Systemnahe Programmierung in Rust

- “The Book” / Fortgeschrittene Sprachmerkmale / Kap. 19 -

Hubert Högl

Hochschule Augsburg / Informatik

2022-12-15 14:25:55

Unsafe Rust (19.1)

Unsafe Rust

- Bisher nur “safe Rust”: Compiler ist konservativ, er lehnt auch manche Programme ab, die sicher wären.
- Nun “unsafe Rust”:
  - Die ProgrammiererIn übernimmt selber die Verantwortung. Mit Sorgfalt prüfen, dass keine Speicherfehler entstehen (z.B. Nullpointer Dereferenzierung).
  - Hardwarenahe Programmierung geht nur im unsicheren Modus. Beispiele sind mit einem Betriebssystem über Systemaufrufe zu kommunizieren oder selbst ein Betriebssystem zu schreiben.
- Unsafe Block:

```
unsafe {
    ...
}
```

Unsafe Rust (2)

Im Unsafe Block ist erlaubt:

- Dereferenzieren eines Rohzeigers
- Aufrufen einer unsicheren Funktion oder Methode
- Zugreifen auf oder Ändern einer veränderlichen statischen Variablen
- Implementieren eines unsicheren Merkmals (trait)
- Zugreifen auf Felder in union

Ziele:

- Isolieren des unsicheren Code Blocks und Bereitstellen einer sicheren API.
- Unsafe Blöcke klein halten.

Unsafe Rust (3) - Dereferenzieren eines Rohzeigers

- Bisher waren in Rust Referenzen immer gültig.
- In Unsafe Rust gibt es nun rohe Zeiger (*raw pointer*):

```
*const T
*mut T
```

  Der Stern ist Teil des Typnamens (keine Dereferenzierung)

- Rohzeiger sind anders als Referenzen und intelligente Zeiger:
  - Sie dürfen die Ausleihregeln ignorieren, indem sie sowohl unveränderliche als auch veränderliche Zeiger oder mehrere veränderliche Zeiger auf die gleiche Stelle haben.
  - Sie zeigen nicht garantiert auf gültigen Speicher.
  - Sie dürfen null sein.
  - Sie implementieren keine automatische Bereinigung.

Unsafe Rust (4) - Dereferenzieren eines Rohzeigers (2)

Rohzeiger können im sicheren Code erzeugt, aber nicht dereferenziert werden!

```
let mut num = 5;    // r1 und r2 zeigen auf num

let r1 = &num as *const i32;    // nicht aenderbar
let r2 = &mut num as *mut i32;    // aenderbar

unsafe {
    println!("r1 ist: {}", *r1);
    println!("r2 ist: {}", *r2);
}
```

Zeiger auf eine absolute Speicheradresse:

```
let address = 0x012345usize;
let r = address as *const i32;
```

Unsafe Rust (5) - Aufrufen einer unsicheren Funktion oder Methode

```rust
unsafe fn dangerous() { ... }  // Funktionsrumpf ist unsafe Block

unsafe {
    dangerous();
}
```

Sichere Abstraktion von unsicherem Code

Beispiel:

```rust
fn split_at_mut(values: &mut [i32], mid: usize) -> (&mut [i32], &mut [i32]) { ... }
```

Unsafe Rust (6)

```rust
use std::slice;

fn split_at_mut(values: &mut [i32], mid: usize) -> (&mut [i32], &mut [i32]) {
    let len = values.len();
    let ptr = values.as_mut_ptr();

    assert!(mid <= len);

    unsafe {
        (
            slice::from_raw_parts_mut(ptr, mid),
            slice::from_raw_parts_mut(ptr.add(mid), len - mid),
        )
    }
}
```

Unsafe Rust (7)

Geht nicht:

```
use std::slice;

let address = 0x01234usize;
let r = address as *mut i32;

let values: &[i32] = unsafe { slice::from_raw_parts_mut(r, 10000) };
```

Unsafe Rust (8)

Verwenden von extern-Funktionen um externen Code aufzurufen

```
extern "C" {   // C ABI
    fn abs(input: i32) -> i32;
}

fn main() {
    unsafe {
        println!("Absolutwert von -3 gemäß C: {}",
        abs(-3));
    }
}
```

