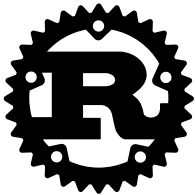
Rust
Florob
Meta
Ownership
Einführung
Beispiel
Features
ADTs
Patterns
Iterators
Unsafe



Rust

Florian “Florob” Zeitz

2015-10-28

Gliederung

Meta

Ownership
Einführung
Beispiel

Features
ADTs
Patterns
Iterators
Unsafe

1. Meta
2. Ownership
   - Einführung
   - Beispiel
3. Features
   - Algebraische Datentypen
   - Pattern Matching
   - Iterators
   - Unsafe

Gliederung

Meta

Ownership
Einführung
Beispiel

Features
ADTs
Patterns
Iterators
Unsafe

1. Meta
2. Ownership
   - Einführung
   - Beispiel
3. Features
   - Algebraische Datentypen
   - Pattern Matching
   - Iterators
   - Unsafe

- Systemprogrammiersprache
- kompiliert (LLVM backend)
- statisch typisiert
- stark typisiert
- affines Typsystem
- low-level, mit high-level Funktionalität
- große Community
- inspiriert von: C++, Erlang, Haskell, OCaml, Swift, ...

Geschichte

Rust
Florob
Meta
Ownership
Einführung
Beispiel
Features
ADTs
Patterns
Iterators
Unsafe

ab 2006	persönliches Projekt von Graydon Hoare
	Compiler in OCaml
seit 2009	Entwicklung unterstützt von Mozilla, als Teil von Mozilla Research
seit 2011	selbst compilierender Compiler
seit 2014	Sprachänderungen durch RFC Prozess
Mai 2015	Veröffentlichung von Rust 1.0

Philosophie

Rust
Florob
Meta
Ownership
Einführung
Beispiel
Features
ADTs
Patterns
Iterators
Unsafe

- Speichersicher
  - kein use-after-free
  - keine out-of-bounds Zugriffe
- keine data races
- minimale Laufzeitumgebung
- kein obligatorischer Garbage Collector
- keine versteckten Kosten
- immutable by default
- zero-cost abstractions

Gliederung

Meta

Ownership
Einführung
Beispiel

Features
ADTs
Patterns
Iterators
Unsafe

1. Meta
2. Ownership
   - Einführung
   - Beispiel
3. Features
   - Algebraische Datentypen
   - Pattern Matching
   - Iterators
   - Unsafe

Gliederung

Meta

Ownership

Einführung

Beispiel

Features

ADTs

Patterns

Iterators

Unsafe

1. Meta
2. Ownership
   - Einführung
   - Beispiel
3. Features
   - Algebraische Datentypen
   - Pattern Matching
   - Iterators
   - Unsafe

C++: Realloc


```cpp
#include <iostream>
#include <vector>
#include <string>

int main() {
  std::vector<std::string> v;

  v.emplace_back("Foo");
  auto const &x = v[0];
  v.emplace_back("Bar");
  std::cout << x << '\n';

  return 0;
}
```

C++: Realloc

Rust
Florob
Meta
Ownership
Einführung
Beispiel
Features
ADTs
Patterns
Iterators
Unsafe

```
$ clang++ -std=c++14 -Wall vector.cc -o vector-cc
$ ./vector-cc
zsh: segmentation fault (core dumped)  ./vector-cc
```

C++: Iterator Invalidation


```cpp
#include <iostream>
#include <string>
#include <vector>

int main() {
    std::vector<std::string> v = { "F", "o", "o" };

    for (auto const &it : v) {
        v.push_back(it + it);
    }
    for (auto const &it : v) {
        std::cout << it << '\n';
    }

    return 0;
}
```

C++: Iterator Invalidation


```
$ clang++ -std=c++14 -Wall iter.cc -O -o iter-cc
$ ./iter-cc
F
o
o
FF


$
```

Beobachtung

Rust
Florob
Meta
Ownership
Einführung
Beispiel
Features
ADTs
Patterns
Iterators
Unsafe

Probleme entstehen bei der Kombination von:

Veränderlichkeit + **Aliasing**

```
v.emplace_back(...)
v.push_back(...)
```

```
auto const &x = v
auto const &it : v
```

