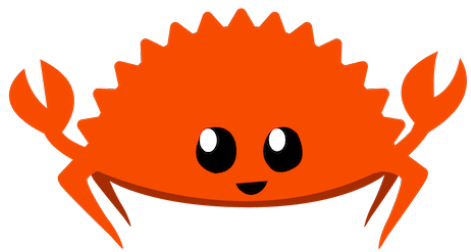
Programmieren mit *Rust*

Thema 6: Generische Typen, Traits und Lebensdauern

Dirk Müller und Robert Baumgartl

2. Mai 2024



Übersicht

1. Generische Typen:
   - Syntax
   - ~ in Strukturen und Aufzählungstypen
   - ~ in Methodendefinitionen
2. Traits: Gemeinsames Verhalten von Typen
   - Syntax, Definition, Implementierung
   - Traits als Parameter; *Trait Bounds*
3. Lebensdauern: TODO

Generische Typen

Grundidee:

- Parameter einer Funktion können viele verschiedene Werte annehmen; der Parameter abstrahiert vom konkreten Wert
- Generische Typen können verschiedene Typen annehmen; das Generic abstrahiert vom konkreten Typ

Generische Typen

```rust
fn last_elem_i32(list: &[i32]) -> i32 {
    let mut last = list[0];
    for item in list {
        last = *item;
    }
    last
}
fn last_elem_f64(list: &[f64]) -> f64 {
    let mut last = list[0];
    for item in list {
        last = *item;
    }
    last
}

fn main() {
    let integer_list = vec![34, 50, 25, 100, 65];
    let result1 = last_elem_i32(&integer_list);
    println!("The last element of integer_list is {}", result1);

    let fp_list = vec![3.14, 2.71, 1.89, 0.0005, 17.3*10E-34, 8.8];
    let result2 = last_elem_f64(&fp_list);
    println!("The last element of fp_list is {}", result2);
}
```

- Funktionen zum Bestimmen des letzten Elementes
- pro Typ eine Funktion nötig

Generische Typen

```rust
fn last_elem<T>(list: &[T]) -> &T {
    let mut last = &list[0];
    for item in list {
        last = item;
    }
    last
}

fn main() {
    let integer_list = vec![34, 50, 25, 100, 65];
    let result1 = last_elem(&integer_list);
    println!("The last element of integer_list is {}", result1);

    let fp_list = vec![3.14, 2.71, 1.89, 0.0005, 17.3*10E-34, 8.8];
    let result2 = last_elem(&fp_list);
    println!("The last element of fp_list is {}", result2);
}
```

- Benennung des Parameters nach Funktionsname (in <>) beliebig; Konvention: `T`

Generische Typen

Das funktioniert vorerst nicht . . .

```
fn sum<T> (x: T, y: T) -> T {
    x+y
}

fn main() {
    let (a,b) = (23,42);
    let (c,d) = (3.1415, 2.7182);
    println!("The sum of {a} and {b} is {}.", sum(a,b));
    println!("The sum of {c} and {d} is {}.", sum(c,d));
}
```

```
  Compiling generic-bsp2 v0.1.0 (/home/robge/src/generic-bsp2)
error[E0369]: cannot add `T` to `T`
 --> src/main.rs:2:6
  |
2 |     x+y
  |     -^- T
```

- Operation '+' nicht über allen denkbaren Typen definiert

Generische Typen in Strukturen

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

fn main() {
    let p1_int = Point { x: 5, y: 10, z: 0 };
    let p2_fp = Point { x: 1.0, y: 4.0, z: -3.14 };
    let p3_wrong = Point { x: 3, y: 4.0, z: 0.0 };
}
```

- Definition von `Point` erfordert beliebigen, aber gleichen Typ

Mehrere generische Typen in Strukturen

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

fn main() {
    let p1_int = Point { x: 5, y: 10, z: 0 };
    let p2_fp = Point { x: 1.0, y: 4.0, z: -3.14 };
    let p3_not_wrong_anymore = Point { x: 3, y: 4.0, z: 0.0
    ↪  };
}
```

