Short Paper: Rusty Types for Solid Safety

Sergio Benitez
Stanford University
353 Serra Mall
Stanford, CA 94305
sbenitez@stanford.edu

ABSTRACT

Programs operating "close to the metal" necessarily handle memory directly. Because of this, they must be written in languages like C or C++. These languages lack any kind of guarantee on memory or race safety, often leading to security vulnerabilities and unreliable software. Ideally, we would like a practical language that gives programmers direct control over memory and aliasing while also offering race and memory safety guarantees.

We present Rusty Types and an accompanying type system, inspired by the Rust language, that enable memory-safe and race-free references through ownership and restricted aliasing in the type system. In this paper, we formally describe a small subset of the syntax, semantics, and type system of Metal, our Rust-based language that enjoys Rusty Types. Our type system models references and ownership as capabilities, where bindings have indirect capabilities on value locations. We also present speculative extensions to Rusty Types that allow greater flexibility in single threaded programs while maintaining the same guarantees.

1. INTRODUCTION

Aliasing via references is a polarizing subject. In practice, aliasing often leads to memory and concurrency errors, which invariably lead to security and reliability issues. On the contrary, references are easy to understand in principal, inexpensive, and essential to achieving performance in software. As such, languages targeting low-level software typically allow unrestricted aliasing without memory or safety guarantees. Ideally, we would like a practical language with precise control of aliasing and memory that also exhibits race and memory safety.

A significant amount of work, including that on linear types [32], ownership types [14], unique types [23], and alias types [29] has brought us closer to this goal. These building blocks can be used to enable memory-safe and race-free references by restricting object encapsulation and tracking aliases [6, 15]. Of particular interest is the notion of "borrowing" references: creating pointers to otherwise unique and mutable references, tracking those pointers, and restoring uniqueness and mutability when the pointers are deemed unusable [8]. Paired with permissions [26] or capabilities [15], such a system offers the guarantees we desire.

Unfortunately, existing systems based on these ideas either target languages where memory cannot be managed by the programmer or are not flexible enough to be used in practice. For instance, Naden et al.'s recent work on a "type system for borrowing permissions" [26] does not allow borrows from unique references to be stored in non-local storage such as the heap or in other non-local references. Cyclone, while allowing full control of memory, imposes the same restriction [20]. These limitations are foreign to most programmers and may help explain why these languages have not seen adoption in practice.

The objective of this paper is to resolve these limitations. We introduce Rusty Types, and an accompanying type system, in the context of Metal, our Rust-based [28] language. Rusty Types give a programmer guarantees about the aliasing of an object given only its type. For example, if a "mutable borrow" is allowed, the programmer is guaranteed to have a unique reference to the underlying object. Similarly, if an "immutable borrow" is allowed, the programmer is guaranteed that no "mutable borrows" are active.

Rusty Types are *intuitive*, as evidenced by the growing popularity of Rust. Unlike other type systems that provide similar guarantees, Rusty Types are *implicitly* managed by the type system and require minimal syntax. To enable this, our type system (described in §3) models ownership and borrows as *capabilities* where variables have *indirect* capabilities on abstract value locations. Our formalization is reminiscent of the calculus of capabilities [15], which we expand on, together with other related work, in §6.

2. WHAT MAKES TYPES *RUSTY?*

Take a large satchel. Toss in linear types [32], ownership types [14], unique types [23], alias types [29], borrowing [26], permissions [9], and capabilities [15]. Add flexibility, practicality, and mix thoroughly for 25 years. If all goes well, you will find Rusty Types in your satchel.

2.1 Linearity

Except for immutable references, all Rusty Types behave linearly. This means that exactly one binding to a given object is allowed at any point in the program. In other words, variables may not actively alias the same object. When a binding attempts to alias an object using an existing vari-

PLAS '16, October 24, 2016, Vienna, Austria.

ACM ISBN 978-1-4503-4574-3/16/10.

DOI: http://dx.doi.org/10.1145/2993600.2993604

able, the existing variable becomes unusable. The object is *moved* to the new variable, and the new variable *owns* the object. For example, in the following Rust-like program,

```
1 let x = Vector([1, 2, 3]);
2 let y = x;
```

the `Vector` initially owned by `x` is moved to `y` in line 2. Any appearance of `x` after the move is a type error. This behavior applies to fields of structures, and fields of fields of structures, and so on, leading to *partially moved* objects.

2.2 Memory Safety and Race Freedom

Rusty Types are designed for programs that require or would benefit from unmanaged, direct references to memory. Unlike most existing languages with such references, programs written with Rusty Types are statically guaranteed to be memory-safe and race-free.

Rusty Types maintain several key invariants that lead to memory and race safety. To start, every storage location is guaranteed to have either: **(a)** 1 mutable reference and 0 immutable references to it, *or* **(b)** 0 mutable references and n immutable references to it. This invariant directly prevents races as it prohibits concurrent writers and readers to a single memory location. By itself, however, this invariant is not enough to guarantee memory safety, especially in the presence of moveable objects. For instance, since a given variable becomes unusable after its object has been moved, the storage locations associated with that variable may be reused or freed. As a result, any references to that variable will be dangling and invalid after a move.

To prevent this, Rusty Types also ensure that each object has a unique binding, the *owner*, and that references to an object or its contents are created transitively through its owner. The type system guarantees that an object's owner does not change (the result of a move) while references are outstanding. Conversely, the type system allows change of ownership when there are no outstanding references.