Nun Rust aus C aufrufen (kein `unsafe` nötig):

```
#[no_mangle]
pub extern "C" fn call_from_c() {
    println!("Rust-Funktion von C aufgerufen!");
}
```

Unsafe Rust (9) - Zugreifen oder Ändern einer veränderlichen, statischen Variable

Statische Variable sind globale Variable

In sicherem Code nur lesbar

Haben feste Adresse im Speicher (im Unterschied zu Konstanten, diese dürfen dupliziert werden)

Lesen und Schreiben bei `static mut` muss in `unsafe` Block sein, da in einem anderen Thread ein schreibender Zugriff erfolgen könnte (siehe Kap. 16).

```
static mut COUNTER: u32 = 0;

fn add_to_count(inc: u32) {
    unsafe { COUNTER += inc; }
}

fn main() {
    add_to_count(3);
    unsafe { println!("COUNTER: {}", COUNTER); }
}
```

Unsafe Rust (6) - Implementieren eines unsicheren Merkmals

Ein Trait ist unsafe, wenn mindestens eine Methode unsafe ist.

```
unsafe trait Foo {
    // Methoden kommen hierhin
}

unsafe impl Foo for i32 {
    // Methoden-Implementierungen kommen hierhin
}

fn main() {}
```

Unsafe Rust (7) - Zugreifen auf Felder einer Vereinigung (union)

Unions sind ähnlich wie `struct`, jedoch teilen sich alle Felder den gleichen Speicher. Unions werden häufig in C Code verwendet.

```
#[repr(C)]
union MyUnion {
    f1: u32,
    f2: f32,
}

let u = MyUnion { f1: 1 };
let f = unsafe { u.f1 };
```

Quelle: https://doc.rust-lang.org/reference/items/unions.html

Traits (19.2)

Traits

Siehe auch die Abschnitte 10.2 (Merkmale), 17.2 (Merkmalsobjekte)

Merkmal mit **assoziiertem Typ**

```
pub trait Iterator {
    type Item;

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

impl Iterator for Counter {
    type Item = u32;

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

- Beachte Unterschied generischer Datentyp $\leftrightarrow$ assoziierter Typ

Standardparameter für generische Typen und Operatorüberladung

Trait definiert in `std::ops::Add`:

```
trait Add<Rhs=Self> {
    type Output;

    fn add(self, rhs: Rhs) -> Self::Output;
}
```

Standardtypparameter (*default type parameter*): `Rhs=Self`

Beispiel für Standardtyp:

```
struct Point {
    x: i32,
    y: i32,
}
```

... // siehe naechste Seite

Standardparameter für generische Typen und Operatorüberladung (2)

```
... // siehe vorherige Seite

impl Add for Point {
    type Output = Point;

    fn add(self, other: Point) -> Point {
        Point {
            x: self.x + other.x,
            y: self.y + other.y,
        }
    }
}
```

Beachte: Hinter `impl Add` muss man nichts in spitzen Klammern angeben. Der "Normalfall" ist, dass zwei gleiche Typen addiert werden.

Standardparameter für generische Typen und Operatorüberladung (3)

Nun Rhs *nicht* der Standardtyp:

```
use std::ops::Add;

struct Millimeters(u32);
struct Meters(u32);

impl Add<Meters> for Millimeters {
    type Output = Millimeters;

    fn add(self, other: Meters) -> Millimeters {
        Millimeters(self.0 + (other.0 * 1000))
    }
}
```

Beachte: `impl Add<Meters>`, `Meters` ist nun der vom Standard abweichende Typ.

Standardparameter für generische Typen und Operatorüberladung (4)

Standardtypparameter werden verwendet

- um einen Typ zu erweitern, ohne bestehenden Code zu brechen.
- um eine Anpassung in bestimmten Fällen zu ermöglichen, die die meisten Benutzer nicht benötigen (wie bei `impl Add<Meters>`).

Eindeutige Syntax bei Methodenaufrufen

