Systemnahe Programmierung in Rust

- “The Book” / Generische Programmierung / Kap. 10 -

Hubert Högl

Technische Hochschule Augsburg / Informatik
https://tha.de/~hhoegl

2024-10-05 13:23:14

Generische Programmierung

→ https://doc.rust-lang.org/book/ch10-00-generics.html

Siehe Beispielcode in `material/Beispiele/bk10*.rs`

Funktion die generisch in `T` ist.

```
fn largest<T>(list: &[T]) -> &T {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}
```

Generische Struktur

Struktur generisch in `T`.

```
struct Point<T> {
    x: T,
    y: T,
}
```

Struktur generisch in `T` und `U`

```
struct Point<T, U> {
    x: T,
    y: U,
}

...

let integer_and_float = Point { x: 5, y: 4.0 };
```

Generischer Enum

```rust
enum Option<T> {
    Some(T),
    None,
}

enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Generischer Enum mit gen. Impl

```rust
struct Point<T> {
    x: T,
    y: T,
}

impl<T> Point<T> {
    fn x(&self) -> &T {
        &self.x
    }
}
```

Einschränkung auf f32

```rust
impl Point<f32> {
    fn distance_from_origin(&self) -> f32 {
        (self.x.powi(2) + self.y.powi(2)).sqrt()
    }
}
```

Zur Laufzeit schnell durch Codeduplizierung

`Option<T>` wird für jeden verwendeten Typ durch den Compiler automatisch dupliziert:

```
enum Option_i32 {
    Some(i32),
    None,
}

enum Option_f64 {
    Some(f64),
    None,
}

fn main() {
    let integer = Option_i32::Some(5);
    let float = Option_f64::Some(5.0);
}
```

Keine Laufzeitkosten (Prinzip: “Zero-cost abstractions”)

Traits (Merkmale)

- **Merkmale**

  Funktionalität, die ein bestimmter Typ hat und mit anderen Typen teilen kann.

  Compiler kann prüfen, ob konkrete Typen, die von unserem Code verwendet werden, das richtige Verhalten aufweisen. Das Verhalten wird durch Traits angegeben.

- **Regel**

  Merkmal kann für einen Typ nur dann implementiert werden, wenn entweder das Merkmal oder der Typ lokal im Crate vorhanden ist (Kohärenz, Waisenregel)

- **Merkmalsabgrenzung (trait bound)**

  Generischer Typ wird durch das Vorhandensein von bestimmten Traits festgelegt.

- **Trait Syntax**

```
pub trait Summary {
    fn summarize(&self) -> String;    // nur die Signatur!
}
```

Code (1/2)

```rust
pub struct NewsArticle {
    pub headline: String,
    pub location: String,
    pub author: String,
    pub content: String,
}

pub struct Tweet {
    pub username: String,
    pub content: String,
    pub reply: bool,
    pub retweet: bool,
}
```

Code (2/2)

```rust
impl Summary for NewsArticle {
    fn summarize(&self) -> String {
        format!("{}, von {} ({})", self.headline, self.author, self.location)
    }
}

impl Summary for Tweet {
    fn summarize(&self) -> String {
        format!("{}: {}", self.username, self.content)
    }
}

let tweet = Tweet {
    ...
    };

println!("1 neue Kurznachricht: {}", tweet.summarize());
```

Traits - Standardimplementierung (1)

Standardimplementierung eines Traits (nicht nur Signatur wie oben!)

```
pub trait Summary {
    fn summarize(&self) -> String {
        String::from("(Lies mehr ...)")
    }
}

impl Summary for NewsArticle {
    // leer
}
```

Traits - Standardimplementierung (2)

- Standard-Implementierung kann überschrieben werden
- Standard-Implementierungen können andere Methoden desselben Merkmals aufrufen, auch wenn diese keine Standard-Implementierung haben.
- Standard-Implementierung kann nicht aus der Implementierung derselben Methode aus aufgerufen werden.

Traits - Merkmalsabgrenzung (1)

Traits als Parameter

Statt einen Typ hat der Parameter `item` die Implementierung eines Trait.

Kurzschreibweise

```
pub fn notify(item: impl Summary) {
    println!("Eilmeldung! {}", item.summarize());
}
```

Wo liegt der Unterschied zu `pub fn notify(item: &impl Summary) {`?

Langschreibweise (“trait-bound” Syntax, Merkmalsabgrenzung)

```
pub fn notify<T: Summary>(item: &T) {
    println!("Eilmeldung! {}", item.summarize());
}
```

Traits - Merkmalsabgrenzung (2)

Mehrere Merkmalsabgrenzungen

```
pub fn notify(item: &(impl Summary + Display)) { ... }
```

```
pub fn notify<T: Summary + Display>(item: &T) { ... }
```

Alternative Syntax mit `where`