Together, these invariants ensure that no references are left dangling or pointing to invalid memory. As an example, consider the following Rust-like program:

```
1 let mut x = Vector([1, 2, 3]);
2 let y = &x[0];
3 clear(&mut x);
4 let z = *y;
```

In the program above, `Vector` is a container type, itself a value, that holds a pointer to a heap allocated array, initialized here with 1, 2, and 3. The variable `x` is *bound* mutably to this vector. The `mut` annotation on `x` allows both the variable `x` to be rebound and the vector to be mutated. `y` holds an immutable reference (so it can't modify the underlying object) to the first element of the vector. The `clear` function takes in a mutable reference to the vector in `x` and deallocates the internal array of the passed-in vector, removing all of the elements. The last line in the program dereferences `y` and stores the value in `z`.

Is this program memory safe? No! Since `y` holds a reference to the vector's first element, and the array where the element resides will be deallocated by the call to `clear`, `y` will hold a dangling pointer after `clear` returns, making the dereference on line 4 undefined. Rusty Types statically disallow this program. This is because `y`'s immutable reference to `x[0]` on line 2 prohibits the mutable reference to `x` on line 3, even though the references would point to distinct memory locations, since they descend from the same owner.

2.3 (Re)borrowing

As mentioned earlier, *borrowing* is the act of creating references (known as *borrows*) from, or creating a pointer to, objects or their fields. *Re*borrowing, then, is doing the same but through an existing borrow. Borrowing or reborrowing from a mutable object must render that object immutable or unusable to maintain the invariants described earlier. The original object may be restored its mutability or usability if all of its borrows can be shown to be unusable.

The syntax "&" is used to take an immutable borrow, where the underlying object may not be mutated, while "`&mut`" is used to take a mutable borrow, where mutation of the underlying object is allowed. To illustrate, consider the following program:

```
1 let mut x = V(..); // x: write
2 let a = &mut x;    // x: []; *a: write, a: read
3 let b = &(*a);     // *a: read; {*b, b}: read
```

The comments to the right demonstrate the changes to variables' capabilities on each line. We describe these now. The variable `x` is declared mutable, so it begins with write capabilities on its contents. When `x` is borrowed mutably on line 2, `x` must become unusable to maintain the "1 mutable reference" invariant. As a result, it loses all of its capabilities; `a` gains the capability to read its contents (the borrow) as well as write capabilities *through* its borrow (via `*a`) to the underlying object `x`. On line 3, `x` is *reborrowed* immutably *through* `a`. This results in `a` losing its ability to write `x` though `*a`, leaving both `*a` and `*b` with read access to `x`. Note that `x` does *not* regain its ability to read its contents because the mutable borrow in `a` remains outstanding.

If all borrows to a mutable object are inaccessible, then it is safe to allow that object to be mutated again. With Rusty Types, capabilities can be restored at function boundaries. Consider the following type signature for `clear` from the example in §2.2:

```
1 fn clear(vector: &mut Vector);
```

This function can be called as `clear(&mut x)`. After a call to `clear`, the mutable borrow of `x` is said to be *returned*. As such, `x` may be borrowed mutably once more, as in the following program:

```
1 let mut x = Vector([1, 2, 3]);
2 clear(&mut x);
3 let y = &mut x;
```

Rusty Types determine whether a borrow outlives a function call based solely on the function's signature. The intuition is that a borrow is not returned from a function call if the function's signature indicates that the borrow may outlive, or be stored as a result of, the function call. We provide formal treatment to this analysis, as well as a complete example, in §3.2.2, once we have developed a formal understanding of Rusty Types.

3. FORMALIZING RUSTY TYPES

How do Rusty Types precisely track capabilities through borrows and reborrows to maintain the invariants we desire? In this section, we present a subset of Metal, our Rust-based language, which enjoys Rusty Types. Metal's type system formulates ownership and (re)borrows as *capabilities*, where variables have *indirect* capabilities on value locations. Our flow-sensitive `move` type judgement (§3.2.1) captures capability change, while our `location-capability` relation (also §3.2.1) captures whether a move is allowed.

$$
\begin{array}{rrcl}
\text{location} & \ell & ::= & l \mid \ell.n \\
\text{type} & \tau & ::= & \text{unit} \mid \text{int} \mid Q_\mu(\overline{\tau}) \mid \&\tau \\
& & \mid & \&\text{mut } \tau \mid (\overline{b{:}\tau}){:}\,\tau \mid \tau@l^\dagger \\
\text{value} & v & ::= & i \mid \{\overline{v}\}_{Q_\mu} \mid \text{brw}(\ell)^\dagger \\
\text{path} & p & ::= & x \mid p.n \\
\text{module path} & \mu & ::= & \text{m} \mid \text{m}{::}\mu \\
\text{expression} & e & ::= & v \mid p \mid \&p \mid \&\text{mut } p \mid *e \mid \{\overline{e}\}_{Q_\mu} \\
& & \mid & e.n \mid f_\mu(\overline{e}) \mid \textbf{call } \{s\}^\dagger \\
\text{binding} & b & ::= & x \mid \text{mut } x \\
\text{statement} & s & ::= & e \mid s_1; s_2 \mid \textbf{let } b = e \mid e_1 = e_2 \\
& & \mid & \textbf{return } e \\
\text{visibility} & z & ::= & \textbf{pub} \mid \textbf{priv} \\
\text{declaration} & d & ::= & z \textbf{ struct } Q(\overline{z\ \tau}) \mid z \textbf{ fn } f(\overline{b{:}\tau}){:}\,\tau\,\{s\} \\
\text{program} & \pi & ::= & \overline{d}; s
\end{array}
$$

Figure 1: Core syntax for Metal.

3.1 Syntax

We begin with a core subset of Metal's syntax, shown in Figure 1. Metal's syntax mirrors Rust's where possible. We use an overline to indicate vectors, such as in $\overline{\tau}$, and vertical bars to indicate the length of a given vector, such as in $|\overline{\tau}|$. Some constructors are marked with † to denote that they are not part of the surface syntax, such as $\tau@l^\dagger$.

