TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Garbage Collection & Reference Counting

Memory management in Rust

Clemens Ruck & Alex Egger

Summer Term 2017

Overview

Methods of Memory Management

1. Problems of Manual Memory Management
2. Garbage Collection
3. Rust's approach

Memory Management in Rust

1. Stack Allocation
2. Heap Allocation
3. Reference Counting in Rust
4. The 'unsafe' keyword

Manual memory management

To dynamically allocate memory in C:

```
void *malloc(size_t size);
```

and to free it:

```
void free(void *ptr);
```

⚠ It is easy to forget if and when memory was freed, which leads to many avoidable bugs!

Problems - Memory Leaks

```c
int main(void) {
    while(1) {

        // Allocate some amount of memory on the heap.
        char *c;
        if(!(c = malloc(20 * sizeof(char)))) {
            perror("Could not allocate memory on heap.");
            return 1;
        }

        // Do something with allocated memory.

        // Memory is never freed, and can never be
        // reclaimed by the system.
    }
    return 0;
}
```

Problems - Use after Free

```c
int main(void) {
    // Allocate memory, what a surprise.
    char *c = malloc(10);

    // Do something and then free the memory.
    free(c);

    // The memory now doesn't belong to us anymore, so
    // this will result in a segmentation fault.
    (*c)++;

    return 0;
}
```

Garbage Collection

Garbage Collection is a form of automatic memory management. It attempts to reclaim memory used by objects that are no longer in use.

Example - Mark & Sweep

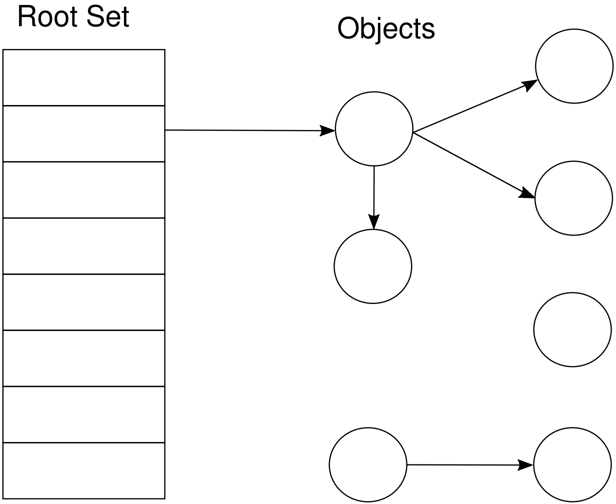


Figure: A graph-represenation of alive objects.

Example - Mark & Sweep

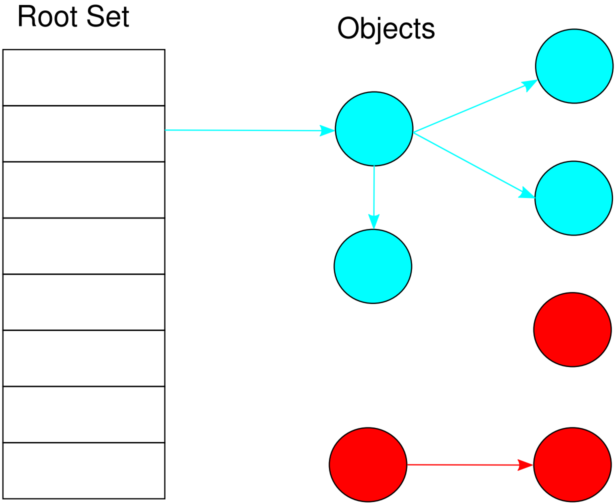


Figure: The 'Mark' stage of the algorithm.

Example - Mark & Sweep

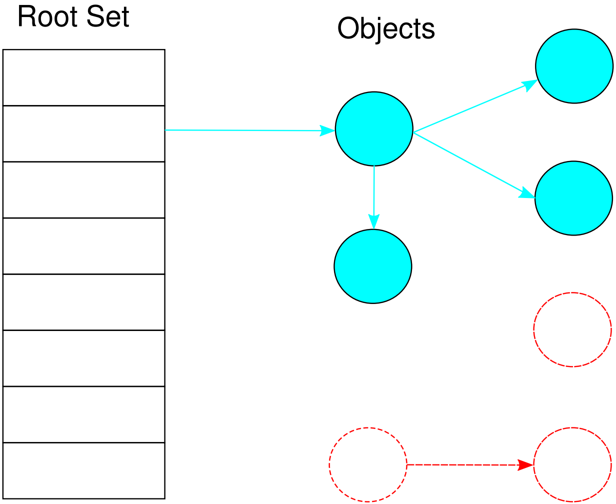


Figure: The 'Sweep' stage of the algorithm.

