Excerpts From: 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. WHAT MAKES TYPES *RUSTY?*

Take a large satchel. Toss in linear types [7], ownership types [2], unique types [4], alias types [6], borrowing [5], permissions [1], and capabilities [3]. Add flexibility, practicality, and mix thoroughly for 25 years. If all goes well, you will find Rusty Types in your satchel.

1.1 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 variable, 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,

PLAS '16, October 24, 2016, Vienna, Austria.
© 2018 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-4574-3/16/10.
DOI: http://dx.doi.org/10.1145/2993600.2993604

```
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.

1.2 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:

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

1.3 (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 §1.2:

```
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:

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

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

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

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

[7] 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.