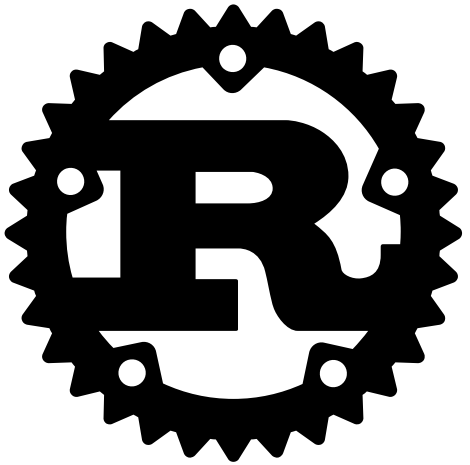
De Rust programmeertaal



Peter Scholtens

Linux
User Group
Nijmegen

Wat is Rust?

"Rust is a systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety. "

[www.rust-lang.org]

Begonnen als project van Graydon Hoare, sinds 2009 gesponsord door Mozilla Research

- Gecompileerde taal, dus snel
- Geschreven uit oogpunt van veiligheid:
  geen null-pointers crashes, geen buffer overrun.
- Gemakkelijker te paralleliseren
- Verfijnd door ervaringen met Servo layout engine
- Self compiling sinds 2011
- https://en.wikipedia.org/wiki/Rust_%28programming_language%29

Waar komt de naam vandaan?

Genoemd naar schimmels,...

https://en.wikipedia.org/wiki/Rust_%28fungus%29

```
<graydon> IOW I don't have a really good explanation. it seemed like a
          good name. (also a substring of "trust", "frustrating", "rustic" and …
          "thrust"?)
<graydon> I think I named it after fungi. rusