Proseminar "The Rust Programming Language"
Summer Term 2017

Garbage Collection & Reference Counting

Clemens Ruck & Alex Egger
Technische Universität München

July 14, 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 displays characteristics of a simple garbage collection technique, however there are a few pitfalls 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. A more severe problem is posed by *reference cycles*, where multiple references reference each other, leading to

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

Figure 1: A program that stores the integer 42 on the heap.

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:

- Creation of loops is impossible
- Modifications of reference counts are infrequent
- Situations with strong memory constraints

3.2 Rust

4 Main

4.1 Allocating data on the heap

It is common to store long-lived data on the heap section of the memory, as the lifetime of the data is then not decided by the lifetime of its local function. Rust allows allocating data on the heap by using the `Box<T>` type[1]. An example of this can be seen in figure 1, where the `Box::new()` method is called to construct a `Box` object containing an integer, effectively placing it on the heap. A conceptual implementation of such a `Box` is given in figure 2.

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

impl<T> Box<T> where T: ?Sized {
    pub fn new(data: T) -> Self {
        // Allocate memory on the
        // heap for the pointer.
        Box {
            pointer: ...,
            size: std::mem::
                  size_of::<T>(),
        }
    }
}

impl<T> Drop for Box<T> {
    fn drop(&mut self) {
        // Free the memory
        // allocated on the heap.
    }
}
```

Figure 2: An exemplary implementation of the `Box` type.

4.2 Reference Counting in Rust

4.3 Raw pointers

5 Conclusion

6 Future work

References

[1] The Rust Language API Documentation - Box Struct. `https://doc.rust-lang.org/std/boxed/struct.Box.html`. Accessed: 2017-07-14.