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 have low level like performance [KN19]. Traditionally, languages that are memory safe will make use of a garbage collector which slows performance [HB05]. Garbage collector languages include C# [RNW+04], Java [GJS96], Python [Mar06], Golang [Tso18] and Javascript [Fla06]. Languages that perform well use manual memory management which is not memory safe when the programmer is not carefull. Dangling pointers [CGMN12], memory leaks [Wil92] and double freeing [Sha13] in languages like C and C++ are prime examples of manual memory management problems [Kok18]. To achieve both memory safety and performance, Rust uses a less popular model known as ownership.

4.1 Ownership

In the ownership model the compiler uses statical analysis [RL19] to track which variable owns a piece of heap data – this does not apply to stack data. Each data piece can only be owned by one variable at a time. The owning variable is called the *owner*. [KN19]

A variable also has a scope. The scope start at the variable declaration and ends at the closing curly bracket for the code block the variable is in. When the owner goes out of scope, Rust returns the memory by calling the *drop* method at the end of the scope. Ownership is manifested in two forms – moving and borrowing. These two forms are explained next. [KN19]

4.1.1 Moving

Moving happens when one variable is assigned to another. The compiler's analysis moves ownership of the data to the new variable from the initial variable. The initial variable is also invalidated at this point [KN19]. An analogy example would be to give a book to a friend. The friend can do anything from anotating to burning the book as they feel fit.

In listing 1 on the following page, on line 2, a heap data object is created and assigned to variable *s*. Line 3 assigns *s* to *t*. But, because *s* is a heap object, the compiler transfers ownership of the data from *s* to *t* and marks *s* as invalid.

When trying to use the data 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.

```
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 'drop' t here
```

Listing 1: Example of ownership transfer

Finally, the scope of t ends on line 6. Since the compiler is able to guarantee t is the only variable owning the data, the compiler can free the memory on line 6.

```
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 'drop' a here
```

Listing 2: Function taking ownership

Having ownership moving makes great memory guarantees within a function, but is annoying when calling another function as seen in listing 2. The *take_ownership* function takes ownership of the heap data which results in memory cleanup code correctly being inserted at the end of its scope on line 10. When *main* calls *take_ownership* the a becomes the new owner of s's data, thus making the call on line 5 invalid. When taking ownership like this is not desired the second form of ownership, borrowing, should be used instead.

4.1.2 Borrowing

Borrowing has a new variable take a references to data rather than becoming its new owner [KN19]. It is analogous to borrowing a book from a friend where a promise is made that the book will be returned to its owner once done with it.

As seen in listing 3 on the following page, borrowing makes the function *take_borrow* take a reference to the data. References are activated with an

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

    println!("String len is {}", s.len());
} // Compiler will 'drop' s here

fn take_borrow(a: &String) {
    a.push_str("suffix"); // cannot borrow *a as mutable
} // a is borrowed and wiil therefore not be dropped
```

Listing 3: Function taking borrow

ampersand (*&*) before the type. Once *take_borrow* has ended, control goes back to *main* - where the cleanup code will be inserted. Having references as function parameters is called borrowing [KN19]. The ampersand is also used in the call argument to signal that the called function will borrow the data.

But in Rust, all variable are immutable by default. So trying to change the data in *take_borrow* causes an error stating the borrow is not mutable on line 9. This is much the same as the borrowed book. One would not make highlights and notes in a book that is borrowed, unless if explicit permission was given to do so by the owner.

4.2 Immutable by default

Mutable borrows are an explixit indication that a function / variable is allowed to change the data.

```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

As seen in listing 4, mutable borrows are activated using *&mut* on the type. Again, it is also used in the call to make it explicit that the function will modify

the data. Variables – on the stack or heap – also need to be declared *mut* to be able to use them as mutable. [KN19]

The two ownership forms - moving and borrowing - together with mutables put some constraints on the code of each variable type and their calls: [KN19]

- Moving will always invalidate the variable.
- Borrowed variables cannot be mutated. But more than one function can borrow the data at the same time in parallel and concurrent code.
- 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.

4.3 Not OOP

4.3.1 Structs

4.3.2 Associate methods

4.3.3 Composition over inheritance

5 Reporting

6 Conclusion

References

[CGMN12] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, page 133âĂŞ143, New York, NY, USA, 2012. Association for Computing Machinery.

[Fla06] David Flanagan. *JavaScript: The Definitive Guide*, chapter 11, pages 171–172. OâĂŹReilly Media, Inc., 2006.

[GJS96] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*, chapter 12, page 245. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edition, 1996.

[HB05] Matthew Hertz and Emery D. Berger. Quantifying the performance of garbage collection vs. explicit memory management. *SIGPLAN Not.*, 40(10):313âĂŞ326, October 2005.

[KN19] Steve Klabnik and Carol Nichols. *The Rust Programming Language (Covers Rust 2018)*. No Starch Press, 2019.

[Kok18] Konrad Kokosa. *Pro .NET Memory Management: For Better Code, Performance, and Scalability*, chapter 1, pages 28–34. Apress, USA, 1st edition, 2018.

[Mar06] Alex Martelli. *Python in a Nutshell (In a Nutshell (OâĂŹReilly))*, chapter 13, pages 269–272. OâĂŹReilly Media, Inc., 2006.

[RL19] Morten Meyer Rasmussen and Nikolaj Lepka. Investigating the benefits of ownership in static information flow security. 2019.

[RNW+04] Simon Robinson, Christian Nagel, Karli Watson, Jay Glynn, and Morgan Skinner. *Professional C#*, chapter 7, pages 192–193. John Wiley & Sons, Inc., Hoboken, NJ, USA, 2004.

[Sha13] John Sharp. *Microsoft Visual C# 2013 Step by Step*, chapter 14, pages 320–321. Microsoft Press, USA, 1st edition, 2013.

[Tso18] Mihalis Tsoukalos. *Mastering Go: Create Golang production applications using network libraries, concurrency, and advanced Go data structures*, chapter 2, pages 47–49. Packt Publishing Ltd, 2018.

[Wil92] Paul R Wilson. Uniprocessor garbage collection techniques. In *International Workshop on Memory Management*, pages 1–42. Springer, 1992.