Types. Category τ defines the syntax for types. It includes unit (unit), integers (int), structs ($Q_\mu(\overline{\tau})$), immutable ($\&\tau$) and mutable ($\&\text{mut } \tau$) borrows, and function types ($(\overline{b{:}\tau}){:}\,\tau$), where $\overline{b{:}\tau}$ are the arguments' types and τ is the return type. Importantly, types can be decorated with locations, such as in $\tau@l^\dagger$; we call this a *located type*.

Values. Category v defines the syntax for values. It includes numbers (i) and struct instances ($\{\overline{v}\}_{Q_\mu}$). It also includes the operational representation of borrows ($\text{brw}(\ell)^\dagger$).

Expressions. Category e defines the syntax for expressions. It includes values (v), paths (p), and the creation ($\&p$, $\&\text{mut } p$) and dereference ($*e$) of borrows. Though we do not discuss them in in this paper, modules are represented by the subscript μ, such as in the function call $f_\mu(\overline{e})$. Expressions also include the operational representation of a function call scope ($\textbf{call } \{s\}^\dagger$).

Statements. Category s includes immutable and mutable bindings ($\textbf{let } b = e$), assignment ($e_1 = e_2$), function return ($\textbf{return } e$), and sequential composition ($s_1; s_2$).

Declarations and Program. The syntactic category for declarations (d) includes structures and functions which must be declared either public (**pub**) or private (**priv**). We ignore visibility qualifications in this paper. A program (category π) is defined as declarations followed by a statement.

3.2 Static Semantics

The type environment is composed primarily of:

$$\Gamma : x \mapsto \tau \qquad \mathcal{A} : x \mapsto \overline{l} \qquad \mathcal{C} : l \mapsto \overline{\kappa}$$

As usual, Γ maps variables to types. $\mathcal{A}$ maps a variable x to a set of locations $\overline{l}$, and $\mathcal{C}$ relates some location l to a set of capabilities $\overline{\kappa}$, where $\kappa := \texttt{move} \mid \texttt{read} \mid \texttt{write}$.

Locations. A mapping $(x \mapsto \overline{l}) \in \mathcal{A}$ indicates that x has *some* capabilities on the set of locations $\overline{l}$, but not *which*. Locations are introduced through **let** statements exclusively. For instance, in $\textbf{let } x = \{\{42\}_{R_\mu}, \{7\}_{R_\mu}\}_{Q_\mu}$, the type of x in Γ after type-checking is

$$Q_\mu(R_\mu(\text{int@l}_4)@\text{l}_2, \text{R}_\mu(\text{int@l}_5)@\text{l}_3)@\text{l}_1,$$

and $\mathcal{A}$ contains $x \mapsto [l_1, l_2, l_3, l_4, l_5]$, which shows that x has capabilities on all of its components. Note that all types in Γ are *fully located*, or composed of only located types.

Capabilities. The exact capabilities x has on a given location l are revealed by $\mathcal{C}(l)$. A capability represents an action that can be taken on a given location. `move` means that a location is moveable; `read` and `write` mean that a location can be read or modified, respectively. In the previous example, for instance, $\forall l \in [l_1, l_2, l_3, l_4, l_5].\ \mathcal{C}(l) = [\texttt{move}, \texttt{read}]$. If x had instead been declared mut x, $\mathcal{C}(l)$ would additionally contain the `write` capability for all l.

Left-Hand Side. When assigning to a path ($p = e$), it is necessary to assert that the left-hand side *path variable*, the variable in the resolved path (e.g., x in $p = x.0.1$), has `write` capabilities on the locations to be modified by the assignment. This implies that we must discover and track the path variable. As such, our left-hand side expression typing rules include a parameter ($\leftarrow x$) that indicates the provenance of a given path.

More precisely, left-hand side expression type judgements have the form $\Gamma, \Psi, \nu \vdash_{\text{lhs}} e : \tau \leftarrow x$ where τ is a fully located type. Ψ maps struct, function, and method identifiers to their declared type, while ν is the module path for the expression currently being type-checked.

Figure 2 gives selected rules for typing left-hand side expressions. (**L1**) is the base case and indicates that the provenance of variable is the variable itself. (**L2**) shows that a struct's fields descend from the struct itself.

Right-Hand Side. The right-hand side expression typing judgement is defined for all types; we use a simple subtyping relation to strip locations from types. The typing rules are fairly standard. To illustrate, Figure 3 gives the typing rule for function application. As expected, the rule checks that argument and return types agree with the function's type.

$$\textbf{(LT1)}\quad \frac{\Gamma(x) = \tau}{\Gamma, \Psi, \nu \vdash_{\text{lhs}} x : \tau \leftarrow x}$$

$$\textbf{(LT2)}\quad \frac{\Gamma, \Psi, \nu \vdash_{\text{lhs}} e : Q_\mu(\overline{\tau})@l \leftarrow x}{\Gamma, \Psi, \nu \vdash_{\text{lhs}} e.i : \overline{\tau}_{i+1} \leftarrow x}$$

Figure 2: Left-hand side typing rules.

$$\textbf{(R1)}\quad \frac{\Psi(f_\mu) = (\overline{b{:}\tau}){:}\,\tau \qquad (\Gamma, \Psi, \nu \vdash_{\text{rhs}} \overline{e}_1 : \overline{\tau}_1\ \ \cdots\ \ \Gamma, \Psi, \nu \vdash_{\text{rhs}} \overline{e}_n : \overline{\tau}_n)}{\Gamma, \Psi, \nu \vdash_{\text{rhs}} f_\mu(\overline{e}) : \tau}$$

Figure 3: Right-hand side typing rule.

