in \{1 \ldots n\}})$.
There is only one rule (RD-PROJ) for deriving $\textsc{read}(H, \alpha@r' \cdot i, l')$.
Thus, by inversion, $\textsc{read}(H, \alpha@r', [l'_i]_{i \in \{1 \ldots n'\}})$.
By induction, $[l_i]_{i \in \{1 \ldots n\}} = [l'_i]_{i \in \{1 \ldots n'\}}$.
Ergo, $n = n'$ and $l_i = l'_i$.

RD-PAY Then $r = \mathsf{pay}\ r'$.
By inversion, $\textsc{read}(H, \alpha@r', \langle c : l \rangle)$.
There is only one rule (RD-PAY) for deriving $\textsc{read}(H, \alpha@\mathsf{pay}\ r', l')$.
Thus, by inversion, $\textsc{read}(H, \alpha@r', \langle c' : l' \rangle)$.

By induction, $\langle c : l \rangle = \langle c' : l' \rangle$.
Ergo, $c = c'$ and $l = l'$.

RD-UNROLL Then $r = \mathsf{unroll}\ [\tau]\ r'$.
By inversion, $\textsc{read}(H, \alpha @ r', \mathsf{roll}\ [\tau']\ l)$.
There is only one rule (RD-UNROLL) for deriving $\textsc{read}(H, \alpha @ \mathsf{unroll}\ [\tau]\ r', l')$.
Thus, by inversion, $\textsc{read}(H, \alpha @ r', \mathsf{roll}\ [\tau'']\ l')$.
By induction, $\mathsf{roll}\ [\tau']\ l = \mathsf{roll}\ [\tau'']\ l'$.
Ergo, $\tau' = \tau''$ and $l = l'$.

□

Path Evaluation

Now that we can read memory, we can define path evaluation. The only actual work of path evaluation is following the dereferences of pointers to produce a route. In order to connect variables to runtime allocations, we use a partial map tracking the allocation labels of variables.

$$V : \text{Variable} \to \text{Allocation}$$

$\boxed{V; H \vdash x @ p \to \alpha @ r}$

$$\text{PE-BASE}\ \frac{V(x) = \alpha}{V; H \vdash x @ \bullet \to \alpha @ \bullet} \qquad \text{PE-DEOWN}\ \frac{V; H \vdash x @ p \to \alpha @ r \qquad \textsc{read}(H, \alpha @ r, \sim \alpha')}{V; H \vdash x @ *p \to \alpha' @ \bullet}$$

$$\text{PE-DEREF}\ \frac{V; H \vdash x @ p \to \alpha @ r \qquad \textsc{read}(H, \alpha @ r, \&\ q\ \alpha' @ r')}{V; H \vdash x @ *p \to \alpha' @ r'} \qquad \text{PE-PROJ}\ \frac{V; H \vdash x @ p \to \alpha @ r}{V; H \vdash x @ p \cdot i \to \alpha @ r \cdot i}$$

$$\text{PE-UNROLL}\ \frac{V; H \vdash x @ p \to \alpha @ r}{V; H \vdash x @ \mathsf{unroll}\ [\tau]\ p \to \alpha @ \mathsf{unroll}\ [\tau]\ r}$$

Our core operations are now defined, but we still need to describe the typing of our runtime data so that we can properly state and prove our type preservation lemmas.

Runtime Typing

Route Typing

Typing routes is much like typing paths; however, the $\mathsf{pay}\ r$ route requires special attention. Since this route points to the payload of a variant, its type depends upon the value of the discriminant of the variant. This is used for handling match by reference. The match arms have a pointer to the payload in scope, but the type of the payload depends upon the value of the discrimiant. For example,

```
// x: <[], int> is a Patina encoding of Option<int>
match x by imm y // y will have route ‘pay r’ were ‘r’ is the route of x
  // [] (None) case
  {
    // y: & imm [] is in scope here
  }
```

```
// int (Some) case
{
  // y: & imm int is in scope here
}
```

We handle this with an additional context that records the discriminant value of a variant stored at a route.

$$\Sigma : \text{Allocation} \to \text{Type}$$

$$\Psi : \text{Allocation} \times \text{Route} \to \text{Integer}$$

$\boxed{\Sigma; \Psi \vdash \alpha @ r : \tau}$

$$\text{RT-BASE}\ \frac{\Sigma(\alpha) = \tau}{\Sigma; \Psi \vdash \alpha @ \bullet : \tau} \qquad \text{RT-PROJ}\ \frac{\Sigma; \Psi \vdash \alpha @ r : [\tau_i]_{i \in \{1 \ldots n\}}}{\Sigma; \Psi \vdash \alpha @ r \cdot i : \tau_i} \qquad \text{RT-PAY}\ \frac{\Psi(\alpha, r) = i \qquad \Sigma; \Psi \vdash \alpha @ r : \langle \tau_i \rangle_{i \in \{1 \ldots n\}}}{\Sigma; \Psi \vdash \alpha @ \mathsf{pay}\ r : \tau_i}$$

$$\text{RT-UNROLL}\ \frac{\Sigma; \Psi \vdash \alpha @ r : \mu X.\tau}{\Sigma; \Psi \vdash \alpha @ \mathsf{unroll}\ [\mu X.\tau]\ r : [X \mapsto \mu X.\tau]\tau}$$

Layout and Cell Typing

Typing for layouts and cells is what you would expect. However, we restrict the type of void to the possible types for other kinds of cells, i.e. int, unique pointers, and borrowed references. This means that multi-cell layout types (variants, tuples, and recursive types) imply a non-void value (though they can still contain voids).

$\boxed{\Sigma; \Psi \vdash l : \tau}$

$$\text{LT-VOIDINT}\ \frac{}{\Sigma; \Psi \vdash \mathsf{void} : \mathsf{int}} \qquad \text{LT-VOIDOWN}\ \frac{}{\Sigma; \Psi \vdash \mathsf{void} : \sim \tau} \qquad \text{LT-VOIDREF}\ \frac{}{\Sigma; \Psi \vdash \mathsf{void} : \& \ \ell \ q \ \tau} \qquad \text{LT-INT}\ \frac{}{\Sigma; \Psi \vdash z : \mathsf{int}} \qquad \text{LT-OWN}\ \frac{\Sigma; \Psi \vdash \alpha @ \bullet : \tau}{\Sigma; \Psi \vdash \sim \alpha : \sim \tau}$$

$$\text{LT-REF}\ \frac{\Sigma; \Psi \vdash \alpha @ r : \tau}{\Sigma; \Psi \vdash \& \ q \ \alpha @ r : \& \ \ell \ q \ \tau} \qquad \text{LT-VOIDVAR}\ \frac{}{\Sigma; \Psi \vdash \langle \mathsf{void} : \mathsf{void} \rangle : \langle \tau_i \rangle_{i \in \{1 \ldots n\}}} \qquad \text{LT-VAR}\ \frac{\Sigma; \Psi \vdash l : \tau_i}{\Sigma; \Psi \vdash \langle i : l \rangle : \langle \tau_i \rangle_{i \in \{1 \ldots n\}}}$$

$$\text{LT-REC}\ \frac{\forall i.\ \Sigma; \Psi \vdash l_i : \tau_i}{\Sigma; \Psi \vdash [l_i]_{i \in \{1 \ldots n\}} : [\tau_i]_{i \in \{1 \ldots n\}}} \qquad \text{LT-ROLL}\ \frac{\Sigma; \Psi \vdash l : [X \mapsto \mu X.\tau]\tau}{\Sigma; \Psi \vdash \mathsf{roll}\ [\mu X.\tau]\ l : \mu X.\tau}$$

Heap Well-Formedness

With runtime typing now in hand, we can say what it means for the heap to be well-formed. We say a heap H is described by a heap type Σ and a discriminant context Ψ if every allocation in H has the type assigned to it by Σ and Ψ correctly records the discrimiants of all the variants in H. The last hypothesis enforces the uniqueness of unique pointers by requiring that if a unique pointer is at any address, then that address is unique. This strengthens our induction hypothesis enough to show that no memory has multiple owners, which we need to prove that no double frees occur. Ensuring every piece of memory has at least one owner is handled elsewhere. Formally,

$$\frac{\begin{array}{l}\operatorname{dom} H = \operatorname{dom} \Sigma \\ \forall \alpha \in \operatorname{dom}(\Sigma).\ \Sigma(\alpha) \text{ closed} \\ \forall \alpha \in \operatorname{dom} H.\ \Sigma; \Psi \vdash H(\alpha) : \Sigma(\alpha) \\ \forall (\alpha, r) \in \operatorname{dom} \Psi.\ \textsc{read}(H, \alpha@r, \langle \Psi(\alpha, r) : l \rangle) \\ \forall \alpha, \alpha', \alpha'' \in \operatorname{dom} H.\ \forall r, r.\ \textsc{read}(H, \alpha@r, \sim \alpha'') \wedge \textsc{read}(H, \alpha'@r', \sim \alpha'') \implies \alpha@r = \alpha'@r'\end{array}}{\vdash H : \Sigma; \Psi}$$

Read Safety

We are now in position to prove that our read operation is safe. That is, if the heap is well formed and a route has some type, then we can read some layout at that route and that layout has that same type. This is effectively a merger of the type preservation and progress lemmas for READ.

Lemma (Read Safety)**.** *If* $\vdash H : \Sigma; \Psi$ *and* $\Sigma; \Psi \vdash \alpha@r : \tau$*, then there is a layout* l *such that* $\textsc{read}(H, \alpha@r, l)$ *and* $\Sigma; \Psi \vdash l : \tau$*.*

Read Safety. We will use induction on the derivation of $\Sigma; \Psi \vdash \alpha@r : \tau$.

RT-BASE Then $r = \bullet$.
By inversion, $\Sigma(\alpha) = \tau$.
From $\vdash H : \Sigma; \Psi$, we know that $\operatorname{dom} H = \operatorname{dom} \Sigma$. Thus, $\exists l.\ H(\alpha) = l$.
Then by RD-BASE, we have $\textsc{read}(H, \alpha@\bullet, l)$.
Again from $\vdash H : \Sigma; \Psi$, we know that $\Sigma; \Psi \vdash H(\alpha) : \Sigma(\alpha)$, which is just $\Sigma; \Psi \vdash l : \tau$ as required.

RT-PROJ Then $r = r' \cdot i$ and $\tau = \tau_i$.
By inversion, $\Sigma; \Psi \vdash \alpha@r' : [\tau_i]_{i \in \{1 \ldots n\}}$.
By induction, $\textsc{read}(H, \alpha@r', l)$ and $\Sigma; \Psi \vdash l : [\tau_i]_{i \in \{1 \ldots n\}}$ for some l.
There is only one rule (LT-REC) for deriving $\Sigma; \Psi \vdash l : [\tau_i]_{i \in \{1 \ldots n\}}$.
Thus, by inversion, $l = [l'_j]_{j \in \{1 \ldots n\}}$ and $\forall j.\ \Sigma; \Psi \vdash l'_j : \tau_j$. Specifically, $\Sigma; \Psi \vdash l'_i : \tau_i$.
Then by RD-PROJ, we have $\textsc{read}(H, \alpha@r' \cdot i, l'_i)$.

RT-PAY Then $r = \mathsf{pay}\ r'$ and $\tau = \tau_i$.
By inversion, $\Psi(\alpha, r') = i$ and $\Sigma; \Psi \vdash \alpha@r' : \langle \tau_i \rangle_{i \in \{1 \ldots n\}}$.
By induction, $\textsc{read}(H, \alpha@r', l)$ and $\Sigma; \Psi \vdash l : \langle \tau_i \rangle_{i \in \{1 \ldots n\}}$.
From $\vdash H : \Sigma; \Psi$, we know that because $\Psi(\alpha, r') = i$ it follows that $\textsc{read}(H, \alpha@r', \langle i : l' \rangle)$ for some l'.
By Read Uniqueness, we know that $l = \langle i : l' \rangle$.
There is only one rule (LT-VAR) for deriving $\Sigma; \Psi \vdash \langle i : l' \rangle : \langle \tau_i \rangle_{i \in \{1 \ldots n\}}$.
Thus, by inversion, $\Sigma; \Psi \vdash l' : \tau_i$.
Then by RD-PAY, we have $\textsc{read}(H, \alpha@\mathsf{pay}\ r', l')$.

RT-UNROLL Then $r = \mathsf{unroll}\ [\mu X.\tau']\ r'$ and $\tau = [X \mapsto \mu X.\tau']\tau'$.
By inversion, $\Sigma; \Psi \vdash \alpha@r' : \mu X.\tau'$.
By induction, $\textsc{read}(H, \alpha@r', l)$ and $\Sigma; \Psi \vdash l : \mu X.\tau'$ for some l.
There is only one rule (LT-ROLL) for deriving $\Sigma; \Psi \vdash l : \mu X.\tau'$.
Thus, by inversion, $l = \mathsf{roll}\ [\mu X.\tau']\ l'$ and $\Sigma; \Psi \vdash l' : [X \mapsto \mu X.\tau']\tau'$.
Then by RD-UNROLL, we have $\textsc{read}(H, \alpha@\mathsf{unroll}\ [\mu X.\tau']\ r', l')$.

□

Path Type Preservation

The final piece we need for preservation is how to relate the typing context Γ, the variable map V, and the heap type Σ. This is simply the requirement that the map V preserves typing.

$$\frac{\forall x \in \operatorname{dom}\Gamma.\ \Gamma(x) = \Sigma(V(x))}{\Gamma; V \vdash \Sigma}$$

Finally, we can state and prove path type preservation. Under a well-formed heap and contexts, typed paths evaluate to routes of the same type.

Lemma (Path Type Preservation)**.** *If* $\vdash H : \Sigma; \Psi$*,* $\Gamma; V \vdash \Sigma$*,* $\Gamma \vdash x@p : \tau$*, and* $V; H \vdash x@p \to \alpha@r$*, then* $\Sigma; \Psi \vdash \alpha@r : \tau$*.*

Path Type Preservation. We will use induction on the derivation of $V; H \vdash x@p \to \alpha@r$.