```
trait Animal {
    fn baby_name() -> String;
}

struct Dog;

impl Dog {
    fn baby_name() -> String { String::from("Spot") }
}

impl Animal for Dog {
    fn baby_name() -> String { String::from("Welpe") }
}

...
<Dog as Animal>::baby_name();    // Animal::baby_name() kompiliert nicht!
```

Voll qualifizierte Syntax

```
<Type as Trait>::function(receiver_if_method, next_arg, ...);
```

Supermerkmale

`OutlinePrint` erfordert das Merkmal `Display`:

```
trait OutlinePrint: fmt::Display {
    fn outline_print(&self) {
        ...
    }
```

- Unterschied zur **Merkmalsabgrenzung** (*trait bound*), siehe Kap. 10.2
- Gegenbeispiel: `impl OutlinePrint for Point {}` würde nicht gehen, solange `Display` nicht für `Point` implementiert ist.

Das Newtype-Muster

- *Waisenregel*: Merkmal kann nur dann auf einem Typ implementiert werden, wenn entweder das Merkmal oder der Typ lokal im Crate vorkommen.
- Umgehen der Waisenregel durch Newtype-Muster: Externe Merkmale auf externen Typen damit möglich.
- Beispiel:

```
struct Wrapper(Vec<String>);

impl fmt::Display for Wrapper {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "[{}]", self.0.join(", "))
    }
}
```

- Falls der äussere Typ `Wrapper` alle Methoden des inneren Typs `Vec` haben soll, muss der `Deref` Trait implementiert werden, so dass der innere Typ zurückgegeben wird.

Fortgeschrittene Typen (19.3)

Das Newtype Muster

(siehe auch 19.2)

- damit Werte nicht verwechselt werden können
- Angabe von Einheiten (Bsp.: Millimeter, Meter)
- Implementierungsdetails eines Typs abstrahieren, d.h. der neue Typ kann ein anderes API als der private innere Typ haben
- Kapselung, um Implementierungsdetails zu verbergen (siehe 17.1)

Typ-Synonyme mit Typ-Alias

```
type Kilometers = i32;  //
type Result<T> = std::result::Result<T, std::io::Error>;  // in std lib
```

Fortgeschrittene Typen (2)

Bessere Lesbarkeit

```
type Thunk = Box<dyn Fn() + Send + 'static>;

let f: Thunk = Box::new(|| println!("hallo"));

fn takes_long_type(f: Thunk) { ... }
fn returns_long_type() -> Thunk { ... }
```

Der “Niemals” Typ

`!` ist ein sogenannter “leerer Typ”, weil er keine Werte hat. Die folgende Funktion kehrt niemals zurück, deshalb wird der `!` Operator auch *never type* genannt:

```
fn bar() -> ! { ... }
```

`continue` gibt `!` zurück, deshalb kompiliert der folgende Code:

```
let guess: u32 = match guess.trim().parse() {
    Ok(num) => num,
    Err(_) => continue, // ! kann in jeden anderen Typ umgewandelt werden
};
```

Fortgeschrittene Typen (3)

Fortsetzung: Never Type

`panic!` gibt `!` zurück, deswegen kann der Rückgabewert von `unwrap()` `T` sein:

```
impl<T> Option<T> {
   pub fn unwrap(self) -> T {
      match self {
         Some(val) => val,
         None => panic!("Aufruf von `Option::unwrap()` auf einem `None`-Wert"),
      }
   }
}
```

Typ des folgenden Ausdrucks ist `!`

```
loop {
    print!("endlos");
}
```

Fortgeschrittene Typen (4)

Dynamisch große Typen und das Merkmal Sized

DST (*dynamically sized types*, oder *unsized types*): Grösse erst zur Laufzeit bekannt.

Beispiel:

- `str` ist DST
- `&str` ist *kein* DST, siehe *string slice*, Kap. 4)

DST muss immer hinter Zeiger liegen: `&str`, `Box<str>`, `Rc<str>` (genauer: Zeiger + Länge),

Geht nicht:

```
let s1: str = "Guten Tag!";
let s2: str = "Wie geht es dir?";
```

Trait Objects sind auch DSTs (Kap. 17.2): `&dyn Trait`, `Box<dyn Trait>`, `Rc<dyn Trait>`

Mit Merkmal `Sized` werden alle Typen automatisch versehen, die eine feste Grösse haben. Die folgende linke generische Funktion wird also wie rechts zu sehen behandelt. `T` kann also nur eine feste Grösse haben:

```
fn generic<T>(t: T) { ... }      =======>    fn generic<T: Sized>(t: T) { ... }
```

Fortgeschrittene Typen (5)

Lockerung mit `?Sized`: “`T` kann `Sized` sein oder nicht” (geht nur bei `Sized`)

```
fn generic<T: ?Sized>(t: &T) { ... }
```

Erweiterte Funktionen und Funktionsabschlüsse (Closures)

Erweiterte Funktionen und Funktionsabschlüsse (1)

Funktionszeiger

Funktionen haben den Typ `fn` (Funktionszeiger, *function pointer*)

```
fn add_one(x: i32) -> i32 {
    x + 1
}

fn do_twice(f: fn(i32) -> i32, arg: i32) -> i32 {
    f(arg) + f(arg)
}

fn main() {
    let answer = do_twice(add_one, 5);

    println!("Die Antwort ist: {}", answer);
}
```

`Fn`, `FnMut`, `FnOnce` sind **Traits**, `fn` ist ein **Typ**. Deswegen kann `fn` direkt in Funktionssignatur verwendet werden, Traits nur über Merkmalsabgrenzungen.

Erweiterte Funktionen und Funktionsabschlüsse (2)

`fn` implementiert `Fn`, `FnMut`, `FnOnce`. Deswegen kann man eine Funktion an der Stelle einsetzen, an der eine Closure erwartet wird.

Beispiel:

```
let list_of_numbers = vec![1, 2, 3];
let list_of_strings: Vec<String> =
list_of_numbers.iter().map(|i| i.to_string()).collect(); // Closure
```

Funktion `to_string()` im Trait `ToString` mit voll qualifiziertem Namen aufrufen:

```
let list_of_numbers = vec![1, 2, 3];
let list_of_strings: Vec<String> =
list_of_numbers.iter().map(ToString::to_string).collect();
```

Erweiterte Funktionen und Funktionsabschlüsse (3)

`Enums`: Jede Variante hat auch eine Initialisierungsfunktion (Erinnerung Kap. 6.1, Konstruktorfunktion, `IpAddr::V4()`).

```
enum Status {
    Value(u32),
    Stop,
}

let list_of_statuses: Vec<Status> = (0u32..20).map(Status::Value).collect();
```

Erweiterte Funktionen und Funktionsabschlüsse (4)

Zurückgeben von Funktionsabschlüssen

Traits sind DSTs, deshalb kann man sie nicht direkt aus Funktionen zurück geben.

Funktionszeiger können auch nicht aus Funktionen zurück gegeben werden.

Lösung: Trait Object

```
fn returns_closure() -> Box<dyn Fn(i32) -> i32> {
            Box::new(|x| x + 1)
}
```

Makros (19.5)

- Deklarative Makros mit `macro_rules!`

  Beispiel: `vec!`
- Prozedurale Makros
  - Benutzerdefinierte Makros mit `#[derive]`
  - Attribut-ähnliche Makros, die benutzerdefinierte Attribute definieren, die bei jedem Element verwendet werden können
  - Funktions-ähnliche Makros, die wie Funktionsaufrufe aussehen, aber auf den als Argument angegebenen Token operieren
- Der Unterschied zwischen Makros und Funktionen

Makros (2)

Deklarative Makros

- "Makros am Beispiel" oder `macro_rules!` Makro
- Vereinfachte Variante von `vec!` selber schreiben, so dass es wie folgt verwendet werden kann: `let v: Vec<u32> = vec![1, 2, 3];`

```
#[macro_export]
macro_rules! vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
```

Lit.: The Little Book of Rust Macros https://veykril.github.io/tlborm/

Makros (3)

Prozedurale Makros zur Code-Generierung aus Attributen

XXX to do

Wie man ein benutzerdefiniertes Makro mit derive schreibt

XXX to do

Attribut-ähnliche Makros

XXX to do

Funktions-ähnliche Makros

XXX to do