Proseminar “The Rust Programming Language”
Summer Term 2017

Garbage Collection & Reference Counting

Clemens Ruck & Alex Egger
Technische Universität München

July 22, 2017

Abstract

This paper aims to give a broad overview of existing memory management techniques and a quick introduction to the technique employed by the Rust programming language. Further it will demonstrate the different possibilities when working with memory in the Rust language, and introduce to the different types commonly used.

1 Introduction

2 Garbage Collection

2.1 General principle

2.2 Reference Counting

Reference counting is a simple garbage collection algorithm, that uses reference counts to deallocate objects that are no longer referenced, and thus free the underlying memory. Here reference count refers to an internal counter, that tracks the amount of active references to a specific object. Every time an active reference is destroyed the internal reference counter for that object is decremented. Correspondingly when a new reference to an object is created, the reference counter is incremented, to reflect the number of active references. When an objects reference count falls to zero, the object becomes inaccessible. The memory used by such an inaccessible object can be then be freed safely.

Reference counting seems superior to any tracing algorithm, because of its simplicity, but there are a few caveats, that are not visible at first glance. Firstly when handling a large amount of objects a deletion may cause a large amount of objects to be freed in a chain reaction. This chain reaction can then use up valuable processing time, resulting in a largely unresponsive application for a user. This pitfall can be circumvented with the following approach: Whenever an object should be ordinarily deleted, it is instead added to a list of objects that are to be freed. The list can then be processed at another point in time, effectively making the whole technique incremental.

A more severe problem is posed by *reference cycles*, where multiple references reference each other, leading to a non-tree-like structure. These cycles can not be reclaimed, since each object depends on another being freed first. These shortcomings render reference counting unsatisfactory in most contexts, but the following situations:

- Reference loops are impossible.
- Modifications of references are infrequent.
- Memory constraints are very tight.

2.3 Ownership & Lifetimes

Memory safety is one of Rusts primary goals and it achieves it also due to the strict rules about ownership and lifetimes employed by the compiler. Rust requires a variable to have *exactly one* owner, where owner means variable binding. This can be understood when looking at figure 1 closely. First we define a `struct Owned` type. We then instantiate a new object of this type and bind it to the binding with the name `x`. Next we tell the binding `y` to take ownership of the object `x` currently owns. If we then try to use `x` again, we will be greeted with a compiler error, since we just broke the ownership rules. Once `y` took ownership the other binding was invalidated, and thus the rule of having exactly one owner was restored. Note here that this example would be optimized away by the compiler, since `Owned` is a Zero Sized Type[3]. When an owner goes out of scope, its corresponding value is cleaned up. Further Rust enforces strict rules about references or borrowing rules, that can be summed up as:

- There may be *one or more* `&T` references to an object at any time.
- There may be *exactly one* `&mut T` reference to an object at any time.

These choices are *mutually exclusive*. By making it impossible to have `&T` and `&mut T` references at the same time Rust tries to avoid data races, where one thread reads data with its `&T` reference, while the other writes using its `&mut T` reference. In figure 2 we create a variable binding called `x`. Then we create a mutable reference to this same variable. And then we'd like to print it's content to the screen. But this code leads to a compiler error, namely "error[E0502]: cannot borrow 'x' as immutable because it is also borrowed as mutable". The function underlying the `println!` macro takes a immutable reference to its argument. So implicitly one was created and the borrowing rules were broken, since we already had an active `&mut u32` reference.
To avoid dangling references Rust uses lifetimes. Every reference has a lifetime associated with it. Usually the compiler infers these lifetimes and simply looks at how long the owner of the object lives.

```rust
#[derive(Debug)]
struct Owned {}

let x = Owned{};

// Y takes ownership of the Owned object.
let y = x;

// This won't work.
println!("{:?}", x);
```

Figure 1: An example of transferring ownership of a variable.

```rust
let mut x: u32 = 42;

// Create a immutable reference.
let x_ref = &mut x;

// This leads to a compiler error!
println!("{}", x);
```

Figure 2: Breaking the borrowing rules.

```rust
let x;
    {
        let y = 32;
        x = &y;
    }
println!("{}", x);
```

Figure 3: A dangling reference caught by lifetimes.

```rust
fn main() {
    let i: u8 = 42;
    let b = Box::new(i);
}
```

Figure 4: A program that stores an u8 on the heap.

```rust
pub struct Box<T> where T: ?Sized {
    pointer: *mut T,
    size: usize,
}
```

Figure 5: A simplified example definition of the Box type.

3 Using memory in Rust

3.1 Allocating data on the heap

It is common practice to store long-lived data on the heap section of the memory, as the lifetime of the data is then decoupled of the lifetime of its local function. Rust allows allocating data on the heap by using the `Box<T>` struct. A `Box` may be created by calling the `Box::new()` method, as can be seen in figure 4. It may seem like there is some kind of magic underneath the `Box` type, but that would be contrary to Rusts goals. In fact a `Box` is only a wrapper around a raw pointer (see 3.4), that points to the heap, and the size of the object that is stored. A simple implementation of this `Box` type is shown in figure 5. When not fully accustomed to the Rust type system one may be puzzled by the `where`-clause in the struct definition. This clause simply says the type that is stored in the `Box` must have a fixed size and may not be a Dynamically Sized Type[1], like for example a Trait.

Now there is still the question of how the memory used by a `Box` is freed. The `Box` struct implements the `Drop` trait. The documentation for this trait describes it as follows: “The `Drop` trait is used to run some code when a value goes out of scope. This is sometimes called a destructor”[6]. The `Box` type uses this trait to free the memory reserved on the heap, when it goes out of scope.

3.2 Reference Counting in Rust

3.3 Interior Mutability with Cell types

“*Interior Mutability* is a design pattern in Rust for allowing you to mutate data even though there are immutable references to that data, which would normally be disallowed by the borrowing rules. The interior mutability pattern involves using unsafe code inside a data structure to bend Rust’s usual rules around mutation and borrowing.”[4]. There are two types that can be used for this pattern, `Cell<T>` and `RefCell<T>`. These types are the owners of the type the encapsulate and they store it on the heap, similar to a `Box<T>`. The main difference between a `Box` and one of the cell types is that borrowing rules are applied at compile-time for the `Box`, but at runtime for the cell types. These types can be used to ensure to the compiler that code that may look like it breaks the borrowing rules at compile-time, actually upholds the borrowing rules at runtime. Should the code break the rules nevertheless, a `panic` will be thrown at runtime. The difference between the `Cell<T>` and `RefCell<T>` is that `Cell<T>` is only usable for types that implement the `Copy` trait, because of the way `Cell<T>` handles interior mutability. It mutates the data in the container by copying it outside, mutating it, and copying it back inside. Obviously only `Copy`-types can be used with this. `RefCell<T>` can be used with any type that implements the `Sized` trait. It uses references to mutate the data, without using the copy mechanism. Figure 6 shows how to construct both a `Cell` and a `RefCell` and how to borrow dynamically at runtime.

3.4 Raw pointers

Rust has two kinds of references that allow pointing to an arbitrary place in memory. These references are called *raw pointers*. The two types are `*const T` and `*mut T`. How to create such raw pointers is shown in figure 7 and in figure 8. The difference between the `const` and the `mut` variants is simply whether the value in the underlying memory

```rust
fn main() {
    let i = 32;
    let cell = Cell:new(i);

    let refcell = RefCell::new(i);

    let r: &u32 = refcell.borrow();
    let mut r_mut: &mut u32
        = refcell.borrow_mut();
}
```

Figure 6: Constructing a `Cell` and a `RefCell` object and borrowing dynamically.

```rust
let count: u32 = 65;
let ptr = &count as *const u32;
let mut_ptr = &mut count as *mut u32;
```

Figure 7: Creating two raw pointers from a reference to an `u32`.

address can be mutated through the pointer, or not. In the example in figure 7 we created two pointers to the same address. One begin a `const *u32` and the other a `mut *u32`. If we were to create a `&u32` and a `&mut u32` we would get a compiler error, but when working with raw pointers the usual borrowing rules don't apply.

3.5 Writing unsafe Code

In some situations a programmer may need to write code, that the static guarantees of the compiler would reject. Such situations may for example occur whenever raw pointers (see 3.4) are used to work around some restriction of Rusts borrowing mechanism. Another possibility is whenever Rust tries to interact with hardware directly, since hardware does not adhere to Rusts rules. The `unsafe` keyword can be used to communicate to the compiler,

```rust
let address = 0x123456;
let ptr = address as *mut u32;
```

Figure 8: Creating a raw pointer from an address.

```rust
let count: u32 = 128;
let ptr = &count as *mut u32;
unsafe {
    *ptr = 130;
}
```

Figure 9: Dereferencing a raw pointer and assigning a new value.

that the designated block contains 'unsafe' code. 'Unsafe' here means exactly the following behaviors:

1. Dereferencing a raw pointer.
2. Calling an unsafe function or method.
3. Accessing or modifying a mutable static variable.
4. Implementing an unsafe trait.
5. Accessing or modifying a field of a `union` type.

[5].

Dereferencing a raw pointer is considered unsafe, because raw pointers (see 3.4) may point to arbitrary memory. This mean a raw pointer could point to memory not even reserved by the program or to NULL. When dereferencing such a pointer care must be taken to check whether the pointer is even valid first. An example of this can be seen in figure 9. Since the pointer here will be always be valid it is unnecessary to check it for validity.