Rusts Lösung

- Ownership: `fn f(x: Type) {...}`
  - ein Besitzer
  - veränderlich†
  - lesbar
  - kann ausgeliehen werden
- Shared borrow: `fn f(x: &Type) {...}`
  - beliebig teilbar (aliasing)
  - unveränderlich
  - lesbar
- Mutable borrow: `fn f(x: &mut Type) {...}`
  - nur eines zur Zeit
  - veränderlich
  - lesbar

Rusts Lösung

- Ownership: `fn f(x: Type) {...}`
  - ein Besitzer
  - veränderlich†
  - lesbar
  - kann ausgeliehen werden
- Shared borrow: `fn f(x: &Type) {...}`
  - beliebig teilbar (aliasing)
  - unveränderlich
  - lesbar
- Mutable borrow: `fn f(x: &mut Type) {...}`
  - nur eines zur Zeit
  - veränderlich
  - lesbar

Rusts Lösung

- Ownership: `fn f(x: Type) {...}`
  - ein Besitzer
  - veränderlich†
  - lesbar
  - kann ausgeliehen werden
- Shared borrow: `fn f(x: &Type) {...}`
  - beliebig teilbar (aliasing)
  - unveränderlich
  - lesbar
- Mutable borrow: `fn f(x: &mut Type) {...}`
  - nur eines zur Zeit
  - veränderlich
  - lesbar

Ownership

Rust
Florob
Meta
Ownership
Einführung
Beispiel
Features
ADTs
Patterns
Iterators
Unsafe

```
1  struct Crop;
2  struct Flour;
3
4  fn grind(_c: Crop) -> Flour {
5      Flour
6      // _c is freed here
7  }
8
9  fn main() {
10     let c = Crop;
11
12     grind(c); // c moves into `grind()`
13     // grind(c); // error: use of moved value: `c`
14 }
```

Shared Borrow


```
1  struct Book { page: u32 }
2
3  fn read(b: &Book) {
4      println!("I read page {}", b.page);
5  }
6
7  fn main() {
8      let b = Book { page: 1 };
9      let l = &b;
10
11     read(&b);
12     read(l);
13     read(&b);
14 }
```

Mutable Borrow


```
7  fn turn_page(b: &mut Book) { b.page += 1; }
8
9  fn main() {
10     let mut b = Book { page: 1 };
11
12     read(&b);
13     turn_page(&mut b);
14     read(&b);
15
16     let l = &b;
17     // turn_page(&mut b); // error: cannot borrow `b` as
18                           // mutable because it is also
19                           // borrowed as immutable
20     read(l);
21 }
```

Rust: Realloc


```
1  fn main() {
2      let mut v: Vec<String> = Vec::new();
3
4      v.push("Foo".to_string());
5      let x: &String = &v[0];
6      v.push("Bar".to_string());
7
8      println!("{}", x);
9  }
```

Rust: Realloc


```
$ rustc vector.rs
vector.rs:6:5: 6:6 error: cannot borrow `v` as mutable because it is
 ↪  also borrowed as immutable
vector.rs:6     v.push("Bar".to_string());
                ^
vector.rs:5:23: 5:24 note: previous borrow of `v` occurs here; the
 ↪  immutable borrow prevents subsequent moves or mutable borrows
 ↪  of `v` until the borrow ends
vector.rs:5     let x: &String = &v[0];
                                 ^
vector.rs:9:2: 9:2 note: previous borrow ends here
vector.rs:1 fn main() {
...
vector.rs:9 }
            ^
error: aborting due to previous error
```

Rust: Iterator Invalidation

Rust
Florob
Meta
Ownership
Einführung
Beispiel
Features
ADTs
Patterns
Iterators
Unsafe

```
1  fn main() {
2      let mut v: Vec<String> = vec!["F".into(), "o".into(),
3                                    "o".into()];
4
5      for it in &v {
6          v.push(it.clone() + &it);
7      }
8      for it in &v {
9          println!("{}", it);
10     }
11 }
```

Rust: Iterator Invalidation


```
1  $ rustc iter.rs
2  iter.rs:5:9: 5:10 error: cannot borrow `v` as mutable because it is
    ↪  also borrowed as immutable
3  iter.rs:5          v.push(it.clone() + &it);
4                     ^
5  iter.rs:4:16: 4:17 note: previous borrow of `v` occurs here; the
    ↪  immutable borrow prevents subsequent moves or mutable borrows
    ↪  of `v` until the borrow ends
6  iter.rs:4      for it in &v {
7                             ^
8  iter.rs:6:6: 6:6 note: previous borrow ends here
9  iter.rs:4      for it in &v {
10 iter.rs:5          v.push(it.clone() + &it);
11 iter.rs:6      }
12                ^
13 error: aborting due to previous error
```

Gliederung

Meta

Ownership
Einführung
Beispiel

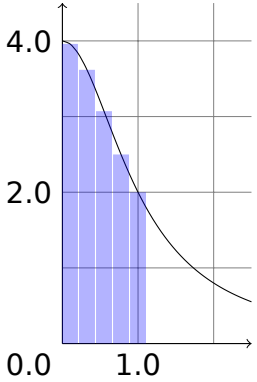
Features
ADTs
Patterns
Iterators
Unsafe

1. Meta
2. Ownership
   - Einführung
   - Beispiel
3. Features
   - Algebraische Datentypen
   - Pattern Matching
   - Iterators
   - Unsafe

Beispiel: Pi Berechnung

$$\pi = 4 \arctan(1)$$
$$= \int_0^1 \frac{4}{1 + x^2} \, dx$$

- Berechne π durch Riemann-Integration
- Nährung des Flächeninhalts durch schmale Rechtecke
- Gut parallelisierbar



C++ Lösung


```cpp
#include <cstdint>
#include <iostream>
#include <thread>
#include <vector>

int main() {
  constexpr uint64_t NUM_THREADS = 4;
  constexpr uint64_t NUM_STEPS = 100000;
  constexpr uint64_t THREAD_STEPS = NUM_STEPS / NUM_THREADS;
  constexpr double STEP = 1.0 / NUM_STEPS;

```

C++ Lösung


```
12    double pi = 0;
13
14    std::vector<std::thread> threads;
15
16    for (int i = 0; i < NUM_THREADS; ++i) {
17      uint64_t lower = THREAD_STEPS * i;
18      uint64_t upper = THREAD_STEPS * (i+1);
19      threads.emplace_back([=, &pi]() {
20        for (uint64_t j = lower; j < upper; ++j) {
21          double x = (j + 0.5) * STEP;
22          pi += 4.0/(1.0 + x*x) * STEP;
23        }
24      });
25    }
```

C++ Lösung


```
26
27    for (auto &t : threads) t.join();
28
29    std::cout.precision(10);
30    std::cout << "Pi = " << pi << '\n';
31
32    return 0;
33  }
```

C++ Lösung


```
1 $ clang++ -std=c++14 -lpthread -Wall pi.cc -o pi-cc
2 $ ./pi-cc
3 Pi = 1.156130797
4 $ ./pi-cc
5 Pi = 1.099799814
```

- Typisches data race
- Thread A liest `pi = 0.1423`
- Thread B liest `pi = 0.1423`
- Thread A schreibt `pi = 0.7609`
- Thread B schreibt `pi = 0.5768`
- `pi = 0.5768`, Thread As Berechnung ist verloren

Rust Lösung A


```
1  use std::thread;
2
3  fn main() {
4      const NUM_THREADS: u64 = 6;
5      const NUM_STEPS: u64 = 100_000;
6      const THREAD_STEPS: u64 = NUM_STEPS / NUM_THREADS;
7      const STEP: f64 = 1.0 / NUM_STEPS as f64;
8
```

Rust Lösung A


```
9       let mut pi: f64 = 0.0;
10
11      let guards: Vec<_> = (0..NUM_THREADS).map(|i| {
12          let lower: u64 = THREAD_STEPS * i;
13          let upper: u64 = THREAD_STEPS * (i+1);
14          let pi_ref: &mut f64 = &mut pi;
15          thread::spawn(move || {
16              for j in lower..upper {
17                  let x: f64 = (j as f64 + 0.5) * STEP;
18                  *pi_ref += 4.0/(1.0 + x*x) * STEP;
19              }
20          })
21      }).collect();
22
```

