Demo: The Rust Language and Type System

Felix S. Klock II
Mozilla Research
pnkfelix@mozilla.com

Nicholas Matsakis
Mozilla Research
nmatsakis@mozilla.com

1. ABSTRACT

Rust is a new programming language for developing reliable and efficient systems. It is designed to support concurrency and parallelism in building applications and libraries that take full advantage of modern hardware. Rust's static type system is safe[1] and expressive and provides strong guarantees about isolation, concurrency, and memory safety.

Rust also offers a clear performance model, making it easier to predict and reason about program efficiency. One important way it accomplishes this is by allowing fine-grained control over memory representations, with direct support for stack allocation and contiguous record storage. The language balances such controls with the absolute requirement for safety: Rust's type system and runtime guarantee the absence of data races, buffer overflows, stack overflows, and accesses to uninitialized or deallocated memory.

In this demonstration, we will briefly review the language features that Rust leverages to accomplish the above goals, focusing in particular on Rust's advanced type system, and then show a collection of concrete examples of program subroutines that are efficient, easy for programmers to reason about, and maintain the above safety property.

If time permits, we will also show the current state of *Servo*, Mozilla's research web browser that is implemented in Rust.

Keywords

Rust, Systems programming, Memory management, Affine type systems

2. DEMONSTRATION OUTLINE

Rust shares several features with the ML family of languages, such as algebraic data types and parametric polymorphism. The syntax and semantics for simple Rust programs can be explained via analogy with corresponding ML programs.

However, Rust also deviates from ML in a number of ways. For this demonstration, we will be focusing on Rust's approach to managing memory, which is quite different from that of ML. In Rust, one can allocate objects inline within other objects; also, a Rust program may also work with (potentially mutable) references into the middle of allocated blocks of memory, *including* blocks that represent particular variants of an algebraic type. This introduces the potential for non-local and non-obvious aliasing.

To control aliasing and ensure type soundness, Rust uses an affine type system that tracks ownership. The unique owner of an object can hand that ownership off to new owner; but the owner may instead hand off *borrowed references* to (or into) the object. These so-called *borrows* obey lexical scope, ensuring that when the original reference goes out of scope, there will not be any outstanding borrowed references to the object (otherwise known as "dangling pointers"). This also implies that when the owner goes out of scope or is otherwise deallocated, the referenced object can be deallocated at the same time. Rust takes advantage of this property by supporting user-defined destructors, enabling RAII[2] patterns popularized by C++.

There are two flavors of borrows: mutable and immutable. Mutable references have a uniqueness property: There can be at most one active mutable borrow of a given piece of state (the owner itself is not allowed to mutate the object for the duration of the mutable borrow, nor are any of the inactive mutable borrows). Immutable references, on the other hand, can be freely copied and their referents can be the source for new immutable borrows (subject to the restriction that all the borrows still respect the lexical scope of the object's owner).

Rust functions can manipulate objects that are owned by local variables arbitrarily far up the control stack. Such functions need to support operations like `*ptr1 = *ptr2;` (writing the dereferenced value of `ptr2` into the memory referenced by `ptr1`), but must also respect the rules: such assignment statements must be prohibited from injecting dangling pointers.

[1] Type soundness has not yet been formally proven for Rust, but type soundness is an explicit design goal for the language.

[2] RAII: "Resource Acquisition Is Initialization"

Such functions need explicitly constrain their inputs that ensure that their execution will not break the rules. Rust has an optional explicit syntax for describing the *lifetime* bounds associated with a reference, which mixes with *lifetime-polymorphism* (analogous to *type-polymorphism*) to provide functions the expressive power they need to manipulate references, without breaking type safety.

Planned topics for the demonstration include:

- Memory safety in Rust
- Ownership and linear types
  - Data Allocation
  - Deallocation and (user-defined) Destructors
  - Inherited mutability
  - Moving and sending data
- Borrowing and freezing of mutable data structures
  - The necessity of (controlled, temporary) aliasing
  - Explicit and implicit lifetime parameters
- Discussion of design space
  - Comparison against other work
  - Moving values based on type, vs `move` keyword
  - Linear mutable values
  - Lexically scoped lifetimes