PE-BASE Then $p = \bullet$ and $r = \bullet$.
By inversion, $V(x) = \alpha$.
There is only one rule (PT-BASE) for deriving $\Gamma \vdash x@\bullet : \tau$.
Thus, by inversion, we have $\Gamma(x) = \tau$.
From $\Gamma; V \vdash \Sigma$, we know $\Gamma(x) = \Sigma(V(x))$. Thus, $\tau = \Sigma(\alpha)$.
Then by RT-BASE, we have $\Sigma; \Psi \vdash \alpha@\bullet : \tau$.

PE-DEOWN Then $p = *p'$ and $r = \bullet$.
By inversion, $V; H \vdash x@p' \to \alpha'@r'$ and $\textsc{read}(H, \alpha'@r', \sim \alpha)$.
There are two possible rules for deriving $\Gamma \vdash x@*p' : \tau$.

PT-DEOWN By inversion, $\Gamma \vdash x@p' : \sim \tau$.
By induction, $\Sigma; \Psi \vdash \alpha'@r' : \sim \tau$.
By Read Safety, we have $\textsc{read}(H, \alpha'@r', l)$ and $\Sigma; \Psi \vdash l : \sim \tau$ for some l.
By Read Uniqueness, we have $l = \sim \alpha$.
There is only one rule (LT-OWN) for deriving $\Sigma; \Psi \vdash \sim \alpha : \sim \tau$.
Thus, by inversion, $\Sigma; \Psi \vdash \alpha@\bullet : \tau$.

PT-DEREF By inversion, $\Gamma \vdash x@p' : \& \ \ell \ q \ \tau$.
By induction, $\Sigma; \Psi \vdash \alpha'@r' : \& \ \ell \ q \ \tau$.
By Read Safety, we have $\textsc{read}(H, \alpha'@r', l)$ and $\Sigma; \Psi \vdash l : \& \ \ell \ q \ \tau$ for some l.
By Read Uniqueness, we have $l = \sim \alpha$.
There are no rules for deriving $\Sigma; \Psi \vdash \sim \alpha : \& \ \ell \ q \ \tau$.
Thus, by inversion, this case is impossible.

PE-DEREF Similar to the PE-DEOWN case, but switching the roles of PT-DEOWN and PT-DEREF.

PE-PROJ Then $p = p' \cdot i$ and $r = r' \cdot i$.
By inversion, $V; H \vdash x@p' \to \alpha@r'$.
There is only one rule (PT-PROJ) for deriving $\Gamma \vdash x@p' \cdot i : \tau$.
Thus, by inversion, we have $\tau = \tau_i$ and $\Gamma \vdash x@p' : [\tau_i]_{i \in \{1 \ldots n\}}$.
By induction, we have $\Sigma; \Psi \vdash \alpha@r' : [\tau_i]_{i \in \{1 \ldots n\}}$.
Then by RT-PROJ, we have $\Sigma; \Psi \vdash \alpha@r' \cdot i : \tau_i$.

PE-UNROLL Then $p = \mathsf{unroll}\ [\tau']\ p'$ and $r = \mathsf{unroll}\ [\tau']\ r'$.
By inversion, $V; H \vdash x@p' \to \alpha@r'$.
There is only one rule (PT-UNROLL) for deriving $\Gamma \vdash x@\mathsf{unroll}\ [\tau']\ p' : \tau$.
Thus, by inversion, we have $\tau' = \mu X.\tau''$, $\tau = [X \mapsto \mu X.\tau'']\tau''$, and $\Gamma \vdash x@p' : \mu X.\tau''$.
By induction, we have $\Sigma; \Psi \vdash \alpha@r' : \mu X.\tau''$.
Then by RT-UNROLL, we have $\Sigma; \Psi \vdash \alpha@\mathsf{unroll}\ [\mu X.\tau'']\ r' : [X \mapsto \mu X.\tau'']\tau''$.

□

Tracking Initialization

Types are only half of the information we need to preserve. To properly type check Patina, we need to know which paths point to initialized memory and which do not. This is important for ensuring we never use uninitialized memory and that we correctly free heap memory. We will accomplish this by creating a shadow heap. Instead of tracking actual values, we will simply track whether a cell is initialized. Paths will serve the same function for this shadow heap as routes do for the real heap.

$$\begin{array}{llll} \text{Hole} & h & ::= & \mathsf{uninit} \mid \mathsf{int} \mid \sim \sigma \mid \& \; q \; \tau \mid \langle \tau_i \rangle_{i \in \{1 \ldots n\}} \\ \text{Shadow} & \sigma & ::= & h \mid [\sigma_i]_{i \in \{1 \ldots n\}} \mid \mathsf{roll} \; [\tau] \; \sigma \end{array}$$

Since shadows and holes are supposed to be a shadow of the heap, it is unsurprising that they are very similar to layouts and cells. However, there are a few key differences. First, variants are considered an atomic value (and thus a hole) rather than a structure. This is because the discrimiant and payload of a variant always share the same initialization state. Second, while the pointer cells contain routes, the pointer holes do not contain paths (the analogue of routes). For references, there is no need because references always point to fully initialized things. Therefore, the reference hole simply records the type of the referent. For unique pointers, there is no path that could possibly be used (otherwise the pointer would not be unique). Therefore, the unique pointer hole records the shadow of the pointed-to heap memory.

Two examples are useful for understanding shadows. First, consider nested pointers:

```
{
  let x: ~~int = ~~3;
  // Here the shadow of x is `~ ~ int'
  // x, *x, and **x are all initialized

  let y: ~int = *x; // *x is now deinitialized
  // Here the shadow of x is `~ uninit' and the shadow of y is `~ int'
  // x is initialized, but *x and **x are not
  // both y and *y are initialized
}
```

Second, consider tuples:

```
{
  let x: [~int, ~int] = [~1, ~2];
  // Here the shadow of x is [~int, ~int]
  // x, x.1, *(x.1), x.2, and *(x.2) are all initialized

  free(x.1);
  // Here the shadow of x is [uninit, ~int]
  // x, x.1, x.2, and *(x.2) are all initialized, but *(x.1) is not
}
```

The shadow heap gives the typechecker enough information to ensure we do not use or dereference uninitialized memory. For example, here is how it can catch dereferencing a null pointer:

```
{
  let x: ~~int = ~~3;
  free(*x); // deallocates **x, leaving *x uninitialized
  // at this point the shadow of x is `~ uninit'
  let y: int = **x; // ERROR! dereferences a null pointer *x
  // the type checker know the shadow of *x is `uninit'
  // if a further dereference was allowed (i.e. **x), a null pointer derefernce would occur
```

```
  // the type checker can avoid null pointer dereferences by forbidding derefs of 'uninit' data

}
```

It can also catch a missing free that would leak memory.

```
{
  let x: ~int = ~3;
  // the shadow of x is '~ int'

  // the stack variable x will be popped off the stack, leaking the heap memory at *x
  // the type checker can identify this error by seeing that the shadow of x contains '~' data
  // by requiring that shadows cannot contain '~' when freed, the type checker prevents leaks
}
```

Checking Shallow Initialization

We will need an analogue of READ for our shadow heap for the simple purpose of extracting the shadow at a path in the same way we need to extract the layout at a route from the heap. However, this analogue also serves as a very useful property checker: any dereferences in the path are dereferences of initialized memory. This property, called *shallow initialization* is exactly what we need for path progress. A shallowly initialized path can successfully dereference all the pointers necessary to evaluate to a route. For this reason, we call this operation SHALLOW. To support dereferencing borrowed references, we need to construct a shape from the referenced type. We do this with the INIT function.

$\boxed{\textsc{init} : \text{Type} \to \text{Shadow}}$

$$
\begin{array}{lcl}
\textsc{init}(\mathsf{int}) & = & \mathsf{int} \\
\textsc{init}(\sim \tau) & = & \sim \textsc{init}(\tau) \\
\textsc{init}(\& \ \ell \ q \ \tau) & = & \& \ q \ \tau \\
\textsc{init}([\tau_i]_{i \in \{1 \ldots n\}}) & = & [\textsc{init}(\tau_i)]_{i \in \{1 \ldots n\}} \\
\textsc{init}(\langle \tau_i \rangle_{i \in \{1 \ldots n\}}) & = & \langle \tau_i \rangle_{i \in \{1 \ldots n\}} \\
\textsc{init}(\mu X.\tau) & = & \mathsf{roll}\ [\mu X.\tau]\ \textsc{init}([X \mapsto \mu X.\tau]\tau)
\end{array}
$$

We also need a new context recording the shadows of stack variables.

$$\Upsilon : \text{Variable} \to \text{Shadow}$$

$\boxed{\textsc{shallow}(\Upsilon, x@p, \sigma)}$

$$\text{SI-BASE} \quad \frac{\Upsilon(x) = \sigma}{\textsc{shallow}(\Upsilon, x@\bullet, \sigma)} \qquad \text{SI-DEOWN} \quad \frac{\textsc{shallow}(\Upsilon, x@p, \sim \sigma)}{\textsc{shallow}(\Upsilon, x@*p, \sigma)} \qquad \text{SI-DEREF} \quad \frac{\textsc{shallow}(\Upsilon, x@p, \& \ q \ \tau) \qquad \textsc{init}(\tau) = \sigma}{\textsc{shallow}(\Upsilon, x@*p, \sigma)}$$

$$\text{SI-PROJ} \quad \frac{\textsc{shallow}(\Upsilon, x@p, [\sigma_i]_{i \in \{1 \ldots n\}})}{\textsc{shallow}(\Upsilon, x@p \cdot i, \sigma_i)} \qquad \text{SI-UNROLL} \quad \frac{\textsc{shallow}(\Upsilon, x@p, \mathsf{roll}\ [\tau']\ \sigma)}{\textsc{shallow}(\Upsilon, x@\mathsf{unroll}\ [\tau]\ p, \sigma)}$$

Shadow Typing

We need to ensure that the shadows assigned to variables are consistent with the types assigned to variables, i.e. we want to avoid situations like $\Gamma(x) = \mathsf{int}$ and $\Upsilon(x) = \& \ \mathsf{imm}\ \mathsf{int}$. We can easily do this by defining

a typing judgement for shadows and defining a well-formedness condition for Υ that ensures it is consistent with Γ. Similar to typing for layouts, uninit can only have types that other holes can have.

$$\boxed{\vdash \sigma : \tau}$$

$$\text{ST-UNINT}\ \frac{}{\vdash \mathsf{uninit} : \mathsf{int}} \qquad \text{ST-UNOWN}\ \frac{}{\vdash \mathsf{uninit} : \sim \tau} \qquad \text{ST-UNREF}\ \frac{}{\vdash \mathsf{uninit} : \& \ \ell \ q \ \tau} \qquad \text{ST-UNVAR}\ \frac{}{\vdash \mathsf{uninit} : \langle \tau_i \rangle_{i \in \{1 \ldots n\}}} \qquad \text{ST-INT}\ \frac{}{\vdash \mathsf{int} : \mathsf{int}}$$

$$\text{ST-OWN}\ \frac{\vdash \sigma : \tau}{\vdash \sim \sigma : \sim \tau} \qquad \text{ST-REF}\ \frac{}{\vdash \& \ q \ \tau : \& \ \ell \ q \ \tau} \qquad \text{ST-VAR}\ \frac{}{\vdash \langle \tau_i \rangle_{i \in \{1 \ldots n\}} : \langle \tau_i \rangle_{i \in \{1 \ldots n\}}} \qquad \text{ST-REC}\ \frac{\forall i.\ \vdash \sigma_i : \tau_i}{\vdash [\sigma_i]_{i \in \{1 \ldots n\}} : [\tau_i]_{i \in \{1 \ldots n\}}}$$

$$\text{ST-ROLL}\ \frac{\vdash \sigma : [X \mapsto \mu X.\tau]\tau}{\vdash \mathsf{roll}\ [\mu X.\tau]\ \sigma : \mu X.\tau}$$

$$\frac{\begin{array}{l}\operatorname{dom}\Gamma = \operatorname{dom}\Upsilon \\ \forall x \in \operatorname{dom}\Gamma.\ \Gamma(x) \text{ closed} \\ \forall x \in \operatorname{dom}\Gamma.\ \vdash \Upsilon(x) : \Gamma(x)\end{array}}{\vdash \Upsilon : \Gamma}$$

The INIT Function Preserves Types

Since the INIT function is supposed to be the fully initialized shadow of the argument type, it should be the case that the resulting shadow has that type. Since INIT is only defined on closed types, we must restrict ourselves to them.

Lemma (INIT Type Preservation)**.** *If τ is closed, then $\vdash \textsc{init}(\tau) : \tau$.*

INIT *Type Preservation.* Induction on the form of τ.

INT If $\tau = \mathsf{int}$, then $\textsc{init}(\mathsf{int}) = \mathsf{int}$. From ST-INT, we have $\vdash \mathsf{int} : \mathsf{int}$.

OWN If $\tau = \sim \tau'$, then $\textsc{init}(\sim \tau') = \sim \textsc{init}(\tau')$.
Since τ is closed, τ' is also closed.
Thus, by induction, $\vdash \textsc{init}(\tau') : \tau'$.
From ST-OWN, we have $\vdash \sim \textsc{init}(\tau') : \sim \tau'$.

REF If $\tau = \& \ \ell \ q \ \tau'$, then $\textsc{init}(\& \ \ell \ q \ \tau') = \& \ q \ \tau'$.
$\vdash \& \ q \ \tau' : \& \ \ell \ q \ \tau'$ follows immediately by ST-REF.

REC If $\tau = [\tau'_i]_{i \in \{1 \ldots n\}}$, then $\textsc{init}([\tau'_i]_{i \in \{1 \ldots n\}}) = [\textsc{init}(\tau'_i)]_{i \in \{1 \ldots n\}}$.
Since τ is closed, τ'_i is closed for all i.
Thus, by induction, for all i we have $\vdash \textsc{init}(\tau'_i) : \tau'_i$.
Then by ST-REC, we have $\vdash [\textsc{init}(\tau'_i)]_{i \in \{1 \ldots n\}} : [\tau'_i]_{i \in \{1 \ldots n\}}$.