Rust Lösung A


```
23      for g in guards { g.join().unwrap(); }
24
25      println!("Pi = {:.10}", pi);
26  }
```

Rust Lösung A


```
$ rustc -O pi.rs
pi.rs:14:32: 14:39 error: cannot infer an appropriate lifetime for borrow expression due to
  ↪   conflicting requirements
pi.rs:14         let pi_ref: &mut f64 = &mut pi;
                                        ^~~~~~~
pi.rs:15:9: 20:11 note: first, the lifetime cannot outlive the call at 15:8...
pi.rs:15         thread::spawn(move || {
pi.rs:16             for j in lower..upper {
pi.rs:17                 let x: f64 = (j as f64 + 0.5) * STEP;
pi.rs:18                 *pi_ref += 4.0/(1.0 + x*x) * STEP;
pi.rs:19             }
pi.rs:20         })
pi.rs:15:28: 20:10 note: ...so that captured variable `pi_ref` does not outlive the enclosing closure
pi.rs:15         thread::spawn(move || {
pi.rs:16             for j in lower..upper {
pi.rs:17                 let x: f64 = (j as f64 + 0.5) * STEP;
pi.rs:18                 *pi_ref += 4.0/(1.0 + x*x) * STEP;
pi.rs:19             }
pi.rs:20         })
pi.rs:14:32: 14:39 note: but, the lifetime must be valid for the expression at 14:31...
pi.rs:14         let pi_ref: &mut f64 = &mut pi;
                                        ^~~~~~~
pi.rs:14:32: 14:39 note: ...so that reference is valid at the time of borrow
pi.rs:14         let pi_ref: &mut f64 = &mut pi;
                                        ^~~~~~~
error: aborting due to previous error
```

Rust Lösung B

Rust
Florob
Meta
Ownership
Einführung
Beispiel
Features
ADTs
Patterns
Iterators
Unsafe

```rust
use std::sync::Mutex;
use std::thread;

fn main() {
    const NUM_THREADS: u64 = 6;
    const NUM_STEPS: u64 = 100_000;
    const THREAD_STEPS: u64 = NUM_STEPS / NUM_THREADS;
    const STEP: f64 = 1.0 / NUM_STEPS as f64;

```

Rust Lösung B


```
10      let pi: Mutex<f64> = Mutex::new(0.0f64);
11
12      let guards: Vec<_> = (0..NUM_THREADS).map(|i| {
13          let lower: u64 = THREAD_STEPS * i;
14          let upper: u64 = THREAD_STEPS * (i+1);
15          let pi_ref = &pi;
16          thread::spawn(move || {
17              for j in lower..upper {
18                  let x: f64 = (j as f64 + 0.5) * STEP;
19                  let mut lock = pi_ref.lock().unwrap();
20                  *lock += 4.0/(1.0 + x*x) * STEP;
21              }
22          })
23      }).collect();
```

Rust Lösung B


```
24
25      for g in guards { g.join().unwrap(); }
26
27      println!("Pi = {:.10}", *pi.lock().unwrap());
28  }
```

Rust Lösung B


```
$ rustc pi_mutex.rs
pi_mutex.rs:12:47: 23:6 error: closure may outlive the current function, but it borrows `pi`, which is
  ↪   owned by the current function [E0373]
pi_mutex.rs:12     let guards: Vec<_> = (0..NUM_THREADS).map(|i| {
pi_mutex.rs:13         let lower: u64 = THREAD_STEPS * i;
pi_mutex.rs:14         let upper: u64 = THREAD_STEPS * (i+1);
pi_mutex.rs:15         let pi_ref = &pi;
pi_mutex.rs:16         thread::spawn(move || {
pi_mutex.rs:17             for j in lower..upper {
                   ...
pi_mutex.rs:15:23: 15:25 note: `pi` is borrowed here
pi_mutex.rs:15         let pi_ref = &pi;
                                    ^~
pi_mutex.rs:12:47: 23:6 help: to force the closure to take ownership of `pi` (and any other referenced
  ↪   variables), use the `move` keyword, as shown:
pi_mutex.rs:       let guards: Vec<_> = (0..NUM_THREADS).map(move |i| {
pi_mutex.rs:           let lower: u64 = THREAD_STEPS * i;
pi_mutex.rs:           let upper: u64 = THREAD_STEPS * (i+1);
pi_mutex.rs:           let pi_ref = &pi;
pi_mutex.rs:           thread::spawn(move || {
pi_mutex.rs:               for j in lower..upper {
                   ...
error: aborting due to previous error
```

