Systemnahe Programmierung in Rust

- "The Book" / Pattern Matching / Kap. 18 -

Hubert Högl

Hochschule Augsburg / Informatik

2022-12-12 17:42:42

Pattern Matching

Kap. 18.1: https://rust-lang-de.github.io/rustbook-de/ch18-01-all-the-places-for-patterns.html

Kombinationen aus

- Literale
- Destrukturierte Arrays, Aufzählungen (enums), Strukturen (structs) oder Tupel
- Variablen
- Wildcards
- Platzhalter

Können verwendet werden in

- `match` Zweigen (Prüfung auf Vollständigkeit bei `match`, nicht bei `if let`)
- `if let`, `else if`, `else if let`, `else`
- `while let`
- `for` (for *pattern* in *sequence*)

siehe nächste Seite ...

Pattern Matching (2)

Fortsetzung ...

- `let`

```
let x = 5;  // let PATTERN = EXPRESSION;
let (x, y, z) = (1, 2, 3);
```

- Einen oder mehrere Werte ignorieren: `_`, `..`
- Funktionsparameter bzw. Closure-Parameter

```
fn foo(x: i32) { ... }  // Muster `x`
n print_coordinates(&(x, y): &(i32, i32)) { ... }  // Muster &(x, y)

let point = (3, 5);
print_coordinates(&point);
```

Abweisbarkeit (18.2)

Unabweisbare Muster

- passen für jeden möglichen übergebenen Wert
- Bsp: `let x = 5;`
- `let`, `for`, Funktionsparameter

Abweisbare Muster

- passen für einen möglichen übergebenen Wert nicht
- Bsp: `if let Some(x) = value`, falls `value` None ist
- `if let`, `while let`

Geht nicht:

```
let Some(x) = some_value;  // let erwartet unabweisbares Muster

if let x = 5 { ... };   // if let erwartet abweisbares Muster
```

Mustersyntax (18.3)

Literale

```
let x = 1;

match x {
    1 => println!("eins"),
    2 => println!("zwei"),
    3 => println!("drei"),
    _ => println!("sonstige"),
}
```

Mustersyntax (2)

Benannte Variablen

```
let x = Some(5);
let y = 10;

match x {
    Some(50) => println!("Habe 50 erhalten"),
    Some(y) => println!("Passt, y = {y}"),  // Beschattung!
    _ => println!("Standardfall, x = {:?}", x),
}

println!("Am Ende: x = {:?}, y = {y}", x);
```

Mustersyntax (3)

Mehrfache Muster

```
#![allow(unused)]
fn main() {
    let x = 1;

    match x {
        1 | 2 => println!("eins oder zwei"),
        3 => println!("drei"),
        _ => println!("sonstige"),
    }
}
```

Mustersyntax (4)

Wertebereiche

```
let x = 5;

match x {
    1..=5 => println!("eins bis fünf"),
    _ => println!("etwas anderes"),
}
```

Geht auch z.B. mit `'a'..='z' => ...`

Mustersyntax (5)

Destrukturierung von Strukturen

```
let p = Point { x: 0, y: 7 };
let Point { x: a, y: b } = p;  // Kurzform x: x, y: y: let Point { x, y } = p;

match p {
    Point { x, y: 0 } => println!("Auf der x-Achse bei {}", x),
    Point { x: 0, y } => println!("Auf der y-Achse bei {}", y),
    Point { x, y } => println!("Auf keiner Achse: ({}, {})", x, y),
}
```

Destrukturieren von Aufzählungen

```
match msg {
    Message::Quit => { ... },
    Message::Move { x, y } => { ... },
    Message::Write(text) => { ... },
    Message::ChangeColor(r, g, b) => { ... },
}
```

Mustersyntax (6)

Destrukturieren verschachtelter Strukturen und Aufzählungen

```
let msg = Message::ChangeColor(Color::Hsv(0, 160, 255));
match msg {
    Message::ChangeColor(Color::Rgb(r, g, b)) => println!(..., r, g, b);
    ...
```

Destrukturieren von Strukturen und Tupeln

```
fn main() {
    struct Point {
        x: i32,
        y: i32,
    }
    let ((feet, inches), Point { x, y }) = ((3, 10), Point { x: 3, y: -10 });
}
```

Mustersyntax (7)

Ignorieren von Werten

- Ignorieren eines Gesamtwertes

```
fn foo(_: i32, y: i32) { ... }
```

- Ignorieren von Teilwerten

```
(Some(_), Some(_)) => { ... }
```

- Ignorieren von unbenutzten Variablen mit _, z.B. `_x` (bindet gültig, so dass z.B. ein move nach `_x` gemacht wird. Ein _ bindet nicht, es wird kein move in _ gemacht.

Mustersyntax (8)

- Ignorieren der verbleibenden Teile eines Wertes mit ...

```
let origin = Point { x: 0, y: 0, z: 0 };

match origin {
    Point { x, .. } => println!("x ist {}", x),
    (first, .., last) => { ... }, // let numbers = (2, 4, 8, 16, 32);
    (.., second, ..) => { ... }, // geht nicht - mehrdeutig!
}
```

Mustersyntax (9)

- Match Guard (`if x % 2 == 0`)

```
let num = Some(4);

match num {
    Some(x) if x % 2 == 0 => println!("Die Zahl {} ist gerade", x),
    Some(x) => println!("Die Zahl {} ist ungerade", x),
    None => (),
}
```

Compiler prüft dann nicht mehr die Vollständigkeit.

Löst auch Beschattung einer äusseren Variablen `y` bei `Some(y)`:

```
Some(n) if n == y => println!("Passt, n = {n}"),
```

Match Guard trifft auf 4, 5 und 6 zu!

```
4 | 5 | 6 if y => println!("ja"),
```

Mustersyntax (10)

at-Operator, @

```
enum Message {
    Hello { id: i32 },
}

let msg = Message::Hello { id: 5 };

match msg {
    Message::Hello {
        id: id_variable  3..=7,
    } => println!("id im Bereich gefunden: {}", id_variable),
    Message::Hello { id: 10..=12 } => {
        println!("id in einem anderen Bereich gefunden")
    }
    Message::Hello { id } => println!("Eine andere id gefunden: {}", id),
    }
```