Memory management in Rust

Values are stack-allocated per default:

```
let x = 12; // Allocated on the stack.
```

To allocate on the heap use `Box<T>`:

```
let y = Box::new(56); // On the heap.
```

Stack Allocation

After line 2:

Address	Name	Value
0	x	42

```
1 fn main() {
2     let x = 42;
3     other();
4 }
5
6 fn other() {
7     let y = 27;
8     let z = 99;
9 }
```

Stack Allocation

After line 8:

Address	Name	Value
2	z	99
1	y	27
0	x	42

```
1 fn main() {
2     let x = 42;
3     other();
4 }
5
6 fn other() {
7     let y = 27;
8     let z = 99;
9 }
```

Stack Allocation

After line 3:

Address	Name	Value
0	x	42

```
1 fn main() {
2     let x = 42;
3     other();
4 }
5
6 fn other() {
7     let y = 27;
8     let z = 99;
9 }
```

Heap Allocation

Adress	Name	Value
1	y	???
0	x	42

```
fn main() {
    let x = 42;
    let y = Box::new(39);
}
```

Heap Allocation

Adress	Name	Value
ffff		39
1	y	
0	x	$->$ ffff

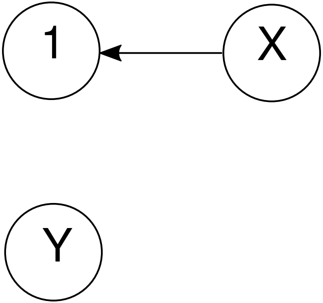
```rust
fn main() {
    let x = 42;
    let y = Box::new(39);
}
```

Comparison: Heap vs. Stack

1. Managing the Stack is trivial
2. Managing the Heap is non-trivial
3. Having stack-allocation as the default allows easier reasoning about the lifetimes of objects

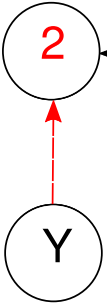
Reference Counting

References can be represented as a directed graph, where the vertices are objects and there is an edge between the nodes if one holds a reference to the other.



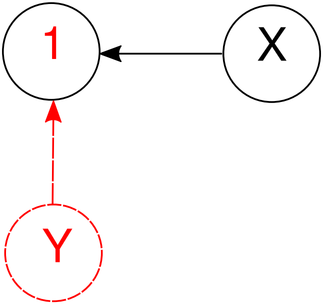
Reference Counting

Everytime a new reference to an object is created, the **reference counter** is incremented.



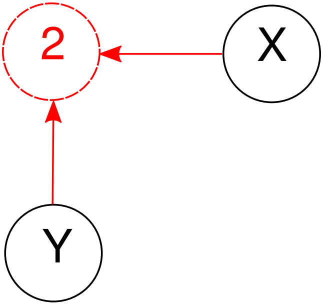
Reference Counting

When an object is freed all references it holds are freed too, and the respective reference counters are decremented.



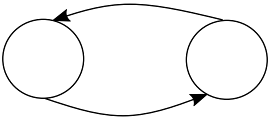
Reference Counting

Objects can only be freed, when their **reference count is 0!**
Otherwise we'll end up with dangling references.



Limitations

⚠ Reference cycles can never be reclaimed!



This can be solved by the use of a dedicated incremental garbage collector, that specifically targets reference cycles. An other approach is to simply disallow cycles in your data structure.

`Rc<T>` in Rust

In Rust we can make use of reference counting with the `Rc<T>` struct. To use it simply call `Rc::new`:

```
let t = ...;
let rc_t = Rc::new(t);
```

To create a new reference and increment the reference counter, use `Rc.clone()`:

```
let second_rc_t = rc_t.clone();
```

⚠ To use reference counting with multi-threaded applications, use `Arc<T>`, which is slightly slower.

Reference Counting - Example

```rust
struct Owner {
    name: String,
    // Fields...
}

struct Car {
    owner: Rc<Owner>,
    // Fields...
}
```

Reference Counting - Example

```
fn main() {
    let owner = Rc::new(Owner{
        name: "Lars",
        // Fields...
    });

    let car = Car {
        owner: owner.clone(),
        // Fields...
    }

    let car2 = Car {
        owner: owner.clone(),
        // Fields...
    }
    // ...
}
```

Reference Counting - Example

```rust
fn main() {
    // ...

    // Drop the local variable 'owner'
    drop(owner);

    // This will still work,
    // since the owner binding survives using Rc!
    println!("{}", car.owner.name);
}
```

The 'unsafe' keyword

The `unsafe` keyword allows:

1. Accessing or updating of a static mutable variable
2. Dereferencing of a raw pointer
3. Calling of other unsafe functions

Raw Pointers

Two types of raw pointers:

1. `*const T`
2. `*mut T`

Raw pointers have no lifetime or ownership. The only guarantee provided is they cannot be dereferenced except in code marked as `unsafe`.

Example

```rust
fn main() {
    let i: u32 = 77;

    // Creating a raw pointer in safe code is
    // perfectly acceptable.
    let x = &i as *const u32;

    // Dereferencing a raw pointer in
    // safe code is not allowed!
    let y = *x;

    // Once we marked the code as unsafe,
    // the compiler let's us do it.
    unsafe {
        let z = *x;
    }
}
```