Seminararbeit Traits und Enums in Rust

Mario Occhinegro
HKA University of Applied Sciences

Inhaltsverzeichnis

Zusammenfassung

1 Einleitung

2 Enums

Enumerationstypen
Auf den ersten Blick identisch.
Java Enum:

```
enum Color{
    red,
    green,
    blue;
}
```

Rust Enum:

```
enum Animal {
    Dog,
    Cat,
    Bird,
}
```

Auf Konkretere Unterschiede gehen wir jetzt ein

2.1 Enums in Rust

- Algebraische Datentypen

2.1.1 Match Statement

- abgeschlossen
- an Haskell angelehnt
- mehr als nur if else
- sehr ergonomisch, aussagekräftig und kurz

```
fn main() {
        let a1 = Animal::Dog;
        match a1{
                Animal::Dog => println!("It's a Dog"),
                Animal::Cat => println!("It's a Cat"),
                Animal::Bird => println!("It's a Bird")
        }
}

enum Animal{
```

```
        Dog,
        Cat,
        Bird
}
```

2.1.2 Der Enum als algebraischer Datentyp

- beliebige Struktur
- werte können sich verändern
- flexibel
- pattern matching lässt uns die einzelnen Werte benutzen

```
fn main() {
    let s1 = Shape::Square(16);
    println!("The area of the shape is {}",s1.area());
}

enum Shape{
    Square(u32),
    Rectangle(u32,u32),
}

impl Shape{
    fn area(&self) -> u32{
        match self {
            Shape::Square(a) => a*a,
            Shape::Rectangle(a,b) => a*b,
        }
    }
}
```

2.1.3 Rekusive Enums und Datentypen

- braucht Box (wie Zeiger)
- Box sonst, rekursive Definition ohne Direktion

```
enum Exp {
    Int {
        val: i32
    },
    Plus {
        left: Box<Exp>,
        right: Box<Exp>
    },
    Mult{
        left: Box<Exp>,
        right: Box<Exp>
    },
```

```
}

impl Exp{
    fn eval(&self) -> i32{
        match self{
            Exp::Int{val} => *val,
            Exp::Plus{left, right} => left.eval() + right.eval() ,
            Exp::Mult{left, right} => left.eval() + right.eval()

        }
    }
}
```

2.1.4 Nested Pattern Matching

- kann noch granulareres pattern matching betreiben

```
pub enum Exp {
    Int {
        val: i32
    },
    Plus {
        left: Box<Exp>,
        right: Box<Exp>
    },
    Mult{
        left: Box<Exp>,
        right: Box<Exp>
    },
}

impl Exp{
   fn eval(&self) -> i32{
       match self{
           Exp::Int{val} => *val,
           Exp::Plus{left, right} => left.eval() + right.eval() ,
           Exp::Mult{left, right} =>
            match **left {
                Exp::Int { val:0 } => return 0,
                _ => return left.eval() * right.eval()
            }
       }
   }
}
```

2.1.5 Generische Enums

- Enums können mit generischen Werten generiert werden

```
enum Option<T> {
        None,
```

```
        Some(T),
}
```

2.1.6 Rust Enums und die Vermeidung von Nullpointer-Ausnahmen

- Java hat ähnliches Konzept aber mit Klassen
- Nullpointer, der große Milliarden € Fehler

```
mintedfn main() {
    match lookUpAnimal(1){
        Some(Animal::Dog) => println!("Found pet was a dog"),
        Some(_) => println!("Found pet with id 1"),
        None => println!("Sadly no pet was found")
    }
}

enum Animal{
    Dog,
    Cat,
    Bird,
}

fn lookUpAnimal(id: i32) -> Option<Animal>{
    if(id == 1){
        return Some(Animal::Dog);
    }else{
        return None
    }
}
```

2.2 Enums in Java

- Enums sind spezielle Klasse
- Enumtypen sind Instanzen
- Instanz statisch und final (per default)

2.2.1 Normale Enums

```
enum Animal{
    Dog,
    Cat,
    Bird
}
```

2.2.2 Enums mit Werten

```
enum AnimalWithValues{
    Dog("Dog", 20),
    Cat("Dog", 10),
```

```
    Bird("Bird", 1);

    public final String label;
    public final int weight;

    //constructor
    private AnimalWithValues(String label, int weight){
        this.label= label;
        this.weight = weight;
    }
}
```

