OwnershipTypesRust

Seminararbeit zum Thema Ownership Types in Rust

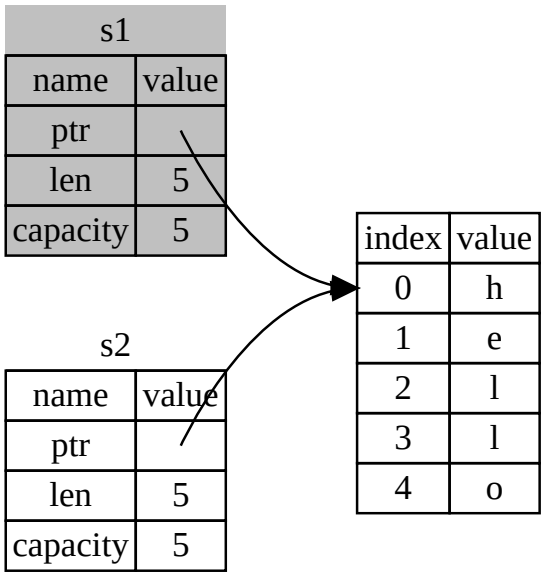
Ownership in Rust

Rust nutzt das Prinzip der Ownership von Objekten zur Speicherverwaltung. Insbesondere um die Verwaltung des auf dem Heap angelegten Speichers. Ähnlich wie Sprachen wie C und C++ besitzt Rust keinen Garbage Collector, dass bedeutet angelegter Speicher muss manuell freigegeben werden. Historisch zeigte sich jedoch das die selbständige Speicherverwaltung zu einigen Problemen führen kann. So muss angelegter Speicher genau einmal freigegeben werden. Um dies in Rust zu garantieren wird das Prinzip der Ownership verwendet. Ein Pointer der auf ein Objekt zeigt und somit seine Speicheraddresse kennt ist der Owner dieses Objekts, sobald dieser Pointer out-of-scope geht wird der verwiesene Speicherbereich automatisch freigegeben. Sollte nun eine zweite Variable angelegt werden, die das selbe Objekt referenziert, so ist dieser Pointer der neue Besitzer des Objekts. Dies hat zur Folge, dass die zuerst Angelegte Varible nicht länger auf das Objekt zugreifen kann um ein doppeltes freigeben des gleichen Speicherbereichs ausschließen zu können.

Beispiele

Move Semantik

In der folgenden Grafik ist zu erkennen das Pointer s1 ungültig wird, sobald Pointer s2 auf das selbe Objekt



verweist.

Quelle: https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html

Simples Beispiel

```rust
fn main(){
    let x = 5; // x = 5
    let y = x; // y = 5
    // Speicher im Stack wird kopiert
    println!("x:{} y:{}", x, y);

    // s1 ist Pointer zu in Heap angelegtem String Object
    let s1 = String::from("String1"); // s1 -> "String1"
    println!("s1 -> {}", s1);


    let s2 = s1; // s2 -> "String1" s1 ungültig
    // Ownership des Heap Speicherbereichs wurde auf s2 übertragen
    /*
    Vergleich C++:
    string* s1 =  new string("String1");
    string* s2 = s1;
    s1 = nullptr;
     */

    let s2_clone = s2.clone(); // Kopie des Speicherbereichs von "Hello World!" wird angelegt
    println!("s2 -> {}, s2_clone -> {}", s2, s2_clone);

    let s3 = s2;
    // println!("s2-> {}", s2); // Funktioniert nicht da s2 nicht mehr Owner ist

} // Stack wird freigegeben. Speicher auf den s2 und s3 zeigen wird freigegeben
```

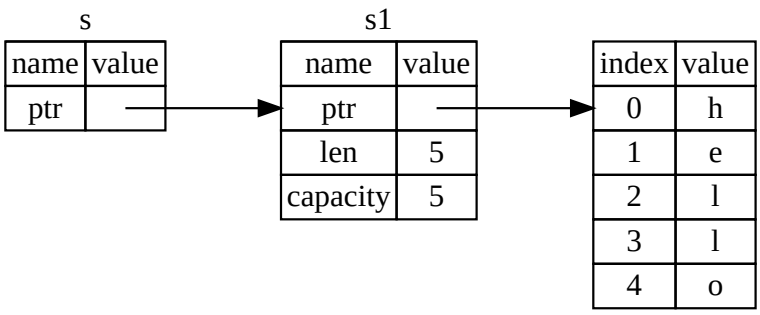
Speicher im Heap besitzt immer einen sogenannten Besitzer, in der Regel ist dieser ein Pointer auf das entsprechende Objekt (vgl. Zeile 8). Sobald ein weiterer Pointer auf das Objekt zeigt ist dieser der neue Besitzer des Objekts und der ursprüngliche Pointer ist nicht mehr gültig (vgl. Zeile 12). Falls das Objekt kopiert werden soll so muss dies explizit angegeben werden (vgl. Zeile 21).

