Rust: Type-Checked Object Ownership and Object Lifetime

By David Johnston

Recommended Version:
nightly (Mac installer)

Rust is a systems programming language that runs blazingly fast, prevents almost all crashes*, and eliminates data races.

Show me more!

Install

Other Downloads

Featuring

- zero-cost abstractions
- move semantics
- guaranteed memory safety
- threads without data races
- trait-based generics
- pattern matching
- type inference
- minimal runtime
- efficient C bindings

```
// This code is editable and runnable!
fn main() {
    // A simple integer calculator:
    // `+` or `-` means add or subtract by 1
    // `*` or `/` means multiply or divide by 2

    let program = "+ + * - /";
    let mut accumulator = 0;

    for token in program.chars() {
        match token {
            '+' => accumulator += 1,
            '-' => accumulator -= 1,
            '*' => accumulator *= 2,
            '/' => accumulator /= 2,
            _ => { /* ignore everything else */ }
        }
    }

    println!("The program \"{}\" calculates the value {}",
             program, accumulator);
}
```

Run

More examples

* In theory. Rust is a work-in-progress and may do anything it likes up to and including eating your laundry.

```
// This code is editable and runnable!
fn main() {
    // A simple integer calculator:
    // `+` or `-` means add or subtract by 1
    // `*` or `/` means multiply or divide by 2

    let program = "+ + * - /";
    let mut accumulator = 0;

    for token in program.chars() {
        match token {
            '+' => accumulator += 1,
            '-' => accumulator -= 1,
            '*' => accumulator *= 2,
            '/' => accumulator /= 2,
            _ => { /* ignore everything else */ }
        }
    }

    println!("The program \"{}\" calculates the value {}",
             program, accumulator);
}
```

Run

Run this on play.rust-lang.org: http://is.gd/J2mxNj

Development

- Started by Graydon Hoare
- Development led by Mozilla Research since 2009+
- Compiler has been self-hosted since 2011 (uses LLVM backend)
- Implementation language for Mozilla Servo
- Large & Active Open Source Community
- Currently, language and standard APIs are fast-changing
- 1.0-*alpha* recently released
- 1.0 release coming soon!

Why I’m Interested

- Rust includes theoretical and practical language design elements.
- Designed to be appropriate substitute for C++.
- Ideas taken from many functional and imperative languages.
- It is an *opinionated* language; the compiler gives strong safety guarantees.
- However, the programmer can opt into `unsafe` operations.

Why I’m Interested

Language and tooling are designed to be useful in modern software engineering *practice*.

SoftEng Usability Examples: Documentation, Testing, and Build Systems

Click or press 'S' to search, '?' for more options...

std

Modules

any
ascii
bitflags
borrow
boxed
cell
char
clone
cmp
collections
default
dynamic_lib
error
f32
f64
ffi
finally
fmt
hash
i16
i32
i64
i8

Module std::option | stable

[-] [+] [src]

[-] Optional values

Type `Option` represents an optional value: every `Option` is either `Some` and contains a value, or `None`, and does not. `Option` types are very common in Rust code, as they have a number of uses:

- Initial values
- Return values for functions that are not defined over their entire input range (partial functions)
- Return value for otherwise reporting simple errors, where `None` is returned on error
- Optional struct fields
- Struct fields that can be loaned or "taken"
- Optional function arguments
- Nullable pointers
- Swapping things out of difficult situations

Options are commonly paired with pattern matching to query the presence of a value and take action, always accounting for the `None` case.

```
fn divide(numerator: f64, denominator: f64) -> Option<f64> {
    if denominator == 0.0 {
        None
    } else {
        Some(numerator / denominator)
    }
}

// The return value of the function is an option
let result = divide(2.0, 3.0);

// Pattern match to retrieve the value
match result {
    // The division was valid
    Some(x) => println!("Result: {}", x),
    // The division was invalid
    None    => println!("Cannot divide by 0")
}
```

```
//! Optional values
//!
//! Type `Option` represents an optional value: every `Option`
//! is either `Some` and contains a value, or `None`, and
//! does not. `Option` types are very common in Rust code, as
//! they have a number of uses:
//!
//! * Initial values
//! * Return values for functions that are not defined
//!   over their entire input range (partial functions)
//! * Return value for otherwise reporting simple errors, where `None` is
//!   returned on error
//! * Optional struct fields
//! * Struct fields that can be loaned or "taken"
//! * Optional function arguments
//! * Nullable pointers
//! * Swapping things out of difficult situations
//!
//! Options are commonly paired with pattern matching to query the presence
//! of a value and take action, always accounting for the `None` case.
//!
//! ```
//! fn divide(numerator: f64, denominator: f64) -> Option<f64> {
//!     if denominator == 0.0 {
//!         None
//!     } else {
//!         Some(numerator / denominator)
//!     }
//! }
//!
//! // The return value of the function is an option
//! let result = divide(2.0, 3.0);
//!
//! // Pattern match to retrieve the value
//! match result {
//!     // The division was valid
//!     Some(x) => println!("Result: {}", x),
//!     // The division was invalid
//!     None    => println!("Cannot divide by 0")
//! }
//! ```
```

The Most Interesting Feature...