2.2.3 Enum Funktionen

2.3 Vergleich von Java und Rust Enums

2.4 Rust Enum Implementationsbeispiele

2.4.1

2.5 Beispielfunktionalität in Java

2.5.1 Expression-Logik in Java

Naiver Ansatz

```
public class Expression{
    public static void main(String[] args) {
        Exp p = Exp.Plus;
//not accessible
        System.out.println(p.left);
        System.out.println(p.right);
    }
}

enum Exp {
    Int {
        //cannot be changed(static, final)
        int val;

public int eval() {
            return this.val;
        }

    },
    Plus {
        Exp left;
        Exp right;

        public int eval() {
```

```
            return this.left.eval() + this.right.eval();
        }
    },
    Mult {
        Exp left;
        Exp right;

        public int eval() {
            return this.left.eval() * this.right.eval();
        }
    };


    public abstract int eval();
}

enum ExpTwo{
    Int,
    Plus,
    Mult
}
```

Ansatz mit Klassen

```
public class Expression {
    public static void main(String[] args) {
        System.out.println("test");
    }
}

abstract class Exp{abstract public int eval();}
class IntExp extends Exp{
    public int val;
    public IntExp(int val){
        this.val = val;
    }
    @Override
    public int eval() {
        return val;
    }
}
class PlusExp extends Exp{
    public Exp left;
    public Exp right;
    public PlusExp(Exp left, Exp right){
        this.left = left;
        this.right = right;
    }
    @Override
    public int eval() {
        return left.eval() + right.eval();
```

```
    }
}

class MultExp extends Exp{
    public Exp left;
    public Exp right;

    public MultExp(Exp left, Exp right){
        this.left = left;
        this.right = right;
    }

    @Override
    public int eval() {
        return left.eval() * right.eval();
    }
}
```

2.5.2 Java Enums am Limit - Wrapperinstanz für den Typ

- Idee, was aber wenn die Instanz ein Wrapper ist
- nicht sehr ergonomisch
- statische variablen schneiden uns

```
mintedpublic class playground{
        public static void main(String[] args) {
                Animal a = Animal.Dog;
                Animal a2 = Animal.Dog;
                Animal b = Animal.Cat;
                System.out.println(a.getObject());
                System.out.println(a2.getObject());
                System.out.println(b.getObject());
                a.setObject("new Dog Value");
                b.setObject("new Cat value");
                System.out.println(a.getObject());
                System.out.println(a.getObject());
                System.out.println(b.getObject());
        }
}

enum Animal{
        Dog(new Wrapper("Doggy")),
        Cat(new Wrapper("Catty"));

        private Wrapper w;
        private Animal(Wrapper w){
                this.w = w;
        }
```

```
        public Object getObject(){
                return w.item;
        }
        public void setObject(Object o){
                w.item = o;
        }
}

class Wrapper{
        Object item;

        public Wrapper(Object o){
                item = o;
        }
}
```

output

```
Doggy
Doggy
Catty
new Dog Value
new Dog Value
new Cat value
```

3 Traits

3.1 Allgemeines zu Traits

1. geteilte funktionalität mit anderen Typen
2. Funktionsmenge über einem Typen
3. Oft mit Interfaces verglichen, sind aber keine Interfaces
4. interfaces sind Typen
5. adressieren ähnliche Probleme, traits aber mächtiger

3.1.1 Traits sind keine Typen

3.2 Traits in Rust

1. Prädikat auf einem Typen

```
mintedtrait Shape{
    fn area(s: &Self) ->i32;
}

struct Square{
    a: i32
}
```

```
impl Shape for Square {
    fn area(s: &Self)->i32{
        s.a*s.a
    }
}

struct Rectangle{
    a: i32,
    b: i32
}

impl Shape for Rectangle {
    fn area(s: &Self)->i32{
        s.a*s.b
    }
}
```

3.2.1 Default-Implementationen

1. geht in java auch

```
fn main() {
    let c1:Cat  = Cat{};
    Animal::makeNoise(&c1);
}

trait Animal{
    fn makeNoise(s: &Self){
        println!("The Animal made a noise");
    }
}
struct Cat{}
impl Animal for Cat{}
```

When running main yields

```
The Animal made a noise
```

3.2.2 Trait Bounds

```
//Das Shape Prädikat muss für A und für B gelten
fn sum_area<A:Shape,B:Shape>(x : &A, y : &B) -> i32 {
    return area(x) + area(y)
 }
```

3.2.3 Multiples Binding

Man kann auch Prädikate/Traits verunden

```
fn sum_area<A:Shape+OtherTraits>(x : &+OtherTraits) -> i32 {
        ...
 }
```

3.2.4 Konditionelle Implementierung mit Trait Bounds

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

struct dog{
    name: String,
    age: u8,
}

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!("The largest member is x = {}", self.x);
        } else {
            println!("The largest member is y = {}", self.y);
        }
    }
}
```

3.2.5 Dynamische Traits

Repräsentieren von Interfaces in Rust
Können Konkrete Typen als Parameter und Rückgabewerte nutzen

```
fn sum_area(x : Box<dyn Shape>, y: Box<dyn Shape>) -> i32 {
    return area(x) + area(y)
 }
```

3.2.6 Kurzschreibweise für dynamische Traits

