Proseminar “The Rust Programming Language” Summer Term 2017
Garbage Collection & Reference Counting

Clemens Ruck & Alex Egger
Technische Universität München

July 16, 2017

Abstract

1 Introduction

2 Paper organization

The main part of a scientific paper is organized in a somewhat similar way each time:

1. You announce, that you will tell about something in the *abstract*
2. You introduce to what you did in the *introduction*
3. You clarify the amount of your specific contribution in the *related work*
4. You make the paper well-founded, by introducing necessary standards in the *basics part*
5. You actually tell about what you did in the *main part*
6. *Optional, if possible:* You evaluate in as neutral a way as possible what You have done in the *evaluation*
7. You summarize, and finally draw conclusions on what you have done in the *conclusion*
8. You sketch, what could be achieved next, based on this work in the *future work*

3 Basics

3.1 Garbage Collection

3.1.1 General principle

3.1.2 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. Everytime 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 visibile 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 ordinarly

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 depdends 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.

3.2 Rust

4 Using memory in Rust

4.1 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 1. 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 4.5), 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 2. 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, 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”[2]. The `Box` type uses this trait to free the memory reserved on the heap, when it goes out of scope.

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

Figure 1: A program that stores an `u8` on the heap.

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

Figure 2: A simplified example definition of the `Box` type.

4.2 Reference Counting in Rust

4.3 Interior Mutability with Cell types

4.4 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 4.5) 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, that the designated block contains 'unsafe' code. 'Unsafe' here means exactly the following behaviours:

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.

[1].

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

Figure 3: An unsafe function, that adds three to a pointer offset by one.

Calling an unsafe function or method is unsafe behaviour, 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 behaviour. An example of this can be seen in figure 3, in which a function called `add_three` takes a mutable raw pointer (see 4.5), offsets it by one, derefences it, and adds three to it. Obviously this is not memory-safe when the pointer does not own the element after it in memory, and as such the function must be marked `unsafe`. The `unsafe` keyword in the function definition additonally acts like an `unsafe` block around the whole function body, which is the reason the dereferencing of the pointer in figure 3 is not wrapped in an `unsafe` block.

4.5 Raw pointers

5 Conclusion

6 Future work

References

[1] 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.

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