Beispiel mit Methoden

```rust
fn main(){
    // s1 ist Pointer zu in Heap angelegtem String Object
    let s1 = String::from("String1"); // s1 -> "String1"
    println!("s1 -> {}", s1);


    let s2 = s1; // s2 -> "String1" s1 ungültig
    // Ownership des Heap Speicherbereichs wurde auf s2 übertragen

    let s2_clone = s2.clone();
    take_ownership(s2); // Ownership wird an Methode übertragen
    //println!("s2-> {}", s2); // Funktioniert nicht da s2 nicht mehr Owner ist

    let s3 = take_ownership_and_give_back(s2_clone);
    println!("s3 -> {}", s3);

} // Stack wird freigegeben. Speicher auf den s2 und s3 zeigen wird freigegeben

fn take_ownership(string: String){ //
    println!("take_ownership string -> {}", string);
} // string Variable out of scope --> Speicher wird freigegeben

fn take_ownership_and_give_back(string: String) -> String {
    println!("take_ownership_and_give_back string -> {}", string);
    string // Return Wert gibt Ownership zurück
}
```

Sollten solche Objekte an Methoden übergeben werden, so ist der neue Besitzer des Objekts ebenfalls die Methode (vgl. Zeile 11). Nur falls die Methode das Objekt beziehungsweise den Pointer auf das Objekt wieder zurückgibt, wird das Objekt nicht aus dem Speicher entfernt (vgl. Zeile 14).

Borrow Semantik




```rust
fn main(){
    let s1 = String::from("hello");
    let len = calculate_length(&s1);
```

```
} // s1 goes out of scope and the String object is deleted

fn calculate_length(s: &String) -> usize {
    s.len()
} // s goes out of scope but since it has no ownership the object is not
deleted
```

Quelle: https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html

Wie in dem Beispiel zu sehen ist verweist bei die Referenz bei der Borrow-Semantik lediglich auf den eigentlichen Besitzer des Objekts

nicht veränderbare Referenz

```rust
fn main(){
    // s1 ist Pointer zu in Heap angelegtem String Object
    let s1 = String::from("String1"); // s1 -> "String1"
    println!("s1 -> {}", s1);

    borrow_value(&s1);

    println!("s1 after method call -> {}", s1);

    let s2 = &s1; // s2 ist NICHT Besitzer des Objekts, verweist aber darauf
    println!("s1 -> {}, s2 -> {}", s1, s2);
} // Stack wird freigegeben. Speicher auf den s2 und s3 zeigen wird freigegeben

fn borrow_value(string: &String){
    println!("borrowed value -> {}", string);
} // Referenz wird übergeben --> Speicher wird nicht freigegeben
```

Mithilfe der Borrow Semantik ist es auch in Rust möglich mehere Pointer auf das selbe Objekt verweisen zu lassen. Dies ist wie im obigen Code Beispiel zu sehen sowohl mit neuen Variblen als auch bei der Übergabe an Methoden möglich.

veränderbare Referenz

```rust
fn main(){
    let mut s = String::from("String 1");

    let r1 = &mut s;
    change(r1);
    printout(r1);

    let r2 = &mut s;
    change(r2);
    printout(r2);

    // printout(r1); // doesn't work because of second mutable borrow
}

fn change(string: &mut String){
    string.push_str(", added something to String");
}

fn printout(string: &String){
    println!("string -> {}", string);
}
```

Um veränderbare Referenzen zu erzeugen wird das Schlüsselwort "mut" verwendet, welches kurz für mutable (deutsch: veränderbar) ist. Pro Owner kann jedoch immer nur eine einzige solche veränderbare Referenz existieren.