VAR If $\tau = \langle \tau'_i \rangle_{i \in \{1 \ldots n\}}$, then $\textsc{init}(\langle \tau'_i \rangle_{i \in \{1 \ldots n\}}) = \langle \tau'_i \rangle_{i \in \{1 \ldots n\}}$.
$\vdash \langle \tau'_i \rangle_{i \in \{1 \ldots n\}} : \langle \tau'_i \rangle_{i \in \{1 \ldots n\}}$ follows immediately from ST-VAR.

FIX If $\tau = \mu X.\tau'$, then $\textsc{init}(\mu X.\tau') = \mathsf{roll}\ [\mu X.\tau']\ \textsc{init}([X \mapsto \mu X.\tau']\tau')$.
Since τ is closed, τ' might not be closed; however, the only possible free variable is X.
Thus, the unrolling $[X \mapsto \mu X.\tau']\tau'$ is closed.
Hence, by induction, $\vdash \textsc{init}([X \mapsto \mu X.\tau']\tau') : [X \mapsto \mu X.\tau']\tau'$.
Then by ST-ROLL, we have $\vdash \mathsf{roll}\ [\mu X.\tau']\ \textsc{init}([X \mapsto \mu X.\tau']\tau') : \mu X.\tau'$.

□

Shallow Preserves Types

Just as with READ, SHALLOW preserves types. This lemma is used primarily to demonstrate the impossibility of certain subcases of dereferences in later proofs.

Lemma (SHALLOW Type Preservation)**.** *If* $\vdash \Upsilon : \Gamma$, $\Gamma \vdash x@p : \tau$, *and* $\textsc{shallow}(\Upsilon, x@p, \sigma)$ *then* $\vdash \sigma : \tau$.

SHALLOW *Type Preservation.* Induction on the derivation of $\textsc{shallow}(\Upsilon, x@p, \sigma)$.

SI-BASE Then $p = \bullet$.
By inversion, $\Upsilon(x) = \sigma$.
There is only one rule (PT-BASE) for deriving $\Gamma \vdash x@\bullet : \tau$.
Thus, by inversion, we have $\Gamma(x) = \tau$.
From $\vdash \Upsilon : \Gamma$, we have $\vdash \Upsilon(x) : \Gamma(x)$.
Ergo, we have $\vdash \sigma : \tau$.

SI-DEOWN Then $p = *p'$.
By inversion, $\textsc{shallow}(\Upsilon, x@p', \sim \sigma)$.
There are two possible rules for deriving $\Gamma \vdash x@*p' : \tau$.

PT-DEOWN By inversion, $\Gamma \vdash x@p' : \sim \tau$.
By induction, $\vdash \sim \sigma : \sim \tau$.
There is only one rule (ST-OWN) for deriving $\vdash \sim \sigma : \sim \tau$.
Thus, by inversion, $\vdash \sigma : \tau$.

PT-DEREF By inversion, $\Gamma \vdash x@p' : \& \ \ell \ q \ \tau$.
By induction, $\vdash \sim \sigma : \& \ \ell \ q \ \tau$.
There are no rules for deriving $\vdash \sim \sigma : \& \ \ell \ q \ \tau$.
Thus, by inversion, this case is impossible.

SI-DEREF Then $p = *p'$.
By inversion, $\textsc{shallow}(\Upsilon, x@p', \& \ q \ \tau)$ and $\textsc{init}(\tau) = \sigma$.
There are two possible rules for deriving $\Gamma \vdash x@*p' : \tau$.

PT-DEOWN By inversion, $\Gamma \vdash x@p' : \sim \tau$.
By induction, $\vdash \& \ q \ \tau : \sim \tau$.
There are no rules for deriving $\vdash \& \ q \ \tau : \sim \tau$.
Thus, by inversion, this case is impossible.

PT-DEREF By inversion, $\Gamma \vdash x@p' : \& \ \ell \ q \ \tau$.
Since $\vdash \Upsilon : \Gamma$ guarantees that $\Gamma(x)$ is closed and a simple inspection of the path typing rules shows that this implies all well typed paths have closed types, we know that τ is closed.
By INIT Type Preservation, $\textsc{init}(\tau) = \sigma$ implies $\vdash \sigma : \tau$.

SI-PROJ Then $p = p' \cdot i$ and $\sigma = \sigma_i$.
By inversion, $\textsc{shallow}(\Upsilon, x@p', [\sigma_i]_{i \in \{1 \ldots n\}})$.
There is only one rule (PT-PROJ) for deriving $\Gamma \vdash x@p' \cdot i : \tau$.
Thus, by inversion, we have $\tau = \tau_i$ and $\Gamma \vdash x@p' : [\tau_i]_{i \in \{1 \ldots n\}}$.
By induction, we have $\vdash [\sigma_i]_{i \in \{1 \ldots n\}} : [\tau_i]_{i \in \{1 \ldots n\}}$.
There is only one rule (ST-REC) for deriving $\vdash [\sigma_i]_{i \in \{1 \ldots n\}} : [\tau_i]_{i \in \{1 \ldots n\}}$.
Thus, by inversion, we have $\vdash \sigma_i : \tau_i$.

SI-UNROLL Then $p = \mathsf{unroll}\ [\tau']\ p'$.
By inversion, $\textsc{shallow}(\Upsilon, x@p', \mathsf{roll}\ [\tau'']\ \sigma)$.
There is only one rule (PT-UNROLL) for deriving $\Gamma \vdash x@\mathsf{unroll}\ [\tau']\ p' : \tau$.
By inversion, $\tau' = \mu X.\tau'''$, $\tau = [X \mapsto \mu X.\tau''']\tau'''$, and $\Gamma \vdash x@p' : \mu X.\tau'''$.
By induction, $\vdash \mathsf{roll}\ [\tau'']\ \sigma : \mu X.\tau'''$.

There is only one rule (ST-ROLL) for deriving this.
Thus, by inversion, we have $\tau'' = \mu X.\tau'''$ and $\vdash \sigma : [X \mapsto \mu X.\tau''']\tau'''$.

□

Shadow Checking the Heap

Of course, we also need to check that our shadow heap correctly models the actual heap. For the most part, it is exactly the relation you would expect. The use of INIT in LS-REF and LS-VAR ensures that the pointees of borrowed references and the payload of initialized variants are fully initialized. The DROPPABLE(l) judgment checks that l owns no initialized heap memory, i.e. $\sim \alpha$ is not reachable from l except through references. These layouts are safe to deallocate because we cannot leak heap memory by doing so. The $V \vdash H : \Upsilon$ judgment ensures that the shadow heap corresponds to the real heap. It also ensures that all heap memory is reachable from the stack and that every heap allocation has at least one owner: either a stack variable or a unique pointer.

$\boxed{\text{DROPPABLE}(l)}$

$$\frac{}{\text{DROPPABLE}(\mathsf{void})}\ \text{D-VOID} \qquad \frac{}{\text{DROPPABLE}(z)}\ \text{D-INT} \qquad \frac{}{\text{DROPPABLE}(\&\ q\ \alpha@r)}\ \text{D-REF} \qquad \frac{\text{DROPPABLE}(c) \qquad \text{DROPPABLE}(l)}{\text{DROPPABLE}(\langle c : l \rangle)}\ \text{D-VAR}$$

$$\frac{\forall i.\ \text{DROPPABLE}(l_i)}{\text{DROPPABLE}([l_i]_{i \in \{1 \ldots n\}})}\ \text{D-REC} \qquad \frac{\text{DROPPABLE}(l)}{\text{DROPPABLE}(\mathsf{roll}\ [\tau]\ l)}\ \text{D-ROLL}$$

$\boxed{H \vdash l : \sigma}$

$$\frac{}{H \vdash \mathsf{void} : \mathsf{uninit}}\ \text{LS-VOID} \qquad \frac{}{H \vdash z : \mathsf{int}}\ \text{LS-INT} \qquad \frac{\text{READ}(H, \alpha@\bullet, l) \qquad H \vdash l : \sigma}{H \vdash \sim \alpha : \sim \sigma}\ \text{LS-OWN} \qquad \frac{\text{READ}(H, \alpha@r, l) \qquad H \vdash l : \text{INIT}(\tau)}{H \vdash \&\ q\ \alpha@r : \&\ q\ \tau}\ \text{LS-REF}$$

$$\frac{}{H \vdash \langle \mathsf{void} : \mathsf{void} \rangle : \mathsf{uninit}}\ \text{LS-VOIDVAR} \qquad \frac{\text{DROPPABLE}(l)}{H \vdash \langle i : l \rangle : \mathsf{uninit}}\ \text{LS-DROPVAR} \qquad \frac{H \vdash l : \text{INIT}(\tau_i)}{H \vdash \langle i : l \rangle : \langle \tau_i \rangle_{i \in \{1 \ldots n\}}}\ \text{LS-VAR} \qquad \frac{\forall i.\ H \vdash l_i : \sigma_i}{H \vdash [l_i]_{i \in \{1 \ldots n\}} : [\sigma_i]_{i \in \{1 \ldots n\}}}\ \text{LS-REC}$$

$$\frac{H \vdash l : \sigma}{H \vdash \mathsf{roll}\ [\mu X.\tau]\ l : \mathsf{roll}\ [\mu X.\tau]\ \sigma}\ \text{LS-ROLL}$$

$$\frac{\begin{array}{l}\operatorname{dom} V = \operatorname{dom} \Upsilon \\ \operatorname{dom} H = \text{REACHABLE}(V, H) \\ \forall \alpha \in \operatorname{dom} H. \exists x. V(x) = \alpha \vee \exists \alpha'@r. \text{READ}(H, \alpha'@r, \sim \alpha) \\ \forall x \in \operatorname{dom} V.\ H \vdash H(V(x)) : \Upsilon(x)\end{array}}{V \vdash H : \Upsilon}$$

Shadow Preservation

The initialization data, in the form of a shadow, is preserved by path evaulation and READ so that the resulting layout is correctly described by the shadow.

Lemma (Shadow Preservation)**.** *If* $V \vdash H : \Upsilon$, $\textsc{shallow}(\Upsilon, x@p, \sigma)$, $V; H \vdash x@p \to \alpha@r$, *and* $\textsc{read}(H, \alpha@r, l)$ *then* $H \vdash l : \sigma$.

Shadow Preservation. Induction on the derivation fo $V; H \vdash x@p \to \alpha@r$.

PE-BASE Then $p = \bullet$ and $r = \bullet$.
By inversion, $V(x) = \alpha$.
There is only one rule (SI-BASE) for deriving $\textsc{shallow}(\Upsilon, x@\bullet, \sigma)$.
Thus, by inversion, $\Upsilon(x) = \sigma$.
There is only one rule (RD-BASE) for deriving $\textsc{read}(H, \alpha@\bullet, l)$.
Thus, by inversion, $H(\alpha) = l$.
By $V \vdash H : \Upsilon$, we have $H \vdash H(V(x)) : \Upsilon(x)$.
However, this is just $H \vdash l : \sigma$ as required.

PE-DEOWN Then $p = *p'$ and $r = \bullet$.
By inversion, $V; H \vdash x@p' \to \alpha'@r'$ and $\textsc{read}(H, \alpha'@r', \sim \alpha)$.
There are two possible rules for deriving $\textsc{shallow}(\Upsilon, x@*p', \sigma)$.

SI-DEOWN By inversion, $\textsc{shallow}(\Upsilon, x@p', \sim \sigma)$.
By induction, $H \vdash \sim \alpha : \sim \sigma$.
There is only one rule (LS-OWN) for deriving this.
Thus, by inversion, we have $\textsc{read}(H, \alpha@\bullet, l')$ and $H \vdash l' : \sigma$.
By Read Uniqueness, we have $l' = l$.
Ergo, $H \vdash l : \sigma$ as required.

SI-DEREF By inversion, $\textsc{shallow}(\Upsilon, x@p', \& \ q \ \tau)$ and $\textsc{init}(\tau) = \sigma$.
By induction, $H \vdash \sim \alpha : \& \ q \ \tau$.
There are no rules for deriving this.
Thus, by inversion, this case is impossible.

PE-DEREF Then $p = *p'$.
By inversion, $V; H \vdash x@p' \to \alpha'@r'$ and $\textsc{read}(H, \alpha'@r', \& \ q \ \alpha@r)$.
There are two possible rules for deriving $\textsc{shallow}(\Upsilon, x@*p', \sigma)$.

SI-DEOWN By inversion, $\textsc{shallow}(\Upsilon, x@p', \sim \sigma)$.
By induction, $H \vdash \& \ q \ \alpha@r : \sim \sigma$.
There are no rules for deriving this.
Thus, by inversion, this case is impossible.

SI-DEREF By inversion, $\textsc{shallow}(\Upsilon, x@p', \& \ q \ \tau)$ and $\textsc{init}(\tau) = \sigma$.
By induction, $H \vdash \& \ q \ \alpha@r : \& \ q \ \tau$.
There is only one rule (LS-REF) for deriving this.
Thus, by induction, we have $\textsc{read}(H, \alpha@r, l')$ and $H \vdash l' : \textsc{init}(\tau)$.
By Read Uniqueness, we have $l' = l$.
Ergo, $H \vdash l : \sigma$ as required.

PE-PROJ Then $p = p' \cdot i$ and $r = r' \cdot i$.
By inversion, $V; H \vdash x@p' \to \alpha@r'$.
There is only one rule (SI-PROJ) for deriving $\textsc{shallow}(\Upsilon, x@p' \cdot i, \sigma)$.
Thus, by inversion, we have $\sigma = \sigma_i$ and $\textsc{shallow}(\Upsilon, x@p', [\sigma_i]_{i \in \{1 \ldots n\}})$.
There is only one rule (RD-PROJ) for deriving $\textsc{read}(H, \alpha@r' \cdot i, l)$.
Thus, by inversion, we have $l = l_i$ and $\textsc{read}(H, \alpha@r', [l_i]_{i \in \{1 \ldots n\}})$.
By induction, we have $H \vdash [l_i]_{i \in \{1 \ldots n\}} : [\sigma_i]_{i \in \{1 \ldots n\}}$.
There is only one rule (LS-REC) for deriving this.
Thus, by inversion, we have $H \vdash l_i : \sigma_i$ as required.