- jede Komponente von `Point` hat eigenen generischen Typ
- beliebig viele generische Typen erlaubt (aber nicht empfohlen)

Generische Typen in Aufzählungstypen

Beispiele

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

Abstraktion eines optionalen Wertes:

- entweder ein (beliebiger) Wert vom (beliebigen) Typ `T`,
- oder nichts

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

Abstraktion eines Resultats (z. B. einer Funktion)

- liefert entweder `Ok`, das einen (gültigen) Wert des (beliebigen) Typs `T` enthält,
- oder (im Fehlerfall) `Err`, das einen Wert vom (beliebigen) Typ `E` zurückliefert (Fehlermeldung, ~code, . . .).

Generische Typen in Methodendefinitionen

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

impl<T> Point<T> {
    fn get_x(&self) -> &T {
        &self.x
    }
    fn get_y(&self) -> &T {
        &self.y
    }
}

fn main() {
    let pt = Point { x: 5, y: 10, z: 0 };
    println!("pt has {} as x and {} as y coordinate",
    ↪  pt.get_x(), pt.get_y());
}
```

Generische Typen in Methodendefinitionen

Anmerkungen

- `<T>` sofort nach **`impl`**-Schlüsselwort als *generic* deklarieren, damit klar ist, dass Methoden für den Typ `Point<T>` definiert werden sollen
- durch Angabe konkreter Typen nach **`impl`** und Namen des **`struct`** kann man Implementierungen für Methoden angeben, die nur mit diesen konkreten Typen arbeiten

Typbeschränkungen bei Generics

Beispiel

```
struct Point<T> {
    x: T,
    y: T,
    z: T,
}
impl<T> Point<T> {
    fn get_x(&self) -> &T {
        &self.x
    }
}
// works only for Point with FP values
impl Point<f32> {
    fn distance_from_origin(&self) -> f32 {
        (self.x*self.x + self.y*self.y + self.z*self.z).sqrt()
    }
}

fn main() {
    let pt = Point { x: 5.0, y: 10.0, z: 0.0 };
    println!("pt is {} units from origin",
    ↪  pt.distance_from_origin());
}
```

Traits

- "(charakteristische) Merkmale, Eigenschaften"

  “A trait defines functionality a particular type has and can share with other types. We can use traits to define shared behavior in an abstract way.” (Klabnik, Kapitel 10.2.)

Definition von Traits

```
pub trait Points3D {
  fn to_origin(&self) -> f64;
  fn output(&self) -> String;
}
```

- Name des Traits: `Summary`
- `pub`: abhängende Crates können diesen Trait verwenden
- innerhalb `{}` werden alle Methodensignaturen aufgeführt, die für diesen Trait notwendig sind
- mehrere Methoden sind möglich
- $\rightsquigarrow$ jeder Typ, der diesen Trait implementiert, muss Körper für diese Methoden implementieren

Implementierung eines Traits für einen Typ

- ähnlich der Implementierung „regulärer“ Methoden
- `impl traitname for typname`, danach die Methodensignaturen mit Körper in `{}`
- vor Nutzung ggf. noch Trait und Typen in Gültigkeitsbereich einbringen
- Beispiel: `trait-bsp1/main.rs` (extern)

Default-Implementierungen

- anstelle alle Methoden eines Traits immer implementieren zu müssen, kann man eine Default-Implementierung angeben:
- Funktions„körper“ in Trait-Definition
- Trait-Implementierung leer (`{}`)

```
pub trait Points3D {
    fn to_origin(&self) -> f64 {
        0.0 // default implementation
    }
    fn output(&self) -> String {
        format!("To be done") // default implementation
    }
}
```