Calling an unsafe function or method is unsafe behavior, because these unsafe functions generally assume that some invariant is true before they are called. A programmer may have unknowingly broken the invariant before calling the function, resulting in a memory violation or other unwanted behavior. An example of this can be seen in figure 10, in which a function called `add_three` takes a mutable raw pointer (see 3.4), offsets it by one, deferences it, and adds three to it. Obviously this is not memory-safe when the pointer does not own

```rust
unsafe fn add_three(ptr: *mut u8) {
        *(ptr.offset(1)) += 3;
}
```

Figure 10: An unsafe function, that adds three to a pointer offset by one.

```rust
static mut COUNT: u8 = 0;

fn main() {
    unsafe {
        COUNT = 1;
    }
}
```

Figure 11: Updating a static mutable variable in an `unsafe` block.

the element after it in memory, and as such the function must be marked `unsafe`. The `unsafe` keyword in the function definition additionally acts like an `unsafe` block around the whole function body, which is the reason the dereferencing of the pointer in figure 10 is not wrapped in an `unsafe` block.

Accessing or updating a static mutable variable is defined to be unsafe behavior. Static mutable variables are comparable to global state in other programming languages. Hoare describes Rust as having a "safe concurrency model"[2]. This is especially visible when looking at the concept of borrowing, where Rust enforces safe concurrency through the restrictions on `&T` and `&mut T` references. Additionally it is Rusts responsibility to ensure data races on global state can not happen. Thus the Rust compiler will not allow accessing or updating such global state, without declaring the operation as `unsafe`, after which it is the programmers responsibility to ensure correct handling of data races. An example of updating a static mutable variable can be seen in figure 11.

Implementing an unsafe trait is similar to an unsafe function. Designating a block as `unsafe` tells the compiler, that there is some invariant that

```rust
unsafe trait UnsafeTrait {}

unsafe impl UnsafeTrait for u8 {}
```

Figure 12: The structure of an unsafe trait and its implementation.

```rust
struct Test {
    ptr: *mut u8, // Doesn't
                  // implement Send.
}

unsafe impl Send for Test {
// Somehow make the
// raw pointer thread-safe.
}
```

Figure 13: Implementing the `Send` trait for a struct with non-`Send` fields.

he cannot check by himself, but that is crucial for program execution. The structure of such an `unsafe trait` can be seen in figure 12. A common example of this is when implementing the `Send` trait for a type the compiler cannot automatically derive it for. The documentation for the `Send` trait says "This trait is automatically implemented when the compiler determines it's appropriate."[7]. Appropriate here means when all fields of the type implement the `Send` trait too. Have a look at figure 13 in which he have a type `Test` that has a raw pointer as a field. Raw pointers do not implement the `Send` trait, since they have no guarantees about being used in shared environments. This leads to our struct `Test` not automatically having the `Send` trait. We have to implement it manually with an `unsafe impl`-block as shown.

Accessing or modifying a field of a `union` type is unsafe by definition. Unions are similar to Enums, but do not store a tag to identify the variant used. They were first introduced in stable Rust 1.19 (released on: 20-07-2017)[8]. Without this tag the compiler does not know, which variant we want to use, and can not make sure we access it

```
union MyUnion {
    f1: u32,
    f2: u16,
}

let u = MyUnion{f2: 45};

unsafe {
    u.f2 = 3;
};

let v = unsafe {u.f2};
```

Figure 14: Accessing and modifying a field of a `union`.

correctly.

4 Conclusion

5 Future work

References

[1] Dynamically sized types in Rust. `https://blog.babelmonkeys.de/2014/03/18/dst.html`. Accessed: 2017-07-20.

[2] Interview on Rust, a Systems Programming Language Developed by Mozilla. `https://www.infoq.com/news/2012/08/Interview-Rust`. Accessed: 2017-07-17.

[3] Rust Nomicon - Exotically Sized Types. `https://doc.rust-lang.org/nomicon/exotic-sizes.html#zero-sized-types-zsts`. Accessed: 2017-07-20.

[4] The Rust Book, Second Edition - Interior Mutabilty. `https://doc.rust-lang.org/book/second-edition/ch15-05-interior-mutability.html`. Accessed: 2017-07-17.

[5] The Rust Book, Second Edition - Unsafe Rust. `https://doc.rust-lang.org/stable/book/second-edition/ch19-01-unsafe-rust.html`. Accessed: 2017-07-14.

[6] The Rust Language API Documentation - Drop Trait. `https://doc.rust-lang.org/std/ops/trait.Drop.html`. Accessed: 2017-07-14.

[7] The Rust Language API Documentation - Send Trait. `https://doc.rust-lang.org/std/marker/trait.Send.html`. Accessed: 2017-07-17.

[8] The Rust Programming Language Blog - Announcing Rust 1.19. `https://blog.rust-lang.org/2017/07/20/Rust-1.19.html`. Accessed: 2017-07-21.