3.2.1 Move Semantics

Variables and locations gain and lose capabilities through the flow-sensitive `move` type judgment of the form

$$\Gamma, \Psi, \nu, \mathcal{A}, \mathcal{C} \vdash_{\text{move}} e \Rightarrow \mathcal{A}', \mathcal{C}'.$$

The judgement describes the change in capabilities $(\mathcal{A}', \mathcal{C}')$ resulting from an expression e being used as the right-hand side of an assignment or binding. This judgement maintains the following invariant:

$$\forall l.\ \texttt{write} \in \mathcal{C}(l) \implies \exists x. \forall y.\ l \in \mathcal{A}(y) \implies x = y.$$

In words, if a variable holds a write capability to some location, it is the only variable with capabilities to that location.

$$\textbf{(CR1)}\quad \frac{\bar{l} \subseteq \mathcal{A}(x) \qquad \forall l \in \bar{l}\,.\,\kappa \in \mathcal{C}(l)}{\mathcal{A}, \mathcal{C} \vdash_{\text{can}} \kappa(\bar{l}) \leftarrow x}$$

Figure 4: Location-capability relation.

$$\text{owned}(\text{unit@l}) = \{\text{l}\} \qquad \text{owned}(\text{int@l}) = \{\text{l}\}$$
$$\text{owned}(\text{Q}_\mu(\overline{\tau})\text{@l}) = (\cup_{\tau \in \overline{\tau}} \text{owned}(\tau)) \cup \{\text{l}\}$$
$$\text{owned}(\&\tau\text{@l}) = \{\text{l}\} \qquad \text{owned}(\&\text{mut } \tau\text{@l}) = \{\text{l}\}$$

Figure 5: Definition for owned(τ@l).

This invariant is key to enabling the memory and race safety guarantees of Rusty Types in Metal.

Checking Capabilities. The `move` type judgment depends on the `location-capability` relation of the form

$$\mathcal{A}, \mathcal{C} \vdash_{\text{can}} \kappa(\bar{l}) \leftarrow x,$$

read "capability κ for locations $\bar{l}$ are held by variable x." Figure 4 gives the derivation for the judgement.

The set of locations to be checked varies given the circumstance. To obtain the appropriate set of locations for a given type, the type system uses functions like the one in Figure 5. There are three such functions: `all`, `owned`, and `mut`, which are defined for fully located types. The all(·) function extracts all locations reachable from type τ, including those accessible via borrows. The owned(·) function extracts the set of location originally allocated for type τ, that is, all locations excluding those accessible via borrows. The mut(·) function extracts the set of locations originally mutable by τ via mutable borrows.

Moving. Figure 6 gives selected `move` rules. Note the variable x from the provenance chain of the left-hand side typing rule. (**MR1**) ensures that immutable borrows to a path can only be created when all of the path's locations are readable ($\mathcal{A}, \mathcal{C} \vdash_{\text{can}} \texttt{read}(\text{all}(\tau)) \leftarrow x$). In particular, this disallows borrows to moved locations that would otherwise result in dangling pointers. If the immutable borrow is allowed, `move` and `write` capabilities are revoked for all locations readable in τ. This ensures that aliases in $\mathcal{A}$ are only allowed `read` access to the affected locations and that the locations cannot be moved while references exist. We write the revocation as

$$\mathcal{C} \oplus \{l \mapsto \mathcal{C}(l) - [\texttt{move}, \texttt{write}]\}_{l \in \text{all}(\tau)},$$

where $\oplus$ and $-$ denote mapping update and list subtraction operators, respectively. The subscript denotes the elements to apply the operation over.

(**MR2**) ensures that mutable borrows to a path can only be created when the path can write to all of it's owned (owned(τ)) and mutably borrowed (mut(τ)) locations. If the mutable borrow is allowed, the previous capability holder loses all capabilities over its owned and mutable locations. We write this as $\mathcal{A} \oplus \{x \mapsto \mathcal{A}(x) - \bar{l}\}$. This restriction is necessary to maintain the "1 mutable reference" invariant. As in rule (**MR1**), x loses ownership over the borrowed structure.

(**MR3**) ensures that a path can be moved if it has all of the capabilities its type implies. In particular, this means that the path must be able to `move` its owned locations ($\mathcal{A}, \mathcal{C} \vdash_{\text{can}} \texttt{move}(\text{owned}(\tau)) \leftarrow x$), ensuring nothing is borrowed, `read` its locations ($\mathcal{A}, \mathcal{C} \vdash_{\text{can}} \texttt{read}(\text{all}(\tau)) \leftarrow x$), ensuring all of its locations are still accessible, and `write` all of its mutable locations ($\mathcal{A}, \mathcal{C} \vdash_{\text{can}} \texttt{write}(\text{mut}(\tau)) \leftarrow x$). As a result of the move, all capabilities held by variable x via τ are revoked ($\mathcal{A} \oplus \{x \mapsto \mathcal{A}(x) - \text{all}(\tau)\}$). Consequently, path p can no longer be used, enforcing the linear nature of paths.

Statements. Figure 7 gives the most interesting statement typing rules involving move semantics: assignment and binding. (**ST1**) ensures that the left-hand and right-hand side bare types, types without location annotations, are equivalent ($(\Gamma, \Psi, \nu \vdash_{\text{rhs}} e_i : \tau)_{i \in \{l,r\}}$). It also ensures that x, the path provenance variable, has `write` capabilities to all locations that will be affected by the assignment ($\mathcal{A}, \mathcal{C} \vdash_{\text{can}} \texttt{write}(\text{owned}(\tau_l)) \leftarrow x$). Observe that e_l's provenance variable does not gain new capabilities, while e_r's provenance variable may *lose* capabilities.