```
impl Points3D for IntPoint {
    // no implementation yet -> default impl kicks in
}
```

Traits als Funktionsparameter

Idee: Definition (und Implementierung) von Funktionen, die über vielen verschiedenen Typen (die alle den angegebenen Trait implementieren) funktionieren.

Allgemeine Syntax:

```
// trait as parameter
pub fn printpoint(pt: &impl Points3D) {
    println!("Point lies at {}", pt.output());
}
```

- „**`impl trait`“-Syntax
- Funktionen, die den Trait konstituieren, können in der definierten Funktion genutzt werden (hier: `output()`)
- funktioniert mit allen Typen, die Trait `Points3D` implementieren

Beschränkungen von Traits (*Trait Bounds*)

- äquivalente Langform
- gebräuchlicher, da ausdrucksstärker

```
pub fn printpoint<T: Points3D>(pt: &T) {
  println!("Point lies at {}", pt.output());
}
```

Mehrere Traits als Parameter

Entweder so:

```
pub fn combinepoints(pt1: &impl Points3D,
  pt2: &impl Points3D) {
  ...
}
```

- gleiche und unterschiedliche Typen für `pt1` und `pt2` möglich

. . . oder so:

```
pub fn combinepoints<T: Points3D>(pt1: &T, pt2: &T) {
  ...
}
```

- erzwingt gleiche Typen für `pt1` und `pt2`.

Angabe mehrerer *Trait Bounds*

- Was, wenn mehrere *Trait Bounds* zur Implementierung nötig sind?

⇝ So:

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

- `item` muss die Traits `Summary` **und** `Display` anbieten

. . . oder so:

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

Trait Bounds mit `where` Clauses

Problem: Angabe vieler *Trait Bounds* ist unübersichtlich, daher anstelle:

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

. . . einfacher kodieren:

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

Beispiel: Maximum-Funktion für 2 Parameter beliebigen Typs

```
fn max<T> (i1: T, i2: T) -> T {
    if i1 > i2 { i1 }
    else { i2 }
}
fn main() {
    println!("max(3, 4) = {}", max(3, 4));
    println!("max('z', '0') = '{}'", max('z', '0'));
    println!("max(\"Hamster\", \"Hase\") = \"{}\"",
    ↪  max("Hamster", "Hase"));
    println!("max(3.14, -3.14) = {}", max(3.14, -3.14));
    println!("max(3.14, 3.0/0.0) = {}", max(3.14, 3.0/0.0));
}
```

- übersetzt nicht, weil $<$ nicht für jeden Typ implementiert ist

```
help: consider restricting type parameter `T`
  |
1 | fn max<T: std::cmp::PartialOrd> (i1: T, i2: T) -> T {
  |         ++++++++++++++++++++++
```

Beispiel: Maximum-Funktion für 2 Parameter beliebigen Typs

- Trait Bound nötig:

```
fn max<T: std::cmp::PartialOrd> (i1: T, i2: T) -> T {
```

```
   Compiling max2 v0.1.0 (/home/robge/txt/job/htw/rust/src/max2)
    Finished dev [unoptimized + debuginfo] target(s) in 0.22s
     Running `target/debug/max2`
max(3, 4) = 4
max('z', '0') = 'z'
max("Hamster", "Hase") = "Hase"
max(3.14, -3.14) = 3.14
max(3.14, 3.0/0.0) = inf
```

Nächstes Problem: Ergebnis in letzter Zeile

- `f64` ist eine **Halbordnung** (reflexiv, antisymmetrisch, transitiv), d. h. , es gibt Elemente, die sich nicht vergleichen lassen
- z. B. `NaN, MIN, MAX_10_EXP` (vgl. `https://doc.rust-lang.org/std/f64/index.html#constants`)

Bessere Maximum-Funktion für 2 Parameter beliebigen Typs

Lifetimes

- . . . gibt es nicht in anderen Programmiersprachen
- Zweck: Verhinderung hängender Referenzen

Was haben wir gelernt?