Safe and explicit memory management via *Ownership* and *Lifetimes*.

Traditional Problems With Explicit Memory Management

However, in providing low-level control, C admits a wide class of dangerous — and extremely common — safety violations, such as incorrect type casts, buffer overruns, dangling-pointer dereferences, and space leaks. As a result, building large systems in C, especially ones including third-party extensions, is perilous. Higher-level, type-safe languages avoid these drawbacks, but in so doing, they often fail to give programmers the control needed in low-level systems.

Grossman et. al., 2002

How These Problems Are Mitigated In Modern C++

```cpp
#include <vector>

int main() {
  std::vector<int> vec (3, 100);  // Stack-allocated data structure points to
                                  // three heap-allocated ints, each set to
                                  // 100.
}  // This is the end of `vec`'s "lifetime".
```

How These Problems Are Mitigated In Modern C++

```
#include <vector>

int main() {
  std::vector<int> vec (3, 100);  // Stack-allocated data structure points to
                                  // three heap-allocated ints, each set to
                                  // 100.
}  // This is the end of `vec`'s "lifetime".
```

The object's *destructor* is called at the end of its lifetime. Any memory it owns is freed in the process.

If the data structure is recursively defined, then its children should be freed as well.

Problems Remain

```cpp
#include <vector>
#include <foobar>

int main() {
  std::vector<int> vec (3, 100);
  foo(vec);  // What might `foo()` do with the memory which `vec` encapsulates?
}
```

Problems Remain

```cpp
#include <vector>
#include <foobar>

int main() {
  std::vector<int> vec (3, 100);
  foo(vec);  // What might `foo()` do with the memory which `vec` encapsulates?
}
```

It is hard to know whether `foo()` only "borrowed" `vec` and any resources (e.g. memory) that it encapsulates.

Rust prevents uncertainty about `foo()` with its type system. Rust uses information from the interface of `foo()` to enforce proper ownership and liveness at all times.

Some Prior Work

- **Ruggieri and Murtaugh, 1988:** Static lifetime analysis for some heap-allocated objects.
- **Baker, 1990:** Extending Hindley-Milner type inference to include storage-use inference.
- **Tofte and Talpin, 1997:** Implementing region-based memory management to remove garbage collection from Standard-ML; implementing ML-Kit.
- **Walker and Watkins, 2001:** Presented a formal model which combined a linear type system with regions.
- **Swamy et. al., 2006:** Described Cyclone, a C-like language combining
    - Statically Scoped Regions
    - Tracked Pointers

Some Prior Work

- **Ruggieri and Murtaugh, 1988:** Static lifetime analysis for some heap-allocated objects.
- **Baker, 1990:** Extending Hindley-Milner type inference to include storage-use inference.
- **Tofte and Talpin, 1997:** Implementing region-based memory management to remove garbage collection from Standard-ML; implementing ML-Kit.
- **Walker and Watkins, 2001:** Presented a formal model which combined a linear type system with regions.
- **Swamy et. al., 2006:** Described Cyclone, a C-like language combining
    - Statically Scoped Regions
    - Tracked Pointers

Cyclone had the most direct influenced Rust's ownership and lifetime model.

A Variable’s Lifetime

```
fn main() {
    let vec = vec![100, 100, 100];  // Stack-allocated data structure points to
                                    // three heap-allocated ints, each set to
                                    // 100.
}  // This is the end of `vec`'s lifetime.
```

The variable is *dropped* at the end of its lifetime. Any memory it owns is freed in the process.

If the data structure is recursively defined, then its children are freed as well.

Rust Makes *Borrowing* Explicit

```
fn foo_borrow(vec: &Vec<u32>) {  // Notice the &.
    // ...
}

fn main() {
    let vec = vec![100, 100, 100];
    foo_borrow(&vec);
}
```

The & represents:

- An address-of operation in C/C++
- A reference in C++
- A *borrow* in Rust

Because of the type signature of `foo_borrow()`, the function body is guaranteed to not stash a reference any memory encapsulated by `vec`.

The callee is guaranteed to give back all ownership of the object to the caller.

Rust Makes *Moving* Ownership Explicit

```
fn foo_move(vec: Vec<u32>) {    // Notice the lack of an &.
    // ...
}

fn main() {
    let vec = vec![100, 100, 100];
    foo_move(vec);
}
```

Because of the type signature of `foo_move()`, the caller takes ownership of `vec`.

After `foo_move(vec)` has been called, `vec` can no longer be used in the body of `main()`.