(**ST2**) types immutable bindings. It deduces the bare type τ of e. It also allocates fresh locations for new values. The locate(·) function in Figure 8 captures this idea; observe that it introduces *fresh locations* for undecorated types.

3.2.2 Function Analysis

Metal determines the set of borrows OL that may outlive a function call through a simple procedure on the function's type. The procedure begins by collecting two type sets: W, the set of all writeable borrow types that appear in the function signature, including those in the return type, and B, the set of all borrow types that can be borrowed or reborrowed given the function signature. OL can then be defined as:

$$OL \coloneqq \bigcup_{w \in W} \{b \mid b \in B.\ *b \overset{\text{lhs}}{\neq} *w \wedge b \overset{\text{rhs}}{=} w\}$$

In words, OL is the union of all borrow types in B that that are not exactly equal (*lhs*, including locations) to a writeable borrow type, but are equal (*rhs*, without locations) to some writeable borrow type. The first inequality prevents a borrow type from being included in OL simply because it is writeable, while the second determines if a borrow can be stored in a given borrow location. To see how this analysis works, consider the following function type[1]:

```
1  (a: &mut Q(&R(int)), b: &R(int)): unit
```

Given this function signature,

$$W = \{\texttt{\&R(int)}\} \qquad B \supset \{\texttt{\&R(int)}, \texttt{\&*\&R(int)}\}$$
$$OL = \{\texttt{\&R(int)}, \texttt{\&*\&R(int)}\}$$

Observe that B and OL include the potential reborrow of `b`. Since `b` $\in OL$, the borrow `b` will not be returned after a call to a function with this signature.

This analysis is used in the flow-sensitive typing judgement for function calls, which we do not show here. In short, types in OL are `moved` by the judgement; the remaining types are unperturbed and maintain their capabilities.

3.3 Soundness

To discuss soundness in brief, we provide a glimpse into Metal's dynamic semantics. Figure 9 gives the small-step operational semantics of the form (F, S, e) for selected left-hand and right-hand side expressions. F carries function type information necessary for function calls. S is a global stack of references and values. It is inductively defined as:

$$S \coloneqq ML :: S \mid \textbf{halt} \qquad M : x \mapsto l \qquad L : l \mapsto v$$

(**L1**) retrieves a variable's location from M, (**L2**) extracts the underlying location of a borrow for a dereference, and (**R1**) retrieves a path's value from L. These three rules provide the nearly complete path for $*p \xrightarrow{rhs}{}^{*} v$.

[1]We elide locations here for simplicity.

$$\textbf{(MR1)}\quad \frac{\Gamma,\Psi,\nu \vdash_{\text{lhs}} p : \tau \leftarrow x \qquad \mathcal{A},\mathcal{C} \vdash_{\text{can}} \texttt{read}(\text{all}(\tau)) \leftarrow x}{\Gamma,\Psi,\nu,\mathcal{A},\mathcal{C} \vdash_{\text{move}} \&p \Rightarrow \mathcal{A},\mathcal{C} \oplus \{l \mapsto \mathcal{C}(l) - [\texttt{move},\texttt{write}]\}_{l \in \text{all}(\tau)}}$$

$$\textbf{(MR2)}\quad \frac{\Gamma,\Psi,\nu \vdash_{\text{lhs}} p : \tau \leftarrow x \qquad \bar{l} = \text{owned}(\tau) \cup \text{mut}(\tau) \qquad \mathcal{A},\mathcal{C} \vdash_{\text{can}} \texttt{write}(\bar{l}) \leftarrow x}{\Gamma,\Psi,\nu,\mathcal{A},\mathcal{C} \vdash_{\text{move}} \&\text{mut } p \Rightarrow \mathcal{A} \oplus \{x \mapsto \mathcal{A}(x) - \bar{l}\},\mathcal{C} \oplus \{l \mapsto \mathcal{C}(l) - [\texttt{move}]\}_{l \in \bar{l}}}$$

$$\textbf{(MR3)}\quad \frac{\Gamma,\Psi,\nu \vdash_{\text{lhs}} p : \tau \leftarrow x \qquad \mathcal{A},\mathcal{C} \vdash_{\text{can}} \texttt{move}(\text{owned}(\tau)) \leftarrow x \qquad \mathcal{A},\mathcal{C} \vdash_{\text{can}} \texttt{read}(\text{all}(\tau)) \leftarrow x \qquad \mathcal{A},\mathcal{C} \vdash_{\text{can}} \texttt{write}(\text{mut}(\tau)) \leftarrow x}{\Gamma,\Psi,\nu,\mathcal{A},\mathcal{C} \vdash_{\text{move}} p \Rightarrow \mathcal{A} \oplus \{x \mapsto \mathcal{A}(x) - \text{all}(\tau)\},\mathcal{C}}$$

Figure 6: Move relations.