PE-UNROLL Then $p = \mathsf{unroll}\ [\tau]\ p'$ and $r = \mathsf{unroll}\ [\tau]\ r'$.
By inversion, $V; H \vdash x@p' \to \alpha@r'$.
There is only one rule (SI-UNROLL) for deriving $\textsc{shallow}(\Upsilon, x@\mathsf{unroll}\ [\tau]\ p', \sigma)$.
Thus, by inversion, we have $\textsc{shallow}(\Upsilon, x@p', \mathsf{roll}\ [\tau']\ \sigma)$.
There is only one rule (RD-UNROLL) for deriving $\textsc{read}(H, \alpha@\mathsf{unroll}\ [\tau]\ r', l)$.
Thus, by inversion, we have $\textsc{read}(H, \alpha@r', \mathsf{roll}\ [\tau'']\ l)$.
By induction, $H \vdash \mathsf{roll}\ [\tau'']\ l : \mathsf{roll}\ [\tau']\ \sigma$.
There is only one rule (LS-ROLL) for deriving this.
Thus, by inversion, we have $\tau' = \tau'' = \mu X.\tau'''$ and $H \vdash l : \sigma$ required.

□

Path Progress

Finally, we have all we need to proof path evaluation progress under the right circumstances. Assuming all our contexts and the heap are well-formed, then a shallowly initialized, well-typed path can evaluate to some route. We can then rely on path preservation to show that route has the same type as the path. Read safety then lets us show that we can read a layout at that route and that it has the same type. Shadow preservation guarantees us that our model of the layout reflects reality. All of the key pieces we need for the higher layers of Patina.

Lemma (Path Progress)**.** *If* $\vdash H : \Sigma; \Psi$*,* $\Gamma; V \vdash \Sigma$*,* $\vdash \Upsilon : \Gamma$*,* $V \vdash H : \Upsilon$*,* $\Gamma \vdash x@p : \tau$*, and* $\textsc{shallow}(\Upsilon, x@p, \sigma)$ *then there is some* $\alpha@r$ *such that* $V; H \vdash x@p \to \alpha@r$*.*

Path Progress. Induction on the derivation of $\Gamma \vdash x@p : \tau$.

PT-BASE Then $p = \bullet$.
By inversion, $\Gamma(x) = \tau$.
From $\Gamma; V \vdash \Sigma$, we have $\Gamma(x) = \Sigma(V(x))$.
Ergo, $\exists \alpha.\ V(x) = \alpha$.
Then by PE-BASE, we have $V; H \vdash x@\bullet \to \alpha@\bullet$.

PT-DEOWN Then $p = *p'$.
By inversion, $\Gamma \vdash x@p' : \sim \tau$.
There are two possible rules for deriving $\textsc{shallow}(\Upsilon, x@*p', \sigma)$.

SI-DEOWN By inversion, $\textsc{shallow}(\Upsilon, x@p', \sim \sigma)$.
By induction, $\exists \alpha@r.\ V; H \vdash x@p' \to \alpha@r$.
By Path Type Preservation, $\Sigma; \Psi \vdash \alpha@r : \sim \tau$.
By Read Safety, $\textsc{read}(H, \alpha@r, l)$ and $\Sigma; \Psi \vdash l : \sim \tau$ for some l.
By Shadow Preservation, $H \vdash l : \sim \sigma$.
There is only one rule (LS-OWN) for deriving this.
Thus, by inversion, we have $l = \sim \alpha'$, $\textsc{read}(H, \alpha'@\bullet, l')$, and $H \vdash l' : \sigma$.
Using PE-DEOWN with $V; H \vdash x@p' \to \alpha@r$ and $\textsc{read}(H, \alpha@r, \sim \alpha')$,
we get $V; H \vdash x@*p' \to \alpha'@\bullet$, which is what was required.

SI-DEREF By inversion, $\textsc{shallow}(\Upsilon, x@p', \&\ q\ \tau)$.
By $\textsc{shallow}$ Type Preservation, $\vdash \&\ q\ \tau : \sim \tau$.
There are no rules for deriving this.
Thus, by inversion, this case is impossible.

PT-DEREF Then $p = *p'$.
By inversion, $\Gamma \vdash x@p' : \&\ \ell\ q\ \tau$.
There are two possible rules for deriving $\textsc{shallow}(\Upsilon, x@*p', \sigma)$.

SI-DEOWN By inversion, $\textsc{shallow}(\Upsilon, x@p', \sim \sigma)$.
By SHALLOW Type Preservation, $\vdash \sim \sigma : \& \ \ell \ q \ \tau$.
There are no rules for deriving this.
Thus, by inversion, this case is impossible.

SI-DEREF By inversion, $\textsc{shallow}(\Upsilon, x@p', \& \ q \ \tau)$.
By induction, $\exists \alpha@r.\ V; H \vdash x@p' \to \alpha@r$.
By Path Type Preservation, $\Sigma; \Psi \vdash \alpha@r : \& \ \ell \ q \ \tau$.
By Read Safety, $\textsc{read}(H, \alpha@r, l)$ and $\Sigma; \Psi \vdash l : \& \ \ell \ q \ \tau$ for some l.
By Shadow Preservation, $H \vdash l : \& \ q \ \tau$.
There is only one rule (LS-REF) for deriving this.
Thus, by inversion, we have $l = \& \ q \ \alpha'@r'$ and $\textsc{read}(H, \alpha'@r', l')$.
Using PE-DEREF with $V; H \vdash x@p' \to \alpha@r$ and $\textsc{read}(H, \alpha@r, \& \ q \ \alpha'@r')$,
we get $V; H \vdash x@*p' \to \alpha'@r'$, which is what was required.

PT-PROJ Then $p = p' \cdot i$ and $\tau = \tau_i$.
By inversion, $\Gamma \vdash x@p' : [\tau_i]_{i \in \{1 \ldots n\}}$.
There is only one rule (SI-PROJ) for deriving $\textsc{shallow}(\Upsilon, x@*p', \sigma)$.
Thus, by inversion, we have $\sigma = \sigma_i$ and $\textsc{shallow}(\Upsilon, x@p', [\sigma_i]_{i \in \{1 \ldots n\}})$.
By induction, $\exists \alpha@r.\ V; H \vdash x@p' \to \alpha@r$.
Then by PE-PROJ, we have $V; H \vdash x@p' \cdot i \to \alpha@r \cdot i$.

PT-UNROLL Then $p = \mathsf{unroll}\ [\mu X.\tau']\ p'$ and $\tau = [X \mapsto \mu X.\tau']\tau'$.
By inversion, $\Gamma \vdash x@p' : \mu X.\tau'$.
There is only one rule (SI-UNROLL) for deriving $\textsc{shallow}(\Upsilon, x@\mathsf{unroll}\ [\mu X.\tau']\ p', \sigma)$.
Thus, by inversion, $\textsc{shallow}(\Upsilon, x@p', \mathsf{roll}\ [\tau'']\ \sigma)$.
By induction, $\exists \alpha@r.\ V; H \vdash x@p' \to \alpha@r$.
Then by PE-UNROLL, we have $V; H \vdash x@\mathsf{unroll}\ [\mu X.\tau']\ p' \to \alpha@\mathsf{unroll}\ [\mu X.\tau']\ r'$.

□

R-Values

R-values in Patina are expressions that can evaluate to a layout. This layer of Patina involves the borrow checker, which ensures the borrowed reference guarantees. In order to focus on the details of the borrow checker, we will pare down our types. Variants, tuples, and recursive types have straightforward behavior here and would unnecessarily clutter the discussion. Going forward, we will use this subset of our earlier language:

Context	Γ	:	Variable $\to$ Type
Shadow Heap	Υ	:	Variable $\to$ Shadow
Map	V	:	Variable $\to$ Allocation
Heap Type	Σ	:	Allocation $\to$ Type
Heap	H	:	Allocation $\to$ Layout

Lifetime	ℓ		
Variable	x		
Integer	z		
Allocation	α		

Qualifier	q	::=	$\mathsf{imm} \mid \mathsf{mut}$
Type	τ	::=	$\mathsf{int} \mid \sim \tau \mid \& \ \ell \ q \ \tau$
Path	p	::=	$\bullet \mid * p$
Cell	c	::=	$\mathsf{void} \mid z \mid \sim \alpha \mid \& \ q \ \alpha$
Layout	l	::=	c
Hole	h	::=	$\mathsf{uninit} \mid \mathsf{int} \mid \sim \sigma \mid \& \ q \ \tau$
Shadow	σ	::=	h

Routes would only consist of $\bullet$, so they are now elided. Without variants, the Ψ context is irrelevant, so it is elided as well.

We define three kinds of expressions: integer constants, using the value at an L-value, and creating a borrowed reference to the value at an L-value.

$$\text{Expression} \quad e \quad ::= \quad z \mid x@p \mid \& \ \ell \ q \ x@p$$

Tracking Loans[1]

When we use an L-value in an expression, we must ensure the operation we are performing on it does not violate the promises of existing loans, and we must ensure that the value it refers to is fully initialized. When we create a borrowed reference, we must ensure that the promises of the borrow can actually be upheld for the duration of the borrow. In order to make these checks, we need to know what is borrowed, how it is borrowed, and for how long it is borrowed. We will accomplish this by adding a stack of loans to every level of the shadow heap:

Hole	h	::=	$\mathsf{uninit} \mid \mathsf{int} \mid \sim \sigma \mid \& \ q \ \sigma$
Bank	\$	::=	$\emptyset \mid \$, (\ell, q)$
Shadow	σ	::=	$\$: h$

Each pair (ℓ, q) describes the duration and manner of a loan, with more recent loans at the front of the stack. Since every path can be borrowed, every path must have a loan stack (or bank) associated with it. The easiest way to do that is to tag every shadow with a bank. Note that the borrowed reference hole has changed. When we only cared about initialization, we used the hole $\& \ q \ \tau$. Since everything reachable via the borrow was initialized, we did not need to record the initialization state. Now, however, we do need to record the loan state of things reachable via the borrow. Thus, we make the reference hole recursive: $\& \ q \ \sigma$.

[1] Rust itself and earlier models of Patina used a dramatically different form of loan tracking and borrow checking. A universal set of loans is kept and threaded through the type checker. Each loan generates a set of paths that cannot be accessed without violating a loan guarantee. The borrow checker would then make sure that no path used was in that set of forbidden paths.

This structure does not lend itself to proofs, so we devised this alternate system. The system we present below is syntax driven and, we think, easier to understand. Since proving this system matches the original would run into the problem we tried to avoid in the first place, we satisfied ourselves with testing the two implementations. The original model and this model agree on all the tests used in the original model.

Shadow Typing with Banks

Since our shadows now include loan information, we need to modify our shadow typing. While we have included lifetimes in our model from the beginning, this is the first time we will actually utilize them. Before we can discuss how to check banks, we must first discuss what lifetimes are, where they come from, and what they give us.

The Lifetime Relation

Lifetimes, as the name implies, encompass a notion of duration. Specifically, they describe the duration a value is allocated and the duration of a loan. Each new block of scope defines a new lifetime. So there is a clear ordering on lifetimes, $\ell \leq \ell'$ if the duration of ℓ is not longer than the duration of ℓ'. More formally:

$$\mathscr{L} \subseteq \text{Lifetime} \times \text{Lifetime}$$

$$\frac{\mathscr{L} \text{ is a poset}}{\vdash \mathscr{L}}$$

where $(\ell, \ell') \in \mathscr{L}$ means $\ell \leq \ell'$. Rust allows some other sources of lifetimes not relevant to the current model: lifetime parameters for functions and *static*, which is the oldest lifetime and typically used for string constants. Though the lifetimes in our model form a total order, the general Rust lifetime relation is only a partial order. Since we intend this model to be a potential basis for all of Rust, we will ignore the totality of our lifetimes.

Bank Coherence

As mentioned before, lifetimes come from scope blocks. This means that every variable has a lifetime, which comes from the block it was defined in. Since variables are deallocated at the end of the block in which they are defined, that lifetime is an accurate model. We record the lifetimes of variables in a new context:

$$L : \text{Variable} \to \text{Lifetime}$$

Furthermore, if variables only exist for a particular lifetime, then it makes sense that loans of memory owned by the variable cannot be longer than the lifetime of the variable. Otherwise the deallocation of the variable could create dangling pointers elsewhere.

Finally, we must ensure that more recent loans are not inconsistent with older loans of the same path. If a path is already loaned out for some lifetime ℓ and we loan it out again for some lifetime ℓ', then $(\ell', \ell) \in \mathscr{L}$. A loan only controls a value during the loan's lifetime, so it cannot make promises for lifetimes longer than that. Additionally, given a mutable loan we can reloan it immutably, but not the other way. A mutable loan knows it has unique access to a value, so it can guarantee the value can be immutable if it wishes. On the other hand, an immutable loan promises that the value will not be changed for its duration, which violates the requirement that a mutable loan should be able to mutate its referent. We characterize this relationship with an ordering on qualifiers: $\mathsf{imm} < \mathsf{mut}$.

$\boxed{\mathscr{L} \vdash \$ \leq \ell}$

$$\frac{}{\mathscr{L} \vdash \emptyset \leq \ell}\ \text{BWF-NONE} \qquad \frac{(\ell', \ell) \in \mathscr{L}}{\mathscr{L} \vdash \emptyset, (\ell', q) \leq \ell}\ \text{BWF-ONE} \qquad \frac{(\ell_2, \ell_1) \in \mathscr{L} \qquad q_2 \leq q_1 \qquad \mathscr{L} \vdash \$, (\ell_1, q_1) \leq \ell}{\mathscr{L} \vdash \$, (\ell_1, q_1), (\ell_2, q_2) \leq \ell}\ \text{BWF-TWO}$$

Bounded Shadow Typing

We need to update our shadow typing judgment to support checking bank coherence, but we need to check two new things as well. First, we will require that variables can only reference memory that they will not outlast. This means we require borrowed references to have lifetimes not shorter than the bounding lifetime, which is the reverse of the requirement on loans. This should make sense, we can only use memory that will exist at least as long as us, but we can only loan out our memory to others that will not last longer than us.