Key Idea Behind Borrow-References

Every borrow-reference has a statically known type. A borrow-reference's type includes both

- the referent's *type*
- the referent's *lifetime*

Key Idea Behind Borrow-References

Every borrow-reference has a statically known type. A borrow-reference's type includes both

- the referent's *type*
- the referent's *lifetime*

```
fn main() {
    let vec = vec![100, 100, 100];
    let vec_ref = &vec;
    println!("{:?}", *vec_ref);
}
```

Lifetime Inference

What are the consequences of having a reference aliasing or a reference *into* some object?

For memory-safety guarantees, we need to make sure that the lifetime of any variable aliasing or pointing into a data structure does not live longer than the object itself.

This applies to both heap and stack allocated objects

```
struct Pair {
    x: u32,
    y: u32
}

// Lifetime of returned the reference needs to be tracked.
fn fst(pair: &Pair) -> &u32 {
    &(pair.x)
}
```

Local Reference Lifetime Inference

In order to guarantee memory safety, the Rust type checker makes sure that an alias to an object never outlives the object itself. (Prevent use-after-free.)

Said another way, the lifetime of a reference is guaranteed to be shorter than or equal to the lifetime of the object itself.

Local reasoning doesn’t seem so hard.

```
fn main() {
    let vec = vec![100, 100, 100];
    let vec_ref = &vec;
    println!("{:?}", *vec_ref);
}
```

Modular Reference Lifetime Inference

```
struct Pair {
    x: u32,
    y: u32
}

fn fst(pair: &Pair) -> &u32 {
    &(pair.x)
}
```

Modular reasoning about the lifetime of the return value is less clear, it is not clear exactly what the lifetime of the given pair will be.

Modular Reference Lifetime Inference

```
fn fst(pair: &Pair) -> &u32 {
    &(pair.x)
}

// Lifetime of returned reference is inferred to be the same as the lifetime of
// the given `pair`. Without lifetime inference the function definition would
// look like this:
fn fst<'a>(pair: &'a Pair) -> &'a u32 {
    &(pair.x)
}
```

Here, we have two borrow-references: `pair` and our unnamed return value. Both have a lifetime associated with their type at compile-time.

Exactly what this lifetime is depends on the context in which `fst()` was called, which is why the lifetime is *parameterized*, not explicit.

Named Lifetimes

Named lifetimes are used in function declarations to let the programmer specify how the lifetime of a returned reference relates to the lifetime of the arguments.

In practice, this information is usually either unnecessary or inferred.

```
fn fst<'a>(pair: &'a Pair) -> &'a u32 {
    &(pair.x)
}
```

Compare Named Lifetimes with Type Parameters

```
struct Pair<T: Int> {
    x: T,
    y: T
}

fn fst<T: Int>(pair: &Pair<T>) -> T {
    pair.x
}

fn main() {
    let pair = Pair { x: 0, y: 1 };
    let x = fst(&pair);
    println!("{:?}", x);
    println!("{:?}", pair.y);
}
```

Compare Named Lifetimes with Type Parameters

```
fn fst<T: Int>(pair: &Pair<T>) -> T {
    pair.x
}
```

```
fn fst<'a>(pair: &'a Pair) -> &'a u32 {
    &(pair.x)
}
```

Named Lifetimes

To reiterate, every borrow-reference's type includes both the referent's type and the referent's *lifetime*.

Lifetimes are thus a core part of the type system.

```
fn fst<'a>(pair: &'a Pair) -> &'a u32 {
    &(pair.x)
}
```

Other Smart Pointer Types

- Borrow References: `&T`
- Unique “Boxed” Object Pointer: `Box<T>`
- Reference Counted Pointer: `Rc<T>`
- Asynchronous Reference Counted Pointer: `Arc<T>`

Other Smart Pointer Types

- Borrow References: `&T`
- Unique “Boxed” Object Pointer: `Box<T>`
- Reference Counted Pointer: `Rc<T>`
- Asynchronous Reference Counted Pointer: `Arc<T>`

These last three are like `Vec<T>`, in the sense that they can be a stack-allocated handle to encapsulate heap-allocated memory.

Thread-Safe Communication By Moving Ownership

The simplest way to create a channel is to use the `channel` function to create a `(Sender, Receiver)` pair. In Rust parlance, a *sender* is a sending endpoint of a channel, and a *receiver* is the receiving endpoint. Consider the following example of calculating two results concurrently:

```
use std::thread::Thread;
use std::sync::mpsc;

let (tx, rx): (mpsc::Sender<u32>, mpsc::Receiver<u32>) = mpsc::channel();

Thread::spawn(move || {
    let result = some_expensive_computation();
    tx.send(result);
});

some_other_expensive_computation();
let result = rx.recv();

fn some_expensive_computation() -> u32 { 42 } // very expensive ;)
fn some_other_expensive_computation() {}      // even more so
```

Comparison with Java

- In Java, is a common design pattern to enclose objects in an anonymous class instance.
- To promote thread safety, enclosed objects must be marked `final` (i.e. immutable).
- The idea is to give the object over to the other thread.
- Rust, on the other hand, includes this idea of moving an object’s ownership in the type checking system.

Questions?