$$\textbf{(ST1)}\quad \frac{\{\Gamma,\Psi,\nu \vdash_{\text{rhs}} e_i : \tau\}_{i \in \{l,r\}} \qquad \Gamma,\Psi,\nu \vdash_{\text{lhs}} e_l : \tau_l \leftarrow x \qquad \mathcal{A},\mathcal{C} \vdash_{\text{can}} \texttt{write}(\text{owned}(\tau_l)) \leftarrow x \qquad \Gamma,\Psi,\nu,\mathcal{A},\mathcal{C} \vdash_{\text{move}} e_r \Rightarrow \mathcal{A}',\mathcal{C}'}{\Gamma,\Psi,\nu,\mathcal{A},\mathcal{C} \vdash_{\text{stmt}} e_l = e_r \Rightarrow \Gamma,\mathcal{A}',\mathcal{C}'}$$

$$\textbf{(ST2)}\quad \frac{\Gamma,\Psi,\nu \vdash_{\text{rhs}} e : \tau \qquad \Gamma,\Psi,\nu,\mathcal{A},\mathcal{C} \vdash_{\text{move}} e \Rightarrow \mathcal{A}',\mathcal{C}' \qquad \tau_d = \text{locate}(\tau)}{\Gamma,\Psi,\nu,\mathcal{A},\mathcal{C} \vdash_{\text{stmt}} \textbf{let } x = e \Rightarrow \Gamma[x \mapsto \tau_d],\mathcal{A}'[x \mapsto \text{all}(\tau_d)],\mathcal{C}'[l \mapsto [\texttt{move},\texttt{read}]]_{l \in \text{owned}(\tau_d)}}$$

Figure 7: Typing rules for statements.

$$\text{locate}(\text{unit}) = \text{unit@l}' \qquad \text{locate}(\text{int}) = \text{int@l}'$$

$$\text{locate}(\text{Q}_\mu(\overline{\tau})) = \text{Q}_\mu(\text{map}(\text{locate},\overline{\tau}))\text{@l}'$$

$$\text{locate}(\&\tau) = (\&\text{locate}(\tau))\text{@l}' \qquad (\text{l}' \text{ fresh})$$

$$\text{locate}(\&\text{mut } \tau) = (\&\text{mut locate}(\tau))\text{@l}' \qquad \text{locate}(\tau\text{@l}) = \tau\text{@l}$$

Figure 8: Definition for locate(τ.)

$$\textbf{(L1)}\ (F, ML :: S, x) \xrightarrow{lhs} (F, ML :: S, M(x))$$

$$\textbf{(L2)}\ (F, S, *\text{brw}(\ell)) \xrightarrow{lhs} (F, S, \ell)$$

$$\textbf{(R1)}\ (F, ML :: S, l.\overline{n}) \xrightarrow{rhs} (F, ML :: S, L(l).\overline{n})$$

Figure 9: Selected Metal operational semantics.

We prove soundness in the typical manner: by proving progress and preservation. More interestingly, to prove memory safety, we show an expanded version of Theorem 1.

Theorem 1 (Memory Safety)**.**
If $\Gamma,\Psi,\nu,\mathcal{A},\mathcal{C} \vdash_{move} *p \Rightarrow \mathcal{A}',\mathcal{C}'$*, then* $(F,S,*p) \xrightarrow{rhs}{}^{*} (F,S',v)$.

This theorem follows from the general progress theorem. Due to space limitations and the evolving nature of Metal, we do not include the full theorems or proofs here. However, we note the isomorphisms between $\mathcal{A}$, $\mathcal{C}$ in the type system and M, L in the operational semantics, respectively.

4. COMPARING METAL AND RUST

Syntactically, Metal is a strict subset of Rust. Semantically, however, Metal can track locations, and thus capabilities, with greater precision. As a result, Metal accepts a wider range of safe programs in its syntactic subset. On the other hand, Metal lacks Rust's region-based [30] "lifetimes". As a result, it is unable to support Rust's `Drop` trait and has coarser grained capability restoration.

4.1 Regions and Destructors

Rust supports object destructors [5] via the `Drop` trait. This works as expected: when an object falls out of scope, a `drop` method is called on the object, allowing the object to clean-up its resources. Objects are *dropped* in LIFO order: if `b` is declared after `a`, `b`'s drop will run before `a`'s. This implies that in a memory-safe program, `a` must not hold references to `b`, or any other object declared after it, since those (dangling) references can be accessed during `a`'s drop.

As presented, Metal cannot enforce this behavior and thus cannot soundly support `Drop`. Rust enforces this behavior using region-based "lifetimes". In Rust, every borrow type is annotated with a region in which the borrow is considered valid. Rust uses the annotation to ensure that a memory location only holds borrows that outlive it. To illustrate this, consider the following program, which is memory-unsafe in the presence of `Drop` but memory-safe otherwise:

```
1 fn my_func() {
2   let p1 = Person(..);        // p1: '2
3   let mut a = Parent(&p1);    //  a: '3; 2 < 3?
4   let p2 = Person(..);        // p2: '4
5   a = Parent(&p2);            //       ; 4 < 3?
6 }
```

We use line numbers to denote the region beginning at that line (so larger regions have smaller numbers), and trace regions and region checks in comments. The comment `x: 'r` says that variable `x` inhabits region `r`, while $r_1 < r_2$? says that the statement at that line requires r_1 to outlive r_2. From this, it is easily seen that the statement on line 5 fails its region check. As expected, type-checking in Rust fails.

Rust's region tracking is in actuality coarser than what we've implied. We have experimented with region-based lifetimes in Metal but have not extended our soundness proofs to cover them, so we elide their formal discussion. Our region analysis in the comments above is based on this experimentation. We found that adding regions to the type-system does not introduce any complexity, accounting for only a single syntactic change and an additional subtyping rule. This addition also improves our function capability restoration analysis; we discuss this in the next section.

4.2 Regions and Capability Restoration

The function analysis for capability restoration presented in §3.2.2 can be significantly improved with region-based lifetimes. To illustrate, we return to the simple example from §3.2.2, where our analysis determined that `b` *may* outlive the function call. However, whether `b` *actually* outlives the function call depends on the function body. We might analyze the function body, but doing so would sacrifice modular type-checking. Regions provide a better solution.