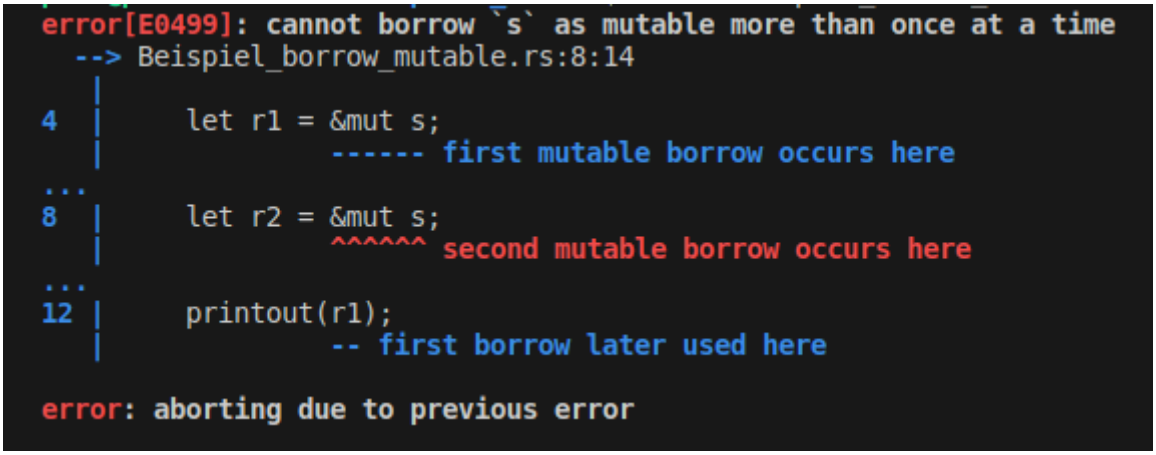
Error Meldungen

Zugriff auf ungültigen Pointer

```
error[E0382]: borrow of moved value: `s2`
  --> Beispiel1.rs:25:25
   |
12 |     let s2 = s1; // s2 -> "String1" s1 ungültig
   |         -- move occurs because `s2` has type `String`, which does not implement the `Copy` trait
...
24 |     take_ownership(s2); // Ownership wird an Methode übertragen
   |                    -- value moved here
25 |     println!("s2-> {}", s2); // Funktioniert nicht da s2 nicht mehr Owner ist
   |                          ^^ value borrowed here after move
   |
note: consider changing this parameter type in function `take_ownership` to borrow instead if owning the value isn't ne
cessary
  --> Beispiel1.rs:32:27
   |
32 | fn take_ownership(string: String){ //
   |    --------------         ^^^^^^ this parameter takes ownership of the value
   |    |
   |    in this function
   = note: this error originates in the macro `$crate::format_args_nl` which comes from the expansion of the macro `pri
ntln` (in Nightly builds, run with -Z macro-backtrace for more info)
help: consider cloning the value if the performance cost is acceptable
   |
24 |     take_ownership(s2.clone()); // Ownership wird an Methode übertragen
   |                    ++++++++

error: aborting due to previous error
```

Der Rust Compiler erkennt selbständig, dass hier versucht wird auf einen Pointer zuzugreifen, der aufgrund der Move-Semantik nicht mehr gültig ist. Es wird angegeben, an welcher Stelle im Quellcode der Pointer ungültig wird, außerdem wird ein möglicher Lösungvorschlag zurückgegeben.

Doppelte Mutable Referenz



In Rust ist immer nur maximal eine veränderbare Referenz auf ein Objekt zugelassen, um sogenannte Race-Conditions zu vermeiden. Folglich führt ein Zugriff auf die Variable "r1" nachdem eine zweite mutable reference angelegt wurde zu einem Fehler.

Die Grenzen des Rust compilers

Unerreichbarer Code

Der Folgende Code kompiliert nicht, da der Rust-Compiler nicht selbständig erkennt, dass ein Teil des Quellcodes niemals ausgeführt wird. Der Rust-Compiler untersucht den Code lediglich Zeile für Zeile und meldet deshalb einen Speicherfehler, der zur Laufzeit jedoch nie auftreten wird.

```rust
fn main(){
    let mut s = String::from("String 1");

    let r1 = &mut s;
    change(r1);
    printout(r1);

    if false{
        let r2 = &mut s;
        change(r2);
        printout(r2);
    }


    printout(r1); // doesn't work because of second mutable borrow
}
```

Stack Overflow durch zyklische Referenzen

Eine Möglichkeit einen Speicherfehler in Rust zu erzeugen, ist das verwenden zyklischer Referenzen. In dem Beispiel reference_cycles.rs ist Code zu finden der mithilfe zyklischer Referenzen einen Stack-Overflow zur Laufzeit verursacht. In dem Beispiel werden zwei Nodes einer linked-List erzeugt, die auf die jeweils andere verweisen. Gibt man sich nun den Nachfolger des ersten Elements aus, so verweisen die beiden Objekte solange gegenseitig aufeinander, bis der Stack voll ist und das Programm abstürzt.

Speicher Manuell Freigeben

Auch in Rust ist es möglich Speicherbereiche manuell freizugeben. Im Beispiel manual_free.rs ist Rust Code zu finden, bei dem versucht wird auf ein Element eines Vektors zuzugreifen, welches bereits manuell gelöscht wurde. Dieses Problem wird zur Compile-Zeit nicht vom Rust-Compiler erkannt, führt jedoch beim ausführen des Programms zum Absturz.