Rust Lösung C


```rust
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    const NUM_THREADS: u64 = 6;
    const NUM_STEPS: u64 = 100_000;
    const THREAD_STEPS: u64 = NUM_STEPS / NUM_THREADS;
    const STEP: f64 = 1.0 / NUM_STEPS as f64;

```

Rust Lösung C


```
10      let pi: Arc<Mutex<f64>> = Arc::new(Mutex::new(0.0f64));
11
12      let guards: Vec<_> = (0..NUM_THREADS).map(|i| {
13          let lower: u64 = THREAD_STEPS * i;
14          let upper: u64 = THREAD_STEPS * (i+1);
15          let pi_ref = pi.clone();
16          thread::spawn(move || {
17              for j in lower..upper {
18                  let x: f64 = (j as f64 + 0.5) * STEP;
19                  let mut lock = pi_ref.lock().unwrap();
20                  *lock += 4.0/(1.0 + x*x) * STEP;
21              }
22          })
23      }).collect();
```

Rust Lösung C


```
24
25      for g in guards { g.join().unwrap(); }
26
27      println!("Pi = {:.10}", *pi.lock().unwrap());
28  }
```

Rust Lösung C


```
1  $ rustc -O pi_arc.rs
2  $ ./pi_arc
3  Pi = 3.1415126520
```

Gliederung

Meta

Ownership
Einführung
Beispiel

Features
ADTs
Patterns
Iterators
Unsafe

1. Meta
2. Ownership
   - Einführung
   - Beispiel
3. Features
   - Algebraische Datentypen
   - Pattern Matching
   - Iterators
   - Unsafe

Gliederung

Meta

Ownership

Einführung

Beispiel

Features

ADTs

Patterns

Iterators

Unsafe

1 Meta

2 Ownership
- Einführung
- Beispiel

3 Features
- Algebraische Datentypen
- Pattern Matching
- Iterators
- Unsafe

Tuple


```
1 // u32 × &str
2 let x: (u32, &str) = (42, "Marvin");
3 println!("{}", x.0);
4 println!("{}", x.1);
```

- Struktureller Produkttyp
- Methoden implementiert für bis zu 12-stellige Tupel

Struct

Rust
Florob
Meta
Ownership
Einführung
Beispiel
Features
ADTs
Patterns
Iterators
Unsafe

```rust
// u32 × &str
struct Person {
        age: u32,
        name: &'static str
}
let x: Person = Person { age: 42, name: "Marvin" };
println!("{}: {}", x.name, x.age);
```

- Produkttyp mit Feldnamen

Enum

Rust
Florob
Meta
Ownership
Einführung
Beispiel
Features
ADTs
Patterns
Iterators
Unsafe

```rust
// C-like
enum Dir {
    North,
    East,
    South,
    West
}
let d: Dir = Dir::East;
```

```rust
// with associated data
enum Shape {
    Rect { x: f32, y: f32 },
    Circ { r: f32 }
}
let c: Shape = Shape::Circ {
    r: 23.0
};
```

- Summentyp
- ähnlich zu tagged Unions

Beispiel `Option`


```
1 enum Option<T> {
2     None,
3     Some(T)
4 }
```

- äquivalent zu Haskells Maybe Monade (`return = Some`, `bind = and_then`)
- statt NULL-Zeigern, nil-Objekten, etc.

