The Rust Language

Nicholas Matsakis
Mozilla Research
nmatsakis@mozilla.com

Felix S. Klock II
Mozilla Research
pnkfelix@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.

Categories and Subject Descriptors

D.3.0 [**E.2**]: Programming Language Standards

Keywords

Rust, Systems programming, Memory management, Affine type systems

2. MEMORY MANIPULATION IN RUST

Rust is a new programming language targeting systems-level applications. Like C++, Rust is designed to map directly to hardware, giving users control over the running time and memory usage of their programs. This implies, for example, that all Rust types can be allocated on the stack and that Rust never requires the use of a garbage collector or other runtime. Unlike C++, however, Rust also offers strong safety guarantees: pure Rust programs are guaranteed to be free of memory errors (dangling pointers, double frees) as well as data races.

To control aliasing and ensure type soundness, Rust incorporates a notion of *ownership* into its type system. The unique owner of an object can hand that ownership off to new owner; but the owner may also 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 leverages the latter property by adding support for user-defined destructors, enabling RAII[2] patterns as 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 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.

A function that manipulates borrowed references needs to constrain its input parameters to ensure that executing the function body will not break the rules. Rust has an optional explicit syntax for describing the *lifetime* bounds associated with a reference. Lifetime bounds mix together with *lifetime-polymorphism* (analogous to *type-polymorphism*) to provide functions the expressive power needed to manipulate memory references, without breaking type safety.

[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"

HILT 2014, October 18–21, 2014, Portland, OR, USA.
Copyright 2014 ACM 978-1-4503-3217-0/14/10 ...$15.00.
http://dx.doi.org/10.1145/2663171.2663188.