With regions, each borrow can be annotated with a label for the region where the contents it points to live. To tell the type-checker that two borrows must be from different, incompatible regions, the programmer can annotate

each borrow with a different label. The type-checker will then ensure that a memory location only holds references with labels for regions compatible with its own. The function type from §3.2.2 can thus be rewritten as:

```
(a: &mut Q(&'a R(int)), b: &'b R(int)): unit
```

Since `*a.0` and `b` have incompatible regions (`'a` and `'b`, respectively), `b` cannot be stored in `*a.0`, and the analysis will not mark `b` as outliving the function call. Thus, the borrow in `b` will be returned after the call.

4.3 Increased Flexibility in Metal

Metal's precise tracking of locations leads it to accept a wider range of safe programs than Rust in its syntactic subset. Consider the following program, where `Q` is assumed to be defined as a struct with type $(Q(\text{int}))$. This program type-checks in Metal but fails to type-check in Rust:

```
1 let mut a = Q(1);    // a: write
2 let mut b = Q(1);    // b: write
3 let mut x = &mut a; // a: []; x, *x: write
4 let y = &mut *x;     // *x: []; *y: write, y: read
5 x = &mut b;          // b: []; *x: write
```

We again use comments, like those in §2.3, to illustrate how Metal's flow-sensitive `move` relations modify capabilities given the statement on each line. Rust rejects the program because it believes the statement on line 5 attempts to modify an object `x` that was borrowed on line 4. Metal, on the other hand, precisely tracks locations and determines the judgement $\mathcal{A}, \mathcal{C} \vdash_{\text{can}} \texttt{write}(\text{owned}(\tau)) \leftarrow x \mid \Gamma(x) = \tau$ holds on line 5, thus allowing the program.

5. REMOVING LIMITATIONS

It is impossible to write certain safe programs and data structures in Metal or Rust, particularly those that involve cycles or multiple mutable references to a single object[2]. This comes as a direct consequence of the aliasing restrictions imposed by the type system. While these restrictions are generally crucial to enforcing memory safety, they are unnecessary in certain contexts.

Specifically, when a set of references are used in a single thread, aliasing restrictions can be relaxed in two ways.

Single Depth. First, multiple mutable and/or immutable references to an object are safe iff those references all refer to a *single depth* of a given object. We define depth as $|\overline{.n}|$ for a reference $\&x\overline{.n}$ to object x. The intuition is that if all references point to a single depth of a given object, then any allowed deallocations or changes in object structure cannot affect existing references. As a result, existing references cannot be invalidated, so the program remains memory-safe.

Any Depth. Second, multiple mutable and/or immutable references to *any* depth of an object are safe iff **1)** the object being pointed to does not contain heap allocations, and **2)** neither the object nor its fields are of sum type. The intuition here is similiar: since the object's structure cannot change under existing references, no existing references can be made invalid, and the program remains memory-safe.

To the best of our knowledge, we are the first to make these observations. We are exploring implementing these ideas in Metal via the addition of an `&local` reference type and an accompanying `local` capability.

[2]Rust has an `unsafe` escape hatch that allows these programs by sacrificing memory and race safety guarantees.

6. RELATED WORK

Ownership. The lineage of type systems based on the idea of object encapsulation is long [3, 11–14, 16, 21, 31] and both inspired and borrow from separation logic [27].

Early ownership based type systems modeled full object encapsulation by restricting objects to single owners [13, 14, 31] with no support for taking references into objects. Later work allowed *temporary* references into objects but either disallowed their storage entirely [3, 12, 21] or allowed storage up to a certain containment point [7]. Some systems allowed multiple owners [1, 11] but disallowed ownership transfer, while others allowed ownership transfer but disallowed multiple owners [2, 6, 16, 25]

Viewed from the lens of ownership types, Rusty Types can be said to allow multiple owners, allow references into objects, allow transfer of ownership, and allow storage of references without restriction (excepting memory safety, of course). While our system is not based on ownership types, the encapsulation and abstraction provided by linear types is reminiscent of ownership types and yields similar benefits.

Alias Types. Alias Types [29, 33] are similarly motivated by low-level alias control. With alias types, programmers can specify constraints on aliases as *stores*. Using stores to control read/write effects requires explicit, sometimes lengthy annotations; Rusty Types make this mostly implicit with minimal syntax. Alias types do not allow "reborrowing" nor pointers to the middle of memory blocks.

Unlike Rusty Types, alias types allow recursive, mutable data structures. As discussed in §5, we are exploring mechanisms to enable recursive, mutable data structures in Metal.

Capabilities. The construction of our type system bears similarity to the seminal work on calculus of capabilities [15]. Of important difference is that objects in our type system have capabilities *indirectly*. This difference is key to enabling borrowing and reborrowing, driven by capability revocation and transfer, an impossibility with the construction in [15].

Fractional Permissions. In fractional permissions, statements are allowed only when the correct set of permissions exists [9, 10]. To track read and write effects, permissions are split: reads are allowed with only a fraction of a permission while writes require "the whole thing". As such, borrowing is a special case. The generality and mathematical abstraction of this approach introduces significant complexities to static analysis and can be unnatural for programmers [26]. As a result, fractional permissions can only be found hidden behind simpler abstractions [4, 19, 22]. In contrast, Rusty Types can be viewed as making permissions implicit, simplifying semantics and lowering the mathematical barrier.