Beispiel Option

Rust
Florob
Meta
Ownership
Einführung
Beispiel
Features
ADTs
Patterns
Iterators
Unsafe

```
1  let v: Vec<u32> = ...;
2
3  v.last().and_then(|l| if l < 3 { Some(l+4) } else { None })
4          .and_then(|l| if l > 12 { Some(l-4) } else { None })
5          .unwrap_or(42);
```

Gliederung

Meta

Ownership
Einführung
Beispiel

Features
ADTs
Patterns
Iterators
Unsafe

1. Meta
2. Ownership
   - Einführung
   - Beispiel
3. Features
   - Algebraische Datentypen
   - Pattern Matching
   - Iterators
   - Unsafe

match

Rust
Florob
Meta
Ownership
Einführung
Beispiel
Features
ADTs
Patterns
Iterators
Unsafe

```
1  let x = 2u32;
2
3  match x {
4      1 => "One",
5      2 | 3 => "Twree",
6      5..9 => "Large small number",
7      _ => "Fallthrough"
8  }
```

match

Rust
Florob
Meta
Ownership
Einführung
Beispiel
Features
ADTs
Patterns
Iterators
Unsafe

```
1  let d = Dir::East;
2
3  match d {
4      Dir::North => println!("Northwards!")
5      Dir::East => println!("Go Anti-West!")
6      Dir::South => println!("Southwards!")
7      Dir::West => println!("Go West!")
8  }
```

match


```
1  let c = Shape::Circ { r: 1.0 };
2
3  match c {
4      Shape::Rect { x, y } => println!("{} x {}", x, y),
5      Shape::Circ { r } => println!("{}", r)
6  }
```

let

Rust
Florob
Meta
Ownership
Einführung
Beispiel
Features
ADTs
Patterns
Iterators
Unsafe

```rust
let Person { age, name } = marv;
let (x, y) = point;
let Person { age: alter, name: vorname } = marv;
```

if let

Rust
Florob
Meta
Ownership
Einführung
Beispiel
Features
ADTs
Patterns
Iterators
Unsafe

```
1 if let Ok(dir) = std::env::var("HOME") {
2     println!("Home dir is {}", dir);
3 }
```

Matching function parameters

Rust
Florob
Meta
Ownership
Einführung
Beispiel
Features
ADTs
Patterns
Iterators
Unsafe

```
1  fn print_person(&Person { age, name }: &Person) {
2      println!("{}: {}", name, age);
3  }
```

Gliederung

Meta

Ownership
Einführung
Beispiel

Features
ADTs
Patterns
Iterators
Unsafe

1. Meta
2. Ownership
   - Einführung
   - Beispiel
3. Features
   - Algebraische Datentypen
   - Pattern Matching
   - Iterators
   - Unsafe

Iterators

Rust
Florob
Meta
Ownership
Einführung
Beispiel
Features
ADTs
Patterns
Iterators
Unsafe

- Objekte die eine `.next()` Methode besitzen
- `for`-Schleifen sind syntactic sugar für wiederholtes Aufrufen von `.next()` bis es `None` zurück gibt

Adaptors

Rust
Florob
Meta
Ownership
Einführung
Beispiel
Features
ADTs
Patterns
Iterators
Unsafe

```
1  fn main() {
2      let mult = (1..).filter(|x| (1..11).all(|y| x % y == 0))
3                      .next().unwrap();
4      println!("{}", mult);
5  }
```

- Finde die kleinste Zahl die von jeder Zahl von 1 bis 10 geteilt wird

Gliederung

Meta

Ownership
Einführung
Beispiel

Features
ADTs
Patterns
Iterators
Unsafe

1. Meta
2. Ownership
   - Einführung
   - Beispiel
3. Features
   - Algebraische Datentypen
   - Pattern Matching
   - Iterators
   - Unsafe

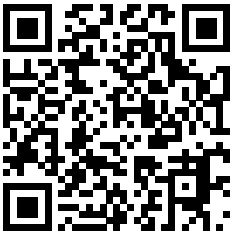
Unsafe

Rust
Florob
Meta
Ownership
Einführung
Beispiel
Features
ADTs
Patterns
Iterators
Unsafe

```
1 pub fn u64_le_array(x: u64) -> [u8; 8] {
2     unsafe { std::mem::transmute(x.to_le()) }
3 }
```

- Nötig um einige sichere Abstraktionen zu erstellen
- ändert **nicht** die Sprachsemantik
- erlaubt das Aufrufen von als `unsafe` markierten Funktionen
- z. B. Pointerarithmetik, Derferenzieren von raw Pointern

Danke für die Aufmerksamkeit.
Fragen?



http://babelmonkeys.de/~florob/talks/RUG-2015-10-28-Rust.pdf