Second, we need to make sure that more specific loans do not contradict more general loans. For example, if a variable x is loaned immutably, then a mutable loan of $*x$ would not make sense. The immutable loan of x promises that no memory reachable via x will change, but the mutable loan of $*x$ promises that the memory reachable via $*x$ can be changed through the borrowed reference. Similarly, if x was loaned mutably, then $*x$ could not be loaned at all. The loan of x promises that only the borrowed reference has access to memory reachable from x, but the loan of $*x$ creates an alias to part of that memory. In general, a wider immutable loan requires more specific loans to not be mutable loans, and a wider mutable loan prevents any more specific loans.

$$\text{Maybe Qualifier} \quad mq \quad ::= \quad \mathsf{none} \mid q$$

$\boxed{\textsc{not-mut}(\$)}$

$$\text{NM-EMPTY}\ \frac{}{\textsc{not-mut}(\emptyset)} \qquad \text{NM-IMM}\ \frac{}{\textsc{not-mut}(\$, (\ell, \mathsf{imm}))}$$

$\boxed{\mathscr{L} \vdash \sigma : \tau \text{ controlled by } mq \leq \ell}$

$$\text{ST-NONE}\ \frac{\mathscr{L} \vdash \emptyset \leq \ell \qquad \mathscr{L} \vdash h : \tau \text{ controlled by } \mathsf{none} \leq \ell}{\mathscr{L} \vdash (\emptyset : h) : \tau \text{ controlled by } \mathsf{none} \leq \ell} \qquad \text{ST-SOME}\ \frac{\mathscr{L} \vdash \$, (\ell', q) \leq \ell \qquad \mathscr{L} \vdash h : \tau \text{ controlled by } q \leq \ell}{\mathscr{L} \vdash (\$, (\ell', q) : h) : \tau \text{ controlled by } \mathsf{none} \leq \ell}$$

$$\text{ST-IMM}\ \frac{\mathscr{L} \vdash \$ \leq \ell \qquad \textsc{not-mut}(\$) \qquad \mathscr{L} \vdash h : \tau \text{ controlled by } \mathsf{imm} \leq \ell}{\mathscr{L} \vdash (\$: h) : \tau \text{ controlled by } \mathsf{imm} \leq \ell}$$

$$\text{ST-MUT}\ \frac{\mathscr{L} \vdash \emptyset \leq \ell \qquad \mathscr{L} \vdash h : \tau \text{ controlled by } \mathsf{mut} \leq \ell}{\mathscr{L} \vdash (\emptyset : h) : \tau \text{ controlled by } \mathsf{mut} \leq \ell}$$

$\boxed{\mathscr{L} \vdash h : \tau \text{ controlled by } mq \leq \ell}$

$$\text{HT-UNINT}\quad \frac{}{\mathscr{L} \vdash \mathsf{uninit} : \mathsf{int} \text{ controlled by } mq \leq \ell} \qquad \text{HT-UNOWN}\quad \frac{}{\mathscr{L} \vdash \mathsf{uninit} : \sim \tau \text{ controlled by } mq \leq \ell}$$

$$\text{HT-UNREF}\quad \frac{(\ell, \ell') \in \mathscr{L}}{\mathscr{L} \vdash \mathsf{uninit} : \& \ \ell' \ q \ \tau \text{ controlled by } mq \leq \ell} \qquad \text{HT-INT}\quad \frac{}{\mathscr{L} \vdash \mathsf{int} : \mathsf{int} \text{ controlled by } mq \leq \ell}$$

$$\text{HT-OWN}\quad \frac{\mathscr{L} \vdash \sigma : \tau \text{ controlled by } mq \leq \ell}{\mathscr{L} \vdash \sim \sigma : \sim \tau \text{ controlled by } mq \leq \ell} \qquad \text{HT-REFMUT}\quad \frac{(\ell, \ell') \in \mathscr{L} \qquad \mathscr{L} \vdash \sigma : \tau \text{ controlled by } mq \leq \ell \qquad \sigma \text{ init}}{\mathscr{L} \vdash \& \ \mathsf{mut} \ \sigma : \& \ \ell' \ \mathsf{mut} \ \tau \text{ controlled by } mq \leq \ell}$$

$$\text{HT-REFIMM}\quad \frac{(\ell, \ell') \in \mathscr{L} \qquad \mathscr{L} \vdash \sigma : \tau \text{ controlled by } \mathsf{imm} \leq \ell \qquad \sigma \text{ init}}{\mathscr{L} \vdash \& \ \mathsf{imm} \ \sigma : \& \ \ell' \ \mathsf{imm} \ \tau \text{ controlled by } mq \leq \ell}$$

where "σ init" in HT-REFMUT and HT-REFIMM is a judgment that ensures a shadow is fully initialized, i.e. it cannot reach a uninit. Note that the interior of immutable borrowed references is always controlled by an immutable loan. This prevents mutable loans of the interior of immutable borrows, which are nonsensical.

Bounded Shadow Heap Well-Formedness

With our shadow typing updated to check banks, we can update what we mean by a well-formed shadow heap. The lifetime relation and the lifetime map are now included. We ensure the lifetime map is well-formed by requiring its domain to be the same as the domain of the type context and the shadow heap, and by requiring that every lifetime in its range to be in the lifetime relation.

$$\frac{\begin{array}{l}\operatorname{dom}\Gamma = \operatorname{dom}\Upsilon = \operatorname{dom} L \\ \forall x \in \operatorname{dom}\Gamma.\ \Gamma(x) \text{ closed} \\ \forall x \in \operatorname{dom} L.\ (L(x), L(x)) \in L \\ \forall x \in \operatorname{dom}\Upsilon.\ \mathscr{L} \vdash \Upsilon(x) : \Gamma(x) \text{ controlled by } \mathsf{none} \leq L(x)\end{array}}{\mathscr{L} \vdash \Upsilon : \Gamma \leq L}$$

Ensuring Operations Do Not Violate Loans

The most important duty of the borrow checker is making sure that performing some operation does not violate the promises of existing loans. We cannot read a path that has promised unique access to another. We cannot write to a path that has promised to remain unchanged or that we cannot read. We cannot move from a path that we do not own or that we cannot write.

Additionally, we can only use a path that is fully initialized, or our program might try to touch an uninitialized value and get stuck. This means that $\textsc{shallow}(\Upsilon, x@p, \sigma)$ holds and that σ init holds. That is, enough memory is initialized to evaluate the L-value to a memory location, and everything reachable from that location is initialized. We use these dual judgments as a base to define our validity checks for reading, writing, and moving. Reading will be more restricted than simple full initialization. Writing will be more restricted than reading. Moving will be more restricted than writing.

Reading

The restriction we add to full initialization to verify that reading a path is valid is that no part of the path or anything reachable from the path has been mutable loaned. A mutable loan promises unique access via the corresponding mutable borrowed reference, which precludes reading the loaned path directly. We can

also short-circuit our check in a two cases. If the path we want to read dereferences an immutable borrow, then we do not need to check the interior of the borrow. By its nature, the immutable borrow guarantees readability, and any mutable loaning of the interior is already forbidden. For similar reasons, when checking the memory reachable from the path we do not need to examine the interior of immutable borrows.

$\boxed{\textsc{contains-no-mut-loans}(\sigma)}$

$$\frac{\textsc{not-mut}(\$)}{\textsc{contains-no-mut-loans}(\$: \mathsf{int})}\ \textsc{cnml-int} \qquad \frac{\textsc{not-mut}(\$) \quad \textsc{contains-no-mut-loans}(\sigma)}{\textsc{contains-no-mut-loans}(\$:\sim \sigma)}\ \textsc{cnml-own}$$

$$\frac{\textsc{not-mut}(\$)}{\textsc{contains-no-mut-loans}(\$: \& \ \mathsf{imm}\ \sigma)}\ \textsc{cnml-refimm} \qquad \frac{\textsc{not-mut}(\$) \quad \textsc{contains-no-mut-loans}(\sigma)}{\textsc{contains-no-mut-loans}(\$: \& \ \mathsf{mut}\ \sigma)}\ \textsc{cnml-refmut}$$

$\boxed{\textsc{can-read}(p, \sigma)}$

$$\frac{\textsc{contains-no-mut-loans}(\sigma)}{\textsc{can-read}(\bullet, \sigma)}\ \textsc{cr-base} \qquad \frac{\textsc{not-mut}(\$) \quad \textsc{can-read}(p, \sigma)}{\textsc{can-read}(*p, \$:\sim \sigma)}\ \textsc{cr-deown} \qquad \frac{\textsc{not-mut}(\$)}{\textsc{can-read}(*p, \$: \& \ \mathsf{imm}\ \sigma)}\ \textsc{cr-derefimm}$$

$$\frac{\textsc{not-mut}(\$) \quad \textsc{can-read}(p, \sigma)}{\textsc{can-read}(*p, \$: \& \ \mathsf{mut}\ \sigma)}\ \textsc{cr-derefmut}$$

where one would check that $x@p$ is readable by $\textsc{can-read}(p, \Upsilon(x))$.

Writing

In order to safely write a path, no part of the path or anything reachable from the path can be loaned at all. An immutable loan implies the data is frozen, which prevents us from writing. A mutable loan implies something else has unique acces to the data, which also prevents us from writing. Additionally, we cannot write to the interior of an immutable borrow, even if we can write to the borrowed reference itself. Memory reachable from that immutable borrow is frozen.

$\boxed{\textsc{contains-no-loans}(\sigma)}$

$$\frac{}{\textsc{contains-no-loans}(\emptyset : \mathsf{int})}\ \textsc{cnl-int} \qquad \frac{\textsc{contains-no-loans}(\sigma)}{\textsc{contains-no-loans}(\emptyset :\sim \sigma)}\ \textsc{cnl-own} \qquad \frac{\textsc{contains-no-loans}(\sigma)}{\textsc{contains-no-loans}(\emptyset : \& \ q \ \sigma)}\ \textsc{cnl-ref}$$

Note that the borrowed reference cases of have merged again after being separate for reading. We cannot guarantee that the interior of an immutable borrow will not be loaned out without actually checking. However, note that $\textsc{contains-no-loans}(\sigma)$ implies $\textsc{contains-no-mut-loans}(\sigma)$.

$\boxed{\textsc{can-write}(p, \sigma)}$

$$\frac{\textsc{contains-no-loans}(\sigma)}{\textsc{can-write}(\bullet, \sigma)}\ \textsc{cw-base} \qquad \frac{\textsc{can-write}(p, \sigma)}{\textsc{can-write}(*p, \emptyset :\sim \sigma)}\ \textsc{cw-deown} \qquad \frac{\textsc{can-write}(p, \sigma)}{\textsc{can-write}(*p, \emptyset : \& \ \mathsf{mut}\ \sigma)}\ \textsc{cw-derefmut}$$

Again, note that $\textsc{can-write}(p, \sigma)$ implies $\textsc{can-read}(p, \sigma)$. Rust requires that the ability to write implies the ability to read, which is reflected in our judgments.

Moving

The final operation, moving out of a path, requires ownership over the path. Moving is used when the value being read has an affine type, which prevents it from being merely copied. The old location of the value must be purged. This requires the ability to write, but since it leaves the original path uninitialized it cannot be used via mutable borrows. Doing so would violate the guarantee that borrowed references always point to initialized memory.

$\boxed{\textsc{can-move}(p, \sigma)}$

$$\frac{\textsc{contains-no-loans}(\sigma)}{\textsc{can-move}(\bullet, \sigma)}\ \textsc{CM-BASE} \qquad \frac{\textsc{can-move}(p, \sigma)}{\textsc{can-move}(*p, \emptyset :\sim \sigma)}\ \textsc{CM-DEOWN}$$

Note that $\textsc{can-move}(p, \sigma)$ implies $\textsc{can-write}(p, \sigma)$.

Affine Types and Copyability

Values of an affine type cannot be used more than once, e.g. unique pointers. We model this in Patina by deinitializing an affine value on use. However, we do not want to do this unnecessarily so we need to distinguish between affine and non-affine types. Affine types must be moved, but non-affine types can be merely copied. In Patina, unique pointers and borrowed mutable references are the sources of affineness. Unique pointers must be affine to preserve their uniqueness. Borrowed mutable references must be affine to preserve the unique access they promise.

$\boxed{\tau \text{ copy}}$

$$\frac{}{\mathsf{int} \text{ copy}}\ \textsc{C-INT} \qquad \frac{}{\& \ell\ \mathsf{imm}\ \tau \text{ copy}}\ \textsc{C-REFIMM}$$

$\boxed{\tau \text{ move}}$

$$\frac{}{\sim \tau \text{ move}}\ \textsc{M-OWN} \qquad \frac{}{\& \ell\ \mathsf{mut}\ \tau \text{ move}}\ \textsc{M-REFMUT}$$

The two judgments here are not particularly interesting without compound types, but they still create a necessary distinction. Note that these two judgments partition the set of types.

Using an L-value

Now that we can check that an operation is safe and we can determine from the type which operation to perform, we can specify how to check that using the value at an L-value is safe and track how doing so changes the shadow heap.

$\boxed{\Gamma; \Upsilon \vdash x@p : \tau \Rightarrow \Upsilon'}$

$$\frac{\Gamma \vdash x@p : \tau \qquad \tau \text{ copy} \qquad \textsc{can-read}(p, \Upsilon(x))}{\Gamma; \Upsilon \vdash x@p : \tau \Rightarrow \Upsilon}\ \textsc{UP-COPY}$$

$$\frac{\Gamma \vdash x@p : \tau \qquad \tau \text{ move} \qquad \textsc{can-move}(p, \Upsilon(x)) \qquad \textsc{use}(\Upsilon, x@p, \Upsilon')}{\Gamma; \Upsilon \vdash x@p : \tau \Rightarrow \Upsilon'}\ \textsc{UP-MOVE}$$

where $\textsc{use}(\Upsilon, x@p, \Upsilon')$ modifies the shadow of $x@p$ to deinitialize the affine values.

Conjecture (L-Value Use Shadow Preservation)**.** *If* $\vdash \mathscr{L}$*,* $\mathscr{L} \vdash \Upsilon : \Gamma \leq L$*, and* $\Gamma; \Upsilon \vdash x@p : \tau \Rightarrow \Upsilon'$ *then* $\mathscr{L} \vdash \Upsilon' : \Gamma \leq L$*.*

Justification. Using a path either leaves the shadow heap unchanged or deinitializes only a part of it, neither of which should change the types of shadows or the lifetimes of loans. Deinitializing part of the shadow heap could invalidate existing loans, but the CAN-READ and CAN-MOVE checks should rule out those cases. □

Ensuring Loan Promises Can Be Upheld

When creating a borrowed reference, the borrow checker must ensure that is it actually possible to make the promises that the loan demands. For example, we cannot promise the interior of a borrowed reference will be valid for longer than that reference's lifetime, nor can we promise that the interior of a immutable borrowed reference will have unique access.

