Proseminar “The Rust Programming Language”
Summer Term 2017

Garbage Collection & Reference Counting

Clemens Ruck & Alex Egger
Technische Universität München

July 15, 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 Unsafe Code

In some scenarios a programmer may write a ’safe’ program, that the Rust compiler deems unsafe. The chapter ’unsafe’ of the first edition of ’The Rust Book’ [1] mentions that safety checks are conservative by nature: there are programs that are actually safe, but the compiler is not able to verify this is true. Usage of the `unsafe` keyword allows to notify the compiler to be less strict in the block that was marked. Concretely such a designated block allows:

1. Accessing and updating mutable static variables.
2. Dereferencing raw pointers.
3. Calling functions marked as unsafe.

A mutable static variable is comparable to a global variable in most other programming languages. Accessing and updating these is deemed an unsafe action, since Rust was conceived with concurreny in

mind. One thread may write to the variable, while another reads, leading to a data race.
Dereferencing a raw pointer (see 4.5) is unsafe, because Rust allows arbitrary pointer arithmetic, which means a pointer may point to memory not even assigned to the program, to memory an unexpected object occupies, or to NULL. This is one of the primary causes of many bugs in the C language, and one of the aspects Rust tries to circumvent by its strict ownership and lifetime rules.
Calling functions marked unsafe must be unsafe, since these functions usually make some assumption about the data they work with. If these assumptions are broken one may end up with unexpected behaviour.
The chapter 'unsafe' of first edition of the 'The Rust Book'[1] mentions the following as behaviours that may be undesirable, but not explicitely unsafe:

- Deadlocks
- Leaks of memory or other resources
- Exiting without calling destructors
- Integer overflow

Further it mentions that there are undefined behaviours, that must be avoided when writing unsafe code:

- Data races
- Dereferencing a NULL or dangling raw pointer
- Reads of uninitialized memory
- Breaking the pointer aliasing rules with raw pointers
- `&T` and `&mut T` follow LLVM's scoped noalias model, expect if the `&T` contains an `UnsafeCell<T>`. Unsafe code must not violate these aliasing guarantees.
- Mutating an immutable value/reference without `UnsafeCell<T>`
- Invoking undefined behaviour via compiler intrinsics:
  - Indexing outside of the bounds of an object with `std::ptr::offset` (offset intrinsic), with the exception of one byte past the end which is permitted.
  - Using `std::ptr::copy_nonoverlapping_memory` (memcpy32/mempy64, intrinsics) on overlapping buffers.
- Invalid values in primitive types, even in private fields/locals:
  - NULL/dangling references or boxes
  - A value other than `false` (0) or `true` (1) in a `bool`
  - A discriminant in an `enum` not included in its type definition
  - A value in a `char` which is a surrogate or above `char::MAX`
  - Non-UTF-8 byte sequence in a `str`
- Unwinding into Rust from foreign code or unwinding from Rust into foreign code.

4.5 Raw pointers

5 Conclusion

6 Future work

References

[1] The Rust Book, First Edition - Unsafe. https://doc.rust-lang.org/book/first-edition/unsafe.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.