```
fn some_function<T: Display + Clone, U: Clone + Debug>(t: &T, u: &U) -> i32 {
        ... }
```

```
fn some_function<T, U>(t: &T, u: &U) -> i32
    where T: Display + Clone,
          U: Clone + Debug
{ ... }
```

Traits - Rückgabetyp implementiert Merkmal

```
fn returns_summarizable() -> impl Summary { ... }
```

- Es darf nur *ein* Typ zurückgegeben werden (im Beispiel `Tweet` oder `NewsArticle`). Wenn man unterschiedliche Typen zurückgeben will muss man “Trait Objects verwenden (The Book, Kap. 17).
- Nützlich auch bei Funktionen die Closures oder Iteratoren zurückgeben, da konkreter Typ nur dem Compiler bekannt.

Traits - Bedingte Implementierung von Methoden (1)

```rust
use std::fmt::Display;
struct Pair<T> {
    x: T,
    y: T,
}
impl<T> Pair<T> {
    fn new(x: T, y: T) -> Self {
        Self { x, y }
    }
}
impl<T: Display + PartialOrd> Pair<T> {
    fn cmp_display(&self) {
        if self.x >= self.y {
            println!("Das größte Element ist x = {}", self.x);
        } else {
            println!("Das größte Element ist y = {}", self.y);
        }
    }
}
```

Traits - Bedingte Implementierung von Methoden (2)

- Implementierungen eines Merkmals für Typen, die Merkmalsabgrenzungen erfüllen, werden als Pauschal-Implementierungen (*blanket implementations*) bezeichnet. Kommt häufig in der Rust Standardbibliothek vor.

  Beispiel: Merkmal `ToString` wird für jeden Typ implementiert, der das Merkmal `Display` hat.

```
impl<T: Display> ToString for T {
    ...
}

...

let s = 3.to_string();
```

Lebensdauer (Kap. 10.3)

- Siehe auch Kap. 4 (Referenzen und Ausleihen)
- Jede Referenz hat eine Lebensdauer (lifetime)
- Ausleihenprüfer (borrow checker)
- Lebensdauer ist ein generischer Typ

```
&i32        // eine Referenz
&'a i32     // eine Referenz mit expliziter Lebensdauer
&'a mut i32 // eine veränderliche Referenz mit expliziter Lebensdauer
```

Funktionssignatur mit Lebensdauern (verbindet die Lebensdauern miteinander)

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Lebensdauer (2)

```rust
fn main() {
    let string1 = String::from("lange Zeichenkette ist lang");

    {
        let string2 = String::from("xyz");
        let result = longest(string1.as_str(), string2.as_str());
        println!("Die längere Zeichenkette ist {}", result);
    }
}

fn longest<'a>(x: &'a str, y: &str) -> &'a str {
    x
}
```

Lebensdauer (3)

Strukturen mit Referenzen brauchen Lebensdauern

Beispiel: `ImportantExcert` kann nicht laenger leben als die Referenz `part`.

```
struct ImportantExcerpt<'a> {
    part: &'a str,
}

fn main() {
    let novel = String::from("Nennen Sie mich Ishmael. Vor einigen Jahren ...");
    let first_sentence = novel.split('.').next().expect("Konnte keinen '.' finden.");
    let i = ImportantExcerpt {
        part: first_sentence,
    };
}
```

Lebensdauer (4)

Lebensdauer-Elision (lifetime elision): Der Compiler kennt Faelle, fuer die man keine Lebensdauer angeben muss.

Eingabelebensdauern fuer Referenzen (input lifetimes)

Ausgabelebensdauern fuer Referenzen (output lifetimes)

Drei Elisions-Regeln :

1. (input) Jede Referenz bekommt eine Lebensdauer
2. (input) Bei genau einem input bekommt der output die gleiche Lebensdauer
3. (output) Bei mehrere inputs von denen einer &self oder &mut self ist (Methode), bekommt der output die Lebensdauer von self.

Lebensdauer (5)

Methoden auf einer Struktur mit Lebensdauern

```
impl<'a> ImportantExcerpt<'a> {
    fn level(&self) -> i32 {   // Regel 1
        3
    }
}

impl<'a> ImportantExcerpt<'a> {
    fn announce_and_return_part(&self, announcement: &str) -> &str {  // Regeln 1 und 3
        println!("Bitte um Aufmerksamkeit: {}", announcement);
        self.part
    }
}
```

Lebensdauer (6)

Statische Lebenszeit

```
let s: &'static str = "Ich habe eine statische Lebensdauer.";
```

Lebensdauer (7)

Generische Typen, Merkmalsabgrenzungen und Lebensdauern

```rust
use std::fmt::Display;

fn longest_with_an_announcement<'a, T>(
    x: &'a str,
    y: &'a str,
    ann: T,
) -> &'a str
where
    T: Display,
{
    println!("Bekanntmachung! {}", ann); // Display
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```