Validity

Both kinds of borrowed references require that their referent be initialized for the lifetime of the reference, which here we call *validity*. Variables are valid for their entire lifetime and can be loaned out for any duration not longer than that. Unique pointers inherit the lifetime of their owner. Borrowed references specify how long they are valid by their lifetime, and they can be loaned out for any duration not longer than that. Since we already know the interior of a borrowed reference is valid for the lifetime of that reference, we do not need to examine further.

$\boxed{\Gamma; \mathscr{L}; L \vdash x@p \text{ valid for } \ell}$

$$\text{VF-BASE}\ \frac{(\ell, L(x)) \in \mathscr{L}}{\Gamma; \mathscr{L}; L \vdash x@\bullet \text{ valid for } \ell} \qquad \text{VF-DEOWN}\ \frac{\Gamma \vdash x@p : \sim \tau \qquad \Gamma; \mathscr{L}; L \vdash x@p \text{ valid for } \ell}{\Gamma; \mathscr{L}; L \vdash x@{*}p \text{ valid for } \ell}$$

$$\text{VF-DEREF}\ \frac{\Gamma \vdash x@p : \& \ \ell' \ q \ \tau \qquad (\ell, \ell') \in \mathscr{L}}{\Gamma; \mathscr{L}; L \vdash x@{*}p \text{ valid for } \ell}$$

Uniqueness

Mutable loans require the loaned memory have *unique* access. If the path being loaned has unique access, then it can give that access to the new mutable borrow. Variables, being top level, are inherently unique along with unique pointers. Mutable borrows provide unique access to their interiors for their duration. Immutable borrows are inherently not unique.

$\boxed{\Gamma; \mathscr{L} \vdash x@p \text{ unique for } \ell}$

$$\text{UF-BASE}\ \frac{}{\Gamma; \mathscr{L} \vdash x@\bullet \text{ unique for } \ell} \qquad \text{UF-DEOWN}\ \frac{\Gamma \vdash x@p : \sim \tau \qquad \Gamma; \mathscr{L} \vdash x@p \text{ unique for } \ell}{\Gamma; \mathscr{L} \vdash x@{*}p \text{ unique for } \ell}$$

$$\text{UF-DEREFMUT}\ \frac{\Gamma \vdash x@p : \& \ \ell' \ \mathsf{mut} \ \tau \qquad (\ell, \ell') \in \mathscr{L} \qquad \Gamma; \mathscr{L} \vdash x@p \text{ unique for } \ell}{\Gamma; \mathscr{L} \vdash x@{*}p \text{ unique for } \ell}$$

Freezability

Immutable loans require the loaned memory to be *frozen*, unchangable by anything, for the duration of the loan. When creating an immutable borrow, we need to check a slightly weaker property: *freezable*, unchangable by anything *else*. A freezable path can be mutated, but only by the path itself. If the path does not mutate itself, e.g. by loaning itself immutably, then we know it is frozen. Unique paths are inherently freezable since no other path exists. Dereferencing through an immutable borrow, while not unique, is freezable because the interior of the borrow is already guaranteed to be frozen for its lifetime.

$\boxed{\Gamma; \mathscr{L} \vdash x@p \text{ freezable for } \ell}$

$$\text{FF-BASE}\ \frac{}{\Gamma; \mathscr{L} \vdash x@\bullet \text{ freezable for } \ell} \qquad \text{FF-DEOWN}\ \frac{\Gamma \vdash x@p : \sim \tau \qquad \Gamma; \mathscr{L} \vdash x@p \text{ freezable for } \ell}{\Gamma; \mathscr{L} \vdash x@{*}p \text{ freezable for } \ell}$$

$$\text{FF-DEREFMUT}\ \frac{\Gamma \vdash x@p : \& \ell' \ \mathsf{mut}\ \tau \qquad (\ell, \ell') \in \mathscr{L} \qquad \Gamma; \mathscr{L} \vdash x@p \text{ freezable for } \ell}{\Gamma; \mathscr{L} \vdash x@{*}p \text{ freezable for } \ell}$$

$$\text{FF-DEREFIMM}\ \frac{\Gamma \vdash x@p : \& \ell' \ \mathsf{imm}\ \tau \qquad (\ell, \ell') \in \mathscr{L}}{\Gamma; \mathscr{L} \vdash x@{*}p \text{ freezable for } \ell}$$

Note how $\Gamma; \mathscr{L} \vdash x@p$ unique for ℓ implies $\Gamma; \mathscr{L} \vdash x@p$ freezable for ℓ. This implies that anything we can borrow mutably we can also borrow immutably, which accurately reflects Rust.

Expression Typing

With our model of the borrow checker in hand, we can finally specify the typing of expressions.

$\boxed{\Gamma; \mathscr{L}; L; \Upsilon \vdash e : \tau; \Upsilon'}$

Integer constants are trivial and do not change the shadow heap.

$$\text{ET-INT}\ \frac{}{\Gamma; \mathscr{L}; L; \Upsilon \vdash z : \mathsf{int}; \Upsilon}$$

Using the value at an L-value simply relies on our earlier judgment.

$$\text{ET-USE}\ \frac{\Gamma; \Upsilon \vdash x@p : \tau \Rightarrow \Upsilon'}{\Gamma; \mathscr{L}; L; \Upsilon \vdash x@p : \tau; \Upsilon'}$$

Creating an immutable borrowed reference requires checking that the immutable loan guarantees will be upheld for the entire duration of the loan. This involves checking that we can read the L-value now, and that we can guarantee its validity and freezability for the duration of the loan. We record the new loan in the output shadow heap.

$$\text{ET-REFIMM}\ \frac{\Gamma; \mathscr{L}; L \vdash x@p \text{ valid for } \ell \qquad \begin{array}{c}\Gamma \vdash x@p : \tau \qquad \textsc{can-read}(p, \Upsilon(x)) \\ \Gamma; \mathscr{L} \vdash x@p \text{ freezable for } \ell \qquad \textsc{record}(\Upsilon, x@p, (\ell, \mathsf{imm}), \Upsilon')\end{array}}{\Gamma; \mathscr{L}; L; \Upsilon \vdash \& \ \ell \ \mathsf{imm}\ x@p : \& \ \ell \ \mathsf{imm}\ \tau; \Upsilon'}$$

Similarly, creating a mutable borrowed reference requires checking the mutable loan guarantees. Instead of reading, we must be able to write. Instead of freezability, we require uniqueness.

ET-REFMUT

$$\frac{\Gamma \vdash x@p : \tau \qquad \textsc{can-write}(p, \Upsilon(x)) \\ \Gamma; \mathscr{L}; L \vdash x@p \text{ valid for } \ell \qquad \Gamma; \mathscr{L} \vdash x@p \text{ unique for } \ell \qquad \textsc{record}(\Upsilon, x@p, (\ell, \mathsf{mut}), \Upsilon')}{\Gamma; \mathscr{L}; L; \Upsilon \vdash \& \ \ell \ \mathsf{mut} \ x@p : \& \ \ell \ \mathsf{mut} \ \tau; \Upsilon'}$$

Conjecture (Expression Shadow Preservation)**.** *If* $\vdash \mathscr{L}$*,* $\mathscr{L} \vdash \Upsilon : \Gamma \leq L$*, and* $\Gamma; \mathscr{L}; L; \Upsilon \vdash e : \tau; \Upsilon'$ *then* $\mathscr{L} \vdash \Upsilon' : \Gamma \leq L$*.*

Justification. This is trivial in the integer case and follows from L-Value Use Shadow Preservation in the L-value use case, so we only need to discuss the borrow creation cases. In both borrow creation cases, the only change to the shadow heap is the addition of one new loan.

When creating an immutable borrow, we know from CAN-READ that no mutable loan exists on way to the path we are borrowing or in memory reachable from it. The shadow of the interior of the loaned path is now controlled by the immutable loan of the path, but since there were no mutable loans there before and it has not changed it should still be well-formed. Similarly, the shadows on the way to the path have not changed directly and they should not be affected by the addition of an immutable loan in their interior. The main concern is whether the addition of the new immutable loan to the bank will result in a well-formed bank. We know from the validity check that the lifetime of the loan is within the lifetime bound of the bank. If the bank is empty or the lifetime of the old loan is not shorter than the lifetime of the new loan, then the bank will still be well-formed. We could restrict RECORD to only succeed in this situation.

When creating a mutable borrow, we know from CAN-WRITE that no loan exists on the way to the path or in memory reachable from it. So similar to the immutable case, the shadow of the interior of the path will still be well-formed even though it is now controlled by a mutable loan. The shadows on the way to the path are still well-formed since they have no loans to begin with. As before, the only question is whether the bank of the path will remain well-formed. We know from CAN-WRITE that the bank is empty, so the new bank, consisting of just one mutable loan, will be well-formed. □

Using a Layout

The runtime analogue of using an L-value is using a layout. Affine values are replaced by void and copyable values are left untouched. It is not defined for void, which prevents expression evalutation from utilizing uninitialized memory. A program that tries to do so will get stuck.

$\boxed{\textsc{use} : \text{Layout} \to \text{Layout}}$

$$\begin{array}{lcl} \textsc{use}(z) & = & z \\ \textsc{use}(\sim \alpha) & = & \mathsf{void} \\ \textsc{use}(\& \ \mathsf{imm} \ \alpha) & = & \& \ \mathsf{imm} \ \alpha \\ \textsc{use}(\& \ \mathsf{mut} \ \alpha) & = & \mathsf{void} \end{array}$$

Note that $\textsc{use}(l) = l'$ implies $\textsc{droppable}(l')$. Once we use a value, we know we can safely deallocate it.

Expression Evaluation

Evaluating expressions is straightforward. The only unusual aspect is the case of using the value at an L-value. To prevent copies of affine values, we must overwrite the original value with the used version. If it is a copyable value, then it will remain unchanged. If it is an affine value, then it will be replaced with void. Note that this means the heap resulting from expression evaluation is either unchanged or a more deinitialized version of the starting heap. Also note that the result heap may temporarily contain a leak

when considered alone, but not when considered together with the result layout.

$\boxed{V; H \vdash e \to l; H'}$

$$\frac{}{V; H \vdash z \to z; H}\ \text{EE-INT} \qquad \frac{V; H \vdash x@p \to \alpha}{V; H \vdash \&\ \ell\ q\ x@p \to \&\ q\ \alpha; H}\ \text{EE-REF}$$

$$\frac{V; H \vdash x@p \to \alpha \qquad \text{READ}(H, \alpha, l) \qquad \text{WRITE}(H, \alpha, \text{USE}(l), H')}{V; H \vdash x@p \to l; H'}\ \text{EE-USE}$$

Expression Preservation

Having defined both evaluation and typing for expressions, we can finally state the preservation lemma for expressions. Assuming all input contexts are well-formed and given a well typed expression that evaluates to a layout, then that layout has the same type as the expression, the resulting shadow heap is well-formed, and the resulting heap is consistent with the resulting shadow heap.

Conjecture (Expression Preservation)**.** *If* $\vdash H : \Sigma$*,* $\Gamma; V \vdash \Sigma$*,* $\vdash \mathscr{L}$*,* $\mathscr{L} \vdash \Upsilon : \Gamma \leq L$*,* $V \vdash H : \Upsilon$*,* $\Gamma; \mathscr{L}; L; \Upsilon \vdash e : \tau; \Upsilon'$*, and* $V; H \vdash e \to l; H'$ *then* $\Sigma \vdash l : \tau$*,* $\vdash H' : \Sigma$*,* $\mathscr{L} \vdash \Upsilon' : \Gamma \leq L$*, and* $V \vdash H' : \Upsilon'$*.*

Justification. This is trivial in the integer case. In the reference case, the heap does not change. From Expression Shadow Preservation, we get the well-formedness of the new shadow heap, which has been extended with one new loan. Since the intialization information in the shadow heap has not changed, it should still model the heap correctly. That the type of the layout matches the type of the expression is simple to establish.

In the use case, we again get the well-formedness of the new shadow heap from Expression Shadow Preservation. The type of the layout we can get via Path Type Preservation and Read Safety. The write we perform on the heap does not change the type of anything, so the new heap will still typecheck under the heap type. In the new shadow heap, the used path was deinitialized (if affine). In the new heap, the used path was voided (if affine). The new heap should be consistent with the new shadow heap. □

The Layout Resulting from Evaluation is Fully Initialized

We should also note that the resulting layout is fully initialized. This is trivial in the integer case. In the reference and the use case we rely on typing judgments that asserted full initialization and the correspondence of the shadow heap with the runtime heap.

Expression Progress

The progress lemma for expressions is surprisingly straightforward. Since we do not nest expressions, no induction is necessary. Each case constructs the resulting values in a straightforward manner.

