UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

COS700 Research

Design Pattern Metaprogramming Foundations in Rust

Abstract Factory and Visitor

Student number: u19239395

Supervisor:
Dr. Linda Marshall

??? 2020

Abstract

Keywords:

1 Introduction

2 Design Patterns

2.1 Abstract Factory

2.2 Visitor

3 Metaprogramming

4 Rust

Rust is a new language created by Mozilla with the aim on being memory safe yet performant. Traditionally, most languages that are memory safe will make use of a garbage collector. While languages that are fast uses memory management.

4.1 Ownership

To achieve both memory safety and performance, Rust uses a less popular method known as ownership. Here the compiler keeps track of which variable owns a piece of memory. Each memory piece can only be owned by one variable at a time. Once the variable goes out of scope, the compiler knows it is the place to insert the memory cleanup code.

```
1  {
2      let s = String::from("string");
3      let t = s;
4  
5      println!("String len is {}", s.len()); // borrow of moved value: `s`
6  } // Compiler will add memory clean up code for t here
```

Listing 1: Example of ownership transfer

Ownership is manifested in three forms. The first form is moving ownership. Referring to the scope in listing 1, on line 2 a new on heap memory object is created and assigned to variable *s*. Line 3 has the memory from *s* assigned to *t*. But, because *s* is a heap object, the compiler transfers ownership of the memory from *s* to *t*.

When trying to use the memory on line 5, via *s*, the compiler therefore throws an error saying that *s* was moved. In fact any reference to *s* after line 3 will always give a compiler error.

Finally, the scope of *t* ends on line 6. Since the compiler is able to guarantee *t* is the only variable using the underlying memory, the compiler can therefore safely insert the memory cleanup code for the heap object on line 6.

Having ownership moving makes great memory guarantees within a function, but is annoying when calling another function as seen in Listing 2. The

```
1   fn main() {
2       let s = String::from("string");
3       take_ownership(s);
4
5       println!("String len is {}", s.len()); // borrow of moved value: `s`
6   }
7
8   fn take_ownership(a: String) {
9       // some code working on a
10  } // Compiler will add memory clean up code for a here
```

Listing 2: Function taking ownership

take_ownership function takes ownership of the memory. When *main* calls *take_ownership* the compiler registers a transfer of the memory owner, thus making the call on line 5 invalid. On the upside, the cleanup code is correctly inserted on line 10.

```
1   fn main() {
2       let s = String::from("string");
3       take_borrow(&s);
4
5       println!("String len is {}", s.len());
6   } // Compiler will add memory clean up code for s here
7
8   fn take_borrow(a: &String) {
9       a.push_str("suffix"); // cannot borrow a as mutable
10  }
```

Listing 3: Function taking borrow

The solution is the second form of ownership, borrowing. As seen in Listing 3, borrow makes the function *take_borrow* only take temporary control of the memory. Once *take_borrow* has ended, control goes back to *main* - where the cleanup code will be inserted. Borrowing is activated with an ampersand (*&*) before the type and in the call.

But in Rust, all variable are immutable by default. So trying to change the underlying memory in *take_borrow* causes an error stating the borrow is not mutable on line 9.

Being unable to change the variable naturally leads to the third and last ownership form, mutable borrows. As seen in Listing 4, mutable borrows are activated using *&mut* on the type and call. Mutable ownership one the other hand just need *mut* when the variable is declared.

The three ownership forms - moving, borrowing and mutable borrowing -

```rust
fn main() {
   let mut s = String::from("string");
   take_borrow(&mut s);

   println!("String len is {}", s.len());
} // Compiler will add memory clean up code for s here

fn take_borrow(a: &mut String) {
   a.push_str("suffix");
}
```

Listing 4: Function taking mutable borrow

put some constraints on the code of each variable type and their calls:

- Moving will always invalidate the variable.
- Borrowed variables cannot be mutated. But more than one function can borrow the memory at the same time.
- Mutable borrowing does allow mutations. But only one function can hold a mutable borrow at a time and no other immutable borrows can exist.

The constraints will always be enforced by the compiler, thus requiring all code to meet them.

5 Reporting

6 Conclusion

References