Funktionale Programmierung in Rust

Gliederung

- Wiederholung Ownership und Borrowing
- Scoping
- Closures
- Eager und Lazy Evaluation
- Lifetimes
- Code
- Tests
- Nachtrag zu Box<T>

Wiederholung Ownership

- Jeder Wert in Rust hat einen Eigentümer (owner).
- Es kann immer nur einen Eigentümer zur gleichen Zeit geben.
- Wenn der Eigentümer den Gültigkeitsbereich verlässt, wird der Wert aufgeräumt.
- Speichersicherheit ohne Garbage Collector

```rust
fn main() {
    let s = String::from("hello");  // s comes into scope

    takes_ownership(s);             // s's value moves into the function...
                                    // ... and so is no longer valid here

    let x = 5;                      // x comes into scope

    makes_copy(x);                  // because i32 implements the Copy trait,
                                    // x does NOT move into the function,
    println!("{}", x);              // so it's okay to use x afterward

} // Here, x goes out of scope, then s. But because s's value was moved, nothing
  // special happens.

fn takes_ownership(some_string: String) { // some_string comes into scope
    println!("{some_string}");
} // Here, some_string goes out of scope and `drop` is called. The backing
  // memory is freed.

fn makes_copy(some_integer: i32) { // some_integer comes into scope
    println!("{some_integer}");
} // Here, some_integer goes out of scope. Nothing special happens.

```

Wiederholung Borrowing

- Anstatt Ownership wird die Referenz auf die Adresse übergeben

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

    let len = calculate_length(&s1);

    println!("The length of '{s1}' is {len}.");
}

fn calculate_length(s: &String) -> usize {
    s.len()
}
```

Shadowing

```rust
fn main() {
    let x = 5;
    println!("x vor Shadowing: {}", x); // Ausgabe: x vor Shadowing: 5

    {
        let x = x * 2; // Shadowing innerhalb eines Blocks
        println!("x im Block: {}", x); // Ausgabe: x im Block: 12
    }

    println!("x nach Block: {}", x); // Ausgabe: x nach Block: 6
}

```

Closures

- Anonyme Funktionen
- Können auf Werte im Gültigkeitsbereich zugreifen
- &T (immutable borrow)
- &mut T (mutable borrow)
- move (Übernahme von Ownership)

```
38      //Mut Closure
39      let mut counter = 0;
40      let mut plus_counter = |&x: &i32| {
41          counter += 1;
42          x + counter
43      };
```

```
27      //Closures
28      let square = |&x: &i32| x * x;
29      let plus_one = |&x: &i32| x + 1;
30      let is_even = |x: &i32| x % 2 == 0;
31      let multiply = |x: &i32| x * 3;
32
```

Closure Traits

- FnOnce
- FnMut
- Fn

```rust
fn filter<F, D>(&self, f: F, target: D) -> D
where
    F: Fn(&T) -> bool,
    D: Datastructure<T>,
{
    let mut new_target = target;
    for item in &self.data {
        if f(item) {
            new_target.insert(item.clone());
        }
    }
    new_target
}
```

```rust
fn map<U, F, D>(&self, mut f: F, target: D) -> D
where
    F: FnMut(&T) -> U,
    D: Datastructure<U>,
{
    let mut new_target = target;
    for item in &self.data {
        let transformed = f(item);
        new_target.insert(transformed);
    }
    new_target
}
```

Lazy und Eager Evaluation

- Iteratoren sind lazy
- Objekt, dass eine Sequenz von Elementen durchläuft
- Interner Zustand speichert nächstes Element
- Iteratoren implementieren Iterator-Trait
- type Item definiert einen generischen Platzhalter
- next() gibt ein Option<Self::Item> zurück
- Sparen Rechenleistung und Speicher

```rust
pub trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;
}
```

Lifetimes in Rust

- Prüfen die Gültigkeit von Referenzen zur Kompilierzeit
- Referenz soll nicht auf freigegebene Daten zeigen
- StackIter darf keine längere Lifetime als der Stack haben

```
141    pub struct StackIter<'a, T> {
142        stack: &'a Stack<T>,
143        index: usize,
144    }
```

Nachtrag: Box<T>

- Größe von Typen muss zur Compile-Zeit bekannt sein
- Next enthält eine neue Node
- Box<T> als Pointer für Heap allokierte Node
- Box<T> besitzt vordefinierte Größe
- Größe kann nicht unendlich groß werden
- Ermöglicht Implementierung von rekursiven Strukturen

Quellen

- https://doc.rust-lang.org/book/
- https://web.mit.edu/rust-lang_v1.25/arch/amd64_ubuntu1404/share/doc/rust/html/book/first-edition/closures.html
- https://doc.rust-lang.org/rust-by-example/
- https://doc.rust-lang.org/book/ch13-00-functional-features.html
- https://web.mit.edu/rust-lang_v1.25/arch/amd64_ubuntu1404/share/doc/rust/html/book/second-edition/ch15-01-box.html