Conjecture (Expression Progress)**.** *If* $\vdash H : \Sigma$*,* $\Gamma; V \vdash \Sigma$*,* $\vdash \mathscr{L}$*,* $\mathscr{L} \vdash \Upsilon : \Gamma \leq L$*,* $V \vdash H : \Upsilon$*, and* $\Gamma; \mathscr{L}; L; \Upsilon \vdash e : \tau; \Upsilon'$ *then there exists a layout* l *and a heap* H' *such that* $V; H \vdash e \to l; H'$*.*

Justification. In the integer case this is trivial. In both reference cases, we can rely on Path Progress to evaluate the L-value and use the address to construct the resulting layout, and the heap does not change. In the use case, we can rely on Path Progress to evaluate the L-value to an address. Then we can rely on Read Safety to find the layout. Since we could read sufficiently, we can also write sufficiently so we find the heap. □

Statements

We have finally reached the outermost shell of Patina! The statements layer is primarily about heap manipulation and chaining expressions. It mostly just relies on judgments we have already defined. skip and $s; s$ are simply for chaining together statements. $x@p \leftarrow e$ is for initializing a path with the result of an expression. $x@p \leftarrow \mathsf{new}\ x@p$ is similar, but allocates the result on the heap and initializes the path with a new unique pointer. Treating this as a statement rather than an expression simplifies the expression preservation and progress lemmas. Note that both versions are for *initialization* and not *assignment*. We only allow initialization of droppable paths, which we guarantee will not leak memory. $x@p \leftrightarrow x@p$ will perform a shallow swap of two writable paths. This is the only way to actually mutate through a mutable borrow since we do not allow the interior of a borrowed reference to be uninitialized even temporarily. $\mathsf{free}\ x@p$ performs a shallow free and is the inverse of new. $\mathsf{push}\ \ell\ x\ \tau\ s$ will create a new stack variable. $\mathsf{pop}\ x\ s$ will pop that stack variable once the statement that requires it finishes. It is for evaluation purposes and should not appear in user code.

$$\text{Statements} \quad s \quad ::= \quad \mathsf{skip} \mid s; s \mid x@p \leftarrow e \mid x@p \leftarrow \mathsf{new}\ x@p \mid x@p \leftrightarrow x@p \mid \mathsf{free}\ x@p \mid \mathsf{push}\ \ell\ x\ \tau\ s \mid \mathsf{pop}\ x\ s$$

Droppable Shadows

Earlier we defined $\textsc{droppable}(l)$, which specified layouts that were safe to deallocate. Now we define an equivalent notion for shadows, which we will use to ensure only droppable layouts are deallocated. We also require a droppable shadow to be unborrowed, which will prevent us from creating dangling borrowed references.

$\boxed{\sigma\ \textsc{droppable}}$

$$\text{DS-UNINIT}\ \frac{}{\emptyset : \mathsf{uninit}\ \textsc{droppable}} \qquad \text{DS-INT}\ \frac{}{\emptyset : \mathsf{int}\ \textsc{droppable}} \qquad \text{DS-REF}\ \frac{\textsc{contains-no-loans}(\sigma)}{\emptyset : \&\ q\ \sigma\ \textsc{droppable}}$$

Subtyping

The only subtyping in Patina is lifetime subtyping. Borrowed references are contravariant in their lifetimes. This means that a reference for a longer lifetime is a subtype of a reference for a shorter lifetime, which is the behavior we want. The reference of a longer lifetime is usable anywhere the reference of a shorter lifetime is. Since mutable references can both read and write, they are invariant in their type argument. Immutable references can only read, so they are covariant in their type argument.

$\boxed{\mathscr{L} \vdash \tau_0 <: \tau_1}$

$$\text{SUB-INT}\ \frac{}{\mathscr{L} \vdash \mathsf{int} <: \mathsf{int}} \qquad \text{SUB-OWN}\ \frac{\mathscr{L} \vdash \tau_1 <: \tau_2}{\mathscr{L} \vdash \sim \tau_1 <: \sim \tau_2} \qquad \text{SUB-REFIMM}\ \frac{(\ell_2, \ell_1) \in \mathscr{L} \qquad \mathscr{L} \vdash \tau_1 <: \tau_2}{\mathscr{L} \vdash \&\ \ell_1\ \mathsf{imm}\ \tau_1 <: \&\ \ell_2\ \mathsf{imm}\ \tau_2}$$

$$\text{SUB-REFMUT}\ \frac{(\ell_2, \ell_1) \in \mathscr{L}}{\mathscr{L} \vdash \&\ \ell_1\ \mathsf{mut}\ \tau <: \&\ \ell_2\ \mathsf{mut}\ \tau}$$

Statement Typing

$\boxed{\Gamma; \mathscr{L}; L; \Upsilon \vdash s : \Upsilon'}$

The skip statement does nothing so it is easy to check.

$$\text{ST-SKIP}\ \frac{}{\Gamma; \mathscr{L}; L; \Upsilon \vdash \mathsf{skip} : \Upsilon}$$

Sequencing two statement simply sequences typechecking by threadings the shadow heap. Note that since Γ is not changing, the statements can only change the initialization and loan information of the shadow heap.

$$\text{ST-SEQ}\ \frac{\Gamma; \mathscr{L}; L; \Upsilon_0 \vdash s_1 : \Upsilon_1 \qquad \Gamma; \mathscr{L}; L; \Upsilon_1 \vdash s_2 : \Upsilon_2}{\Gamma; \mathscr{L}; L; \Upsilon_0 \vdash s_1; s_2 : \Upsilon_2}$$

Swapping requires only that the two L-values have the same type and that they both be writable. Since the writability guarantees no part is loaned, the shadow heap is not changed.

$$\text{ST-SWAP}\ \frac{\begin{array}{ll}\Gamma \vdash x_1@p_1 : \tau & \textsc{can-write}(p_1, \Upsilon(x_1)) \\ \Gamma \vdash x_2@p_2 : \tau & \textsc{can-write}(p_2, \Upsilon(x_2))\end{array}}{\Gamma; \mathscr{L}; L; \Upsilon \vdash x_1@p_1 \leftrightarrow x_2@p_2 : \Upsilon}$$

Initializing with a value requires that the path be droppable so that nothing will be leaked. It also requires the expression type to be a subtype of the path type. As discussed above, this refers to lifetimes: the subtype lifetime is greater than or equal to the lifetime of the supertype. This requirement guarantees that the value stored at the path will be valid for at least as long as the path is valid. Finally, we initialize the path in the output shadow heap.

$$\text{ST-SET}\ \frac{\begin{array}{lll}\Gamma \vdash x@p : \tau_p & \Gamma; \mathscr{L}; L; \Upsilon_0 \vdash e : \tau_e; \Upsilon_1 & \mathscr{L} \vdash \tau_e <: \tau_p \\ \textsc{shallow}(\Upsilon_1, x@p, \sigma) & \sigma\ \textsc{droppable} & \textsc{init}(\Upsilon_1, x@p, \Upsilon_2)\end{array}}{\Gamma; \mathscr{L}; L; \Upsilon_0 \vdash x@p \leftarrow e : \Upsilon_2}$$

Initializing with a pointer is similar to initializing with a value. Except instead of expression typing, we must check that we can use the path on the right hand side.

$$\text{ST-NEW}\ \frac{\begin{array}{lll}\Gamma \vdash x_l@p_l : \sim \tau_l & \Gamma; \Upsilon_0 \vdash x_r@p_r : \sim \tau_r \Rightarrow \Upsilon_1 & \mathscr{L} \vdash \sim \tau_r <: \sim \tau_l \\ \textsc{shallow}(\Upsilon_1, x_l@p_l, \sigma) & \sigma\ \textsc{droppable} & \textsc{init}(\Upsilon_1, x_l@p_l, \Upsilon_2)\end{array}}{\Gamma; \mathscr{L}; L; \Upsilon_0 \vdash x_l@p_l \leftarrow \mathsf{new}\ x_r@p_r : \Upsilon_2}$$

Freeing the memory of a unique pointer requires checking that that memory is droppable. We then mark it used in the output shadow heap, which since this a unique pointer will replace the shadow with uninit.

$$\text{ST-FREE}\ \frac{\Gamma \vdash x@p : \sim \tau \qquad \textsc{shallow}(\Upsilon, x@p, \sim \sigma) \qquad \sigma\ \textsc{droppable} \qquad \textsc{use}(\Upsilon, x@p, \Upsilon')}{\Gamma; \mathscr{L}; L; \Upsilon \vdash \mathsf{free}\ x@p : \Upsilon'}$$

Pushing a new stack variable involves checking that the body is well typed under the appropriately extended contexts. Newly pushed variables start uninitialized. By the end of the body, the variable must be droppable so that popping it will not leak memory. When checking the body, we extend the lifetime relation with the new lifetime by adding the new lifetime as the smallest lifetime in the relation (it is related to everything else). Afterwards, we must filter everything involving this lifetime from the shadow heap. This should only consist of removing loans from banks. We also "deallocate" the stack variable from the shadow heap by restricting the domain to the domain of the input.

$$\text{ST-PUSH}\quad \frac{\begin{array}{c}\textsc{new-smallest-lifetime}(\mathscr{L}, \ell, \mathscr{L}') \\ \Gamma, x \mapsto \tau; \mathscr{L}'; L, x \mapsto \ell; \Upsilon_0, x \mapsto \mathsf{uninit} \vdash s : \Upsilon_1 \\ \Upsilon_1(x)\ \textsc{droppable} \qquad \textsc{filter}(\Upsilon_1, \ell, \Upsilon_2)\end{array}}{\Gamma; \mathscr{L}; L; \Upsilon_0 \vdash \mathsf{push}\ \ell\ x\ \tau\ s : \Upsilon_2|_{\operatorname{dom}\Upsilon_0}}$$

Popping a stack variable is similar, but we do not need to worry about extending contexts. Instead, we must make sure to remove the stack variable when we restrict the domain of the output shadow heap.

$$\text{ST-POP}\quad \frac{\begin{array}{c}\Gamma; \mathscr{L}; L; \Upsilon_0 \vdash s : \Upsilon_1 \\ \Upsilon_1(x)\ \textsc{droppable} \qquad \textsc{filter}(\Upsilon_1, \ell, \Upsilon_2)\end{array}}{\Gamma; \mathscr{L}; L; \Upsilon_0 \vdash \mathsf{pop}\ x\ s : \Upsilon_2|_{\operatorname{dom}\Upsilon_0 - \{x\}}}$$

One interesting thing to note here is that the domain of the input and output shadow heaps is always the same except for the pop case, where it decreases by one. Since pop does not appear in user code, this means that if a program starts with no memory allocated and that program type checks, then the program frees all the memory it allocates. Assuming of course that evaluation preserves the shadow heap.

Statement Evaluation

Statement evaluation goes as you would expect. We use a new helper judgment $\textsc{fresh}(\alpha)$ that generates a unused allocation. As before, we utilize domain restriction to "deallocate". Other than that, we simply use path and expression evaluation combined with READ and WRITE.

$$\boxed{V; H \vdash s \to V'; H' \vdash s'}$$

$$\text{SE-SKIPSEQ}\ \frac{}{V; H \vdash \mathsf{skip}; s \to V; H \vdash s} \qquad \text{SE-STEPSEQ}\ \frac{V; H \vdash s_1 \to V'; H' \vdash s_1'}{V; H \vdash s_1; s_2 \to V'; H' \vdash s_1'; s_2}$$

$$\text{SE-SET}\ \frac{V; H_0 \vdash e \to l; H_1 \qquad V; H_1 \vdash x@p \to \alpha \qquad \textsc{write}(H_1, \alpha, l, H_2)}{V; H_0 \vdash x@p \leftarrow e \to V; H_2 \vdash \mathsf{skip}}$$

$$\text{SE-NEW}\ \frac{\begin{array}{c} V; H_0 \vdash x_r@p_r \to \alpha_r \qquad \textsc{read}(H_0, \alpha_r, l_r) \qquad \textsc{write}(H_0, \alpha_r, \textsc{use}(l_r), H_1) \\ V; H_1 \vdash x_l@p_l \to \alpha_l \qquad \textsc{fresh}(\alpha_n) \qquad \textsc{write}(H_1, \alpha_l, \sim \alpha_n, H_2) \end{array}}{V; H_0 \vdash x_l@p_l \leftarrow \mathsf{new}\ x_r@p_r \to V; H_2, \alpha_n \mapsto l_r \vdash \mathsf{skip}}$$

$$\text{SE-SWAP}\ \frac{\begin{array}{cc} V; H_0 \vdash x_l@p_l \to \alpha_l & V; H_0 \vdash x_r@p_r \to \alpha_r \\ \textsc{read}(H_0, \alpha_l, l_l) & \textsc{read}(H_0, \alpha_r, l_r) \\ \textsc{write}(H_0, \alpha_l, l_r, H_1) & \textsc{write}(H_1, \alpha_r, l_l, H_2) \end{array}}{V; H_0 \vdash x_l@p_l \leftrightarrow x_r@p_r \to V; H_2 \vdash \mathsf{skip}}$$

$$\text{SE-FREE}\ \frac{V; H_0 \vdash x@p \to \alpha \qquad \textsc{read}(H_0, \alpha, \sim \alpha') \qquad \textsc{write}(H_0, \alpha, \textsc{use}(\sim \alpha'), H_1)}{V; H_0 \vdash \mathsf{free}\ x@p \to V; H_1|_{\mathrm{dom}\, H_1 - \{\alpha'\}} \vdash \mathsf{skip}}$$

$$\text{SE-PUSH}\ \frac{\textsc{fresh}(\alpha)}{V; H \vdash \mathsf{push}\ \ell\ x\ \tau\ s \to V, x \mapsto \alpha; H, \alpha \mapsto \mathsf{void} \vdash \mathsf{pop}\ x\ s} \qquad \text{SE-POPSTEP}\ \frac{V; H \vdash s \to V'; H' \vdash s'}{V; H \vdash \mathsf{pop}\ x\ s \to V'; H' \vdash \mathsf{pop}\ x\ s'}$$

$$\text{SE-POPSKIP}\ \frac{}{V; H \vdash \mathsf{pop}\ x\ \mathsf{skip} \to V|_{\mathrm{dom}\, V - \{x\}}; H|_{\mathrm{dom}\, H - \{V(x)\}} \vdash \mathsf{skip}}$$

Statement Preservation

Statement preservation is the assertion that given all our well-formed contexts, a well-typed statement, and the fact that that statement evaluates to another statement, then we can also type the result statement and both statements produce the same output shadow heap, i.e. they effectively end in the same way. Put another way, this lemma asserts that memory safe programs step to memory safe programs.

It is difficult to specify how the new versions of contexts relate to the old versions. They may be larger or smaller depending upon the statement in question. We do not specify this relationship currently. Instead we just rely on the new variable map, heap, and statement to enforce the relationship.