Other Languages. Vault [17, 18], Cyclone [20], and L^3 [24] all bear similarities to Metal. Vault uses linear keys to track objects. Keys can be thought of as non-fractional permissions and suffer from similiar drawbacks. Cyclone features uniqueness types with a relatively limited form of immutable borrows which are not allowed to exit the scope in which they are created. Moreover, moving object fields in Cyclone requires the use of an explicit *swap* operation. L^3's *frozen* locations are similar to Metal's `&` borrows, while *thawed* locations are similar to Metal's `&mut` borrows. However, there is a fundamental and important difference between thawed locations and `&mut` borrows: thawed locations may be aliased by many frozen locations, as long as they have the same type. As discussed in §2, this is memory-unsafe in a lan-

guage where programmers have a direct view of memory. As a result, L^3 and Rusty Types are strictly incompatible.

7. CONCLUSION

We presented a formalization of Metal, our Rust-based language that enjoys Rusty Types. Rusty Types enable memory-safe and race-free references through ownership and restricted aliasing in the type system. We also presented novel and speculative extensions to Rusty Types that increase their flexibility in single threaded contexts.

8. ACKNOWLEDGMENTS

A particularly special thank you to Alejandro Russo for his careful advice and contributions to the type system. Thanks to Riad S. Wahby, David Mazières, and Edward Z. Yang for their comments and feedback. Thanks also to the anonymous reviewers for their feedback and suggestions.

References

[1] J. Aldrich and C. Chambers. *Ownership Domains: Separating Aliasing Policy from Mechanism*, pages 1–25. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[2] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, pages 311–330, New York, NY, USA, 2002. ACM.

[3] P. S. Almeida. *Balloon types: Controlling sharing of state in data types*, pages 32–59. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.

[4] K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 301–320, New York, NY, USA, 2007. ACM.

[5] H.-J. Boehm. Destructors, finalizers, and synchronization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pages 262–272, New York, NY, USA, 2003. ACM.

[6] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, pages 211–230, New York, NY, USA, 2002. ACM.

[7] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. *SIGPLAN Not.*, 38(1):213–223, Jan. 2003.

[8] J. Boyland. Alias burying: Unique variables without destructive reads. *Softw. Pract. Exper.*, 31(6):533–553, May 2001.

[9] J. Boyland. Checking interference with fractional permissions. In *Proceedings of the 10th International Conference on Static Analysis*, SAS'03, pages 55–72, Berlin, Heidelberg, 2003. Springer-Verlag.

[10] J. T. Boyland and W. Retert. Connecting effects and uniqueness with adoption. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 283–295, New York, NY, USA, 2005. ACM.

[11] N. R. Cameron, S. Drossopoulou, J. Noble, and M. J. Smith. Multiple ownership. *SIGPLAN Not.*, 42(10):441–460, Oct. 2007.

[12] D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. *SIGPLAN Not.*, 37(11):292–310, Nov. 2002.

[13] D. G. Clarke, J. Noble, and J. Potter. Simple ownership types for object containment. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 53–76, London, UK, UK, 2001. Springer-Verlag.

[14] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '98, pages 48–64, New York, NY, USA, 1998. ACM.

[15] K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 262–275, New York, NY, USA, 1999. ACM.

[16] D. Cunningham, W. Dietl, S. Drossopoulou, A. Francalanza, P. Müller, and A. J. Summers. Formal methods for components and objects. chapter Universe Types for Topology and Encapsulation, pages 72–112. Springer-Verlag, Berlin, Heidelberg, 2008.

[17] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 59–69, New York, NY, USA, 2001. ACM.

[18] M. Fahndrich and R. DeLine. Adoption and focus: Practical linear types for imperative programming. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 13–24, New York, NY, USA, 2002. ACM.

[19] S. Heule, K. R. M. Leino, P. Müller, and A. J. Summers. Fractional permissions without the fractions. In *Proceedings of the 13th Workshop on Formal Techniques for Java-Like Programs*, FTfJP '11, pages 1:1–1:6, New York, NY, USA, 2011. ACM.

[20] M. Hicks, G. Morrisett, D. Grossman, and T. 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.

[21] J. Hogg. Islands: Aliasing protection in object-oriented languages. In *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '91, pages 271–285, New York, NY, USA, 1991. ACM.

[22] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. Verifast: A powerful, sound, predictable, fast verifier for c and java. In *Proceedings of the Third International Conference on NASA Formal Methods*, NFM'11, pages 41–55, Berlin, Heidelberg, 2011. Springer-Verlag.

[23] N. H. Minsky. Towards alias-free pointers. In *Proceedings of the 10th European Conference on Object-Oriented Programming*, ECCOP '96, pages 189–209, London, UK, UK, 1996. Springer-Verlag.

[24] G. Morrisett, A. Ahmed, and M. Fluet. L3: A linear language with locations. In *Proceedings of the 7th International Conference on Typed Lambda Calculi and Applications*, TLCA'05, pages 293–307, Berlin, Heidelberg, 2005. Springer-Verlag.

[25] P. Müller and A. Rudich. Ownership transfer in universe types. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 461–478, New York, NY, USA, 2007. ACM.

[26] K. Naden, R. Bocchino, J. Aldrich, and K. Bierhoff. A type system for borrowing permissions. *SIGPLAN Not.*, 47(1):557–570, Jan. 2012.

[27] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

[28] Rust Project Developers. The Rust Programming Language, 2016.

[29] F. Smith, D. Walker, and G. Morrisett. *Alias Types*, pages 366–381. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.

[30] M. Tofte and J.-P. Talpin. Region-based memory management. *Inf. Comput.*, 132(2):109–176, Feb. 1997.

[31] J. Vitek and B. Bokowski. Confined types. *SIGPLAN Not.*, 34(10):82–96, Oct. 1999.

[32] P. Wadler. Linear types can change the world! In *IFIP TC 2 Working Conference on Programming Concepts and Methods*, pages 347–359. North Holland, 1990.

[33] D. Walker and G. Morrisett. *Alias Types for Recursive Data Structures*, pages 177–206. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.