```
fn sum_area(x : &(impl Shape), y: &(impl Shape)) -> i32 {
    return area(x) + area(y)
 }
```

3.2.7 Where Clause

3.2.8 Shorthand Schreibweise

Andere Schreibweise, so kann man die Funktion auf einer Instanz des Structs aufrufen

```
trait Shape{
   fn area(&self) -> String;
}

impl Shape for Square{
   fn area(&self) -> i32{
       self.a*self.a
   }
}

fn main() {
   let s = Square{a: 10};
   print!("{}", s.area());
}
```

3.2.9 Schreibeweise bei Uneindeutigkeit

```
fn main() {
   let s = Square{a: 10};
   print!("{}", Shape2::area(&s));
}

struct Square{
    a: u32,
}
trait Shape{
   fn area(&self) -> u32;
}

impl Shape for Square{
   fn area(&self) -> u32{
       self.a*self.a
   }
}
trait Shape2{
   fn area(&self) -> u32;
}

impl Shape2 for Square{
   fn area(&self) -> u32{
       self.a*self.a
   }
}
```

```rust
trait Shape{
   fn area(&self) -> String;
}

impl Shape for Square{
   fn area(&self) -> i32{
       self.a*self.a
   }
}
trait Shape2{
   fn area(&self) -> String;
}

impl Shape2 for Square{
   fn area(&self) -> i32{
       self.a*self.a
   }
}


fn main() {
   let s = Square{a: 10};
   print!("{}", Shape2::area(s));
}
```

3.2.10 Supertraits

- man kann hirarchie nachbauen

```rust
fn main() {
    let s = HskaStudent{name:"Mario", university:"hska", fav_language:"rust", git_username
    comp_sci_student_greeting(&s);
}
trait Person {
    fn name(&self) -> String;
}
trait Student: Person {
    fn university(&self) -> String;
}
trait Programmer {
    fn fav_language(&self) -> String;
}
trait CompSciStudent: Programmer + Student {
    fn git_username(&self) -> String;
}
fn comp_sci_student_greeting<S: CompSciStudent>(student: &S) {
    println!("Hey my name is {}, I study at {}. My favorite language is {} and my git user
}
struct HskaStudent{
    name: &'static str,
```

```
    university: &'static str,
    fav_language: &'static str,
    git_username: &'static str,
}
impl Person for HskaStudent{
    fn name(&self) -> String{
        self.name.to_string()
    }
}
impl Student for HskaStudent{
    fn university(&self) -> String {
        String::from(self.university)
    }
}

impl Programmer for HskaStudent{
    fn fav_language(&self) -> String{
        String::from(self.fav_language)
    }
}
impl CompSciStudent for HskaStudent{
    fn git_username(&self) -> String {
        String::from(self.git_username)
    }
}
```

3.2.11 Referenzierung des eigenen Typen

```
trait genCopy{
    fn genCopy(s: &Self) -> Self;
}

struct Dog{
    name: String,
    age: u8,
}
struct Cat{
    name: String,
    age: u8,
}

impl genCopy for Dog{
    fn genCopy(s: &Self) -> Self {
        return Dog{name: s.name.clone(), age: s.age};
    }
}
impl genCopy for Cat{
    fn genCopy(s: &Self) -> Self {
        return Cat{name: s.name.clone(), age: s.age};
    }
```

```
}
```

3.2.12 Platzhaltertypen

```
fn main(){
    let m = Machine{};
    let a: i8 = 16;
    let b: i32 = TransformAB::transform(&m, a);
}
trait TransformAB{
    type A;
    type B;
    fn transform(s: &Self, a: Self::A) -> Self::B;
}

struct Machine{}
impl TransformAB for Machine{
    type A = i8;
    type B = i32;
    fn transform(s: &Self, a: Self::A) -> Self::B {
        i32::from(a)
    }
}
```

3.2.13 Assoziierte Konstanten

```
fn main(){
    let m = Machine{};
    let a: i8 = 16;
    let b: Vec<i32> = TransformAB::transform(&m, a);
}
trait TransformAB{
    type A;
    type B;
    const TIMES: u8;
    fn transform(s: &Self, a: Self::A) -> Vec<Self::B>;
}

struct Machine{}
impl TransformAB for Machine{
    type A = i8;
    type B = i32;
    const TIMES:u8 = 50;
    fn transform(s: &Self, a: Self::A) -> Vec<Self::B>{
        let mut v = Vec::new();
        let a32 = i32::from(a);
        for i in 0..Self::TIMES  {
            v.push(a32);
        }
        v
```

```
    }
}
```

3.3 Rust Trait Beispiele

3.3.1 Nicht vereinbare Interfaces

3.3.2 Generische Mehrfachimplementierung

3.4 Beispielfunktionalität in Java

4 Vergleich der beiden Ansätze