Conjecture (Statement Preservation)**.** *If* $\vdash H : \Sigma$, $\Gamma; V \vdash \Sigma$, $\vdash \mathscr{L}$, $\mathscr{L} \vdash \Upsilon : \Gamma \leq L$, $V \vdash H : \Upsilon$, $\Gamma; \mathscr{L}; L; \Upsilon \vdash s : \Upsilon_0$, *and* $V; H \vdash s \to V'; H' \vdash s'$ *then there exist* Σ', Γ', $\mathscr{L}'$, L', *and* Υ' *such that* $\vdash H' : \Sigma'$, $\Gamma'; V' \vdash \Sigma'$, $\vdash \mathscr{L}'$, $\mathscr{L}' \vdash \Upsilon' : \Gamma' \leq L'$, $V' \vdash H' : \Upsilon'$, *and* $\Gamma'; \mathscr{L}'; L'; \Upsilon' \vdash s' : \Upsilon_0$.

Justification. The skip-seq case is trivial. The step-seq case follows from induction. In the value initialization (set) case, only the shadow heap and the heap are changing. From Expression Preservation, we know that the heap and shadow heap are still aligned after evaluating the expression. The WRITE will initialize the path, which is reflected in the shadow heap. The pointer initialization (new) case is similar. Since the components of a swap must have the same shadow, the modified heap will still be well-formed. We required that the memory deallocated by free was droppable, which should mean that the heap is still well-formed once we void the pointer. Push steps immediately to pop and only extends contexts, which should still be

well-formed. The pop-step case follows from induction. In the pop-skip case, similar to the free case, we know that the variable is droppable, which should mean the resulting contexts are well-formed. □

Statement Progress

Statement progress is essentially the assertion that a well-typed statement will never attempt to read uninitialized memory, free borrowed memory, and so on. Essentially, that well-typed programs have sufficient memory safety to take a step. The restriction to statements that end with an empty shadow heap allows us to avoid the cases with leaked memory.

Conjecture (Statement Progress)**.** *If* $\vdash H : \Sigma$, $\Gamma; V \vdash \Sigma$, $\vdash \mathscr{L}$, $\mathscr{L} \vdash \Upsilon : \Gamma \leq L$, $V \vdash H : \Upsilon$, $\Gamma; \mathscr{L}; L; \Upsilon \vdash s : \emptyset$ *then there exists* V', H', *and* s' *such that* $V; H \vdash s \to V'; H' \vdash s'$ *or* $s = \mathsf{skip}$.

Justification. The skip case is immediate. The sequence case is either immediate or follows from induction. The swap case and the new case follow from path progress and read safety. The set case is similar, but also requires expression progress. The push case is immediate. The pop case is either immediate or follows from induction. □

Patina Soundness

If our conjectures hold, then we have established soundness for Patina. This means that a well-typed statement that ends with an empty shadow heap will not leak memory, not access uninitialized memory, not create aliased, mutable memory, and so on. In effect, well-typed programs are statically guaranteed to be memory safe.

Future Work

The first extension to the existing work should be to prove our conjectures. This is necessary to put us on a firm footing for other future work, and to ensure we have actually included everything necessary in our model.

The expression and statement layers should be extended to match the path layer. Product types are straightforward to add to the borrow checker and will likely not pose much trouble. Sum types may be more difficult. In particular, matching by reference allows creation of references to the payload of a sum type. The borrow checker must ensure that the sum is not mutated while such a reference exists. Work on adding existential types to imperative languages can likely be adapted for this purpose [7]. Extending the model with recursive types will require the addition of top level (non-closure) recursive functions. Without the capability those functions give us it is impossible to free recursive values. Those functions will necessitate other additions, e.g. lifetime parameters. Type parameters likely do not have significant soundness implications by themselves, but may in their interaction with later features.

Until recently, Rust's closures have been in flux. As of this writing, Rust supports both stack and heap closures and also supports different manners of access to the environment (owning, immutable reference, or mutable reference). In the past, closures have been a common source of borrow checker bugs, which likely means that proving soundness will be difficult. However, it also means that it is one of the most essential features to add to Patina. It would likely be profitable to look to Cyclone [8], which took great care to ensure references could not escape via closures or existential types.

Traits, which are Rust's version of typeclasses or interfaces, may have similar issues. Traits can be used to create what is effectively an object or an existential type. Since objects and closures are so similar, it is likely that proving trait objects sound will face similar difficulties. Rust's closures are actually implemented as syntactic sugar for certain trait implementations in the current version. It is unclear whether this means formalization of closures will be easier or whether formalization of traits will be harder.

Recently, Rust has added some more advanced type system features. In particular, associated types and higher rank lifetime polymorphism are of interest. Associated types are effectively functions from types to types. Neither of these features will likely impact soundness, but they should still be carefully modeled.

There are many potential features that we could explore using our model. Several additional versions of borrowed references has been purposed, e.g. write-only references or references that allow moving out as long as something is moved back in before the reference is released. Adding this to Patina would likely result in a more general borrow checking framework.

One oft-requested feature is the ability to use static values in type parameters. This effectively gives Rust a weak form of dependent typing. The consequences for soundness in a affine dependently typed language with regions are likely significant, and the difficulty in proving them even more so.

Our model made several changes to how systems worked in the Rust compiler. It would be interesting to compare how the model's version compares with the original in terms of performance, ease of understanding, and mantainability.

Related Work

Safe dialects of C, unique pointers, borrowed references, regions, and other topics relevant to Rust each have long and full histories. Rather than attempt to catalogue them in detail and compare Rust with each of its ancestors, we will limit our discussion to the most closely related work we are aware of.

Both ML Kit [12, 13] and Cyclone [8] heavily influenced the design of Rust. However, both of those languages, along with most research in the area, uses regions for memory management. In constrast, Rust's memory is managed by affine types (unique ownership), and uses regions (lifetimes in Rust) more for ensuring the safety of borrowed references than for managing memory. Cyclone's region subtyping is the source of Rust's lifetime subtyping and so they ensure references do not escape in a similar fashion. Like Cyclone, Rust prevents access to uninitialized memory, but Rust goes further than Cyclone by statically preventing dereferences of null pointers. Unlike Cyclone, Rust does not have first class regions, but Rust also lacks Cyclone's dynamic regions, which lessens the need for first class regions.

Cyclone later added unique pointers and a notion of borrowing [11, 9]. Cyclone's version does not prevent memory leaks, which are possible both from freeing a unique pointer that contains another live unique pointer and from overwriting a live unique pointer. Rust statically guarantees no memory leaks in both of these situations. Cyclone requires an explicit swap operation for using unique versions of pointers that do not admit null values. Rust will allow any of its pointers to be null, but guarantees that they will not be used while null. Rust also does not require explicit swaps; Patina uses them, but that is an artifact of the model. Cyclone considers unique pointers "consumed" while a borrow exists, whereas Rust will allow the borrowed unique pointer to be used in certain situations (namely, reading through an immutably borrowed unique pointer is allowed). Cyclone supports abstraction over the aliasability and kind (box versus region) of a pointer, but Rust does not currently support any such abstraction over pointer types.

Overall, Rust and Cyclone are each more expressive in different ways. Cyclone's dynamic regions make it much easier to specify the lifetime of a particular heap data structure, and they allow for heap structures that are difficult or impossible to express in Rust, such as a doubly linked list. Rust supports far more detailed reasoning about when memory should be freed due to its flow-sensitivity, and Rust offers safer and more convenient usage of linear data structures, particularly with regard to freeing them. Cyclone still has much Rust can learn from, e.g. a mechanism similar to dynamic regions could be used to support custom allocators in Rust.

Walker and Watkins discuss how regions and linear types could be combined [14]. Their model supports first class regions, like Cyclone, but unlike Rust. It also introduces the idea of linear regions and provides a mechanism for temporarily aliasing them. This could be a fruitful ground for Rust to explore when looking to support custom allocators. However, their model does not support mutation, which is the source of most of the complexity of Rust's borrow checker, and this may impact its applicability to Rust.

Gordon et al. [6] introduce a static method for ensuring data-race freedom by enforcing that at most one mutable reference to an object exists, which is a similar task to Rust's effort to avoid aliased, mutable memory.

Their work supports read-only references, which cannot be used to mutate, but do not forbid mutation. Since these are by definition aliased, mutable memory, Rust has no equivalent. However, their immutable reference is similar to Rust's immutable borrowed references, and their isolated referece is somewhat similar to Rust's unique pointers and mutable borrowed references. Isolated references can be coverted to writeable references and back, which allows for behavior similar to a mutable borrowed reference. Isolated references can also be converted into immutable references, which is analogous to immutably reborrowing a mutably borrowed reference. Their correspondence with unique pointers is similar. Gordon, Ernst, and Grossman [5] later propose a general notion of rely-guarantee references, which treat aliases similar to different threads of control and use methods from rely-guarantee program logics to ensure safety. Rust's references can likely be encoded in this general framework.

Boyland's alias burying [2] has a similar goal to Rust, i.e. using static checks rather than destructive reads, but differs in many details. Boyland defines a strong version of the uniqueness invariant (true uniqueness) and a weak version (allowing borrowed aliases), and then develops a system based on maintaing the strong invariant. Rust meets only the weak uniqueness invariant, but it also guarantees that aliased, mutable memory never exists, which allows for similar control over mutation as the strong uniqueness invariant. The general idea of alias burying is to allow for aliases, but to invalidate them when a unique field is read. Rust ensures at a unique value will not be read if (mutable) aliases exist. Rust's borrowed references are also more first-class than Boyland's, which cannot be stored in data structures or returned from functions.

Clarke and Wrigstad's external uniqueness [4] has a notion of ownership similar to Rust's. Both have recursive ownership; however, Rust uses it for memory management rather than for access control. Unlike Rust, unique pointers are nullified on use. The Patina model nullifies on use, but soundness ensures it does not matter in practice. There is also a fresh owner notion for borrowed references that accomplishes the same thing as Rust's lifetimes, i.e. preventing escape. Unlike Rust, Clarke and Wrigstad's borrowed references are both copyable and mutable, which means they cannot make the same safety guarantees as Rust, i.e. no access to uninitialized memory. They list several options about how borrowed unique pointers can be handled, but Rust does not cleanly fit into these categories. Rust discriminates on the manner of borrowing and the operation to be performed to determine whether it is safe to use the borrowed unique pointer, but Clarke and Wrigstad suggest that a system must either allow all uses or forbid all uses. Their model also has problems with immutable (final) variables and unique versions of non-nullable pointers because they denote consumption via nullification and the finality prevents that. Rust, however, has both immutable variables and all of its pointers are non-null, but it has no issues mixing these with uniqueness. This is because Rust's ban on null pointers is not a ban on pointers being null, but rather on pointers being used while null. Additionally, Rust's immutable variables' usability can change if a unique value is consumed, which allows Rust to render an immutable variable unusable thereafter. Clarke and Wrigstad speculate that alias burying would enable them to support this combination, which is somewhat true. Rust does manage this via static analysis, but not via alias burying.

Boyland, Noble, and Retert's capabilities for sharing [3] provides a general framework for describing uniqueness and read-only concepts related to pointers. However, Rust's pointers do not quite fit into their capability system. The issue involves immutably loaned unique pointers and immutably loaned mutable borrowed references. In Rust, both allow reading through the pointer, but do not allow reading the pointer itself, which would require a move. The capabilities for sharing system supports only whole object permissions, which prevents us from specifying this case of Rust's pointers in the system.

Potanin et al. [10] survey the various ways of adding immutability to object-oriented languages and how it impacts aliasing, including Javari, OIGJ, and Joe_3. These systems usually add immutable objects and/or read-only references, but they do not provide the no aliased, mutable memory guarantee of Rust. As discussed earlier, read-only references are antithetical to Rust's memory safety invariant. Immutable objects are similar to Rust's immutably borrowed references, but the OO systems often go further by allowing immutability at the field level. Rust does not allow programmers to specify the mutability of particular fields, but immutable borrows can provide a temporary guarantee that is similar. Several of the OO systems have a concept of ownership, but it is primarily about encapsulation and access control. Rust handles encapsulation via its module system rather than via the type system or static analysis. As usual with OO

languages, they rely on garbage collection for memory management, which is very unlike Rust.

Acknowledgements

Thanks to Dan Grossman, Nicholas Matsakis, and Zach Tatlock for their advice on this work and on this paper. Thanks to the Mozilla Corporation for financially supporting this work.

References

[1] The Rust Programming Language. `http://www.rust-lang.org`.

[2] John Boyland. Alias burying: Unique variables without destructive reads. *Software: Practice and Experience*, 31(6):533–553, 2001.

[3] John Boyland, James Noble, and William Retert. Capabilities for sharing. In JørgenLindskov Knudsen, editor, *ECOOP 2001 – Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 2–27. Springer Berlin Heidelberg, 2001.

[4] Dave Clarke and Tobias Wrigstad. External uniqueness is unique enough. In Luca Cardelli, editor, *ECOOP 2003 Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, pages 176–200. Springer Berlin Heidelberg, 2003.

[5] Colin S. Gordon, Michael D. Ernst, and Dan Grossman. Rely-guarantee references for refinement types over aliased mutable data. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 73–84, New York, NY, USA, 2013. ACM.

[6] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. Uniqueness and reference immutability for safe parallelism. *SIGPLAN Not.*, 47(10):21–40, October 2012.

[7] Dan Grossman. Existential types for imperative languages. In Daniel Le Mtayer, editor, *Programming Languages and Systems*, volume 2305 of *Lecture Notes in Computer Science*, pages 21–35. Springer Berlin Heidelberg, 2002.

[8] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in cyclone. *SIGPLAN Not.*, 37(5):282–293, May 2002.

[9] Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. Experience with safe manual memory-management in cyclone. In *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04, pages 73–84, New York, NY, USA, 2004. ACM.

[10] Alex Potanin, Johan Östlund, Yoav Zibin, and Michael D. Ernst. Immutability. In *Aliasing in Object-Oriented Programming*, volume 7850 of *LNCS*, pages 233–269. Springer-Verlag, April 2013.

[11] Nikhil Swamy, Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. Safe manual memory management in cyclone. *Science of Computer Programming*, 62(2):122 – 144, 2006. Special Issue: Five perspectives on modern memory management - Systems, hardware and theory.

[12] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, and Peter Sestoft. Programming wth regions in the MLKit (revised for version 4.3.0). Technical report, IT University of Copenhagen, Denmark, January 2006.

[13] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109 – 176, 1997.

[14] David Walker and Kevin Watkins. On regions and linear types (extended abstract). In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, ICFP '01, pages 181–192, New York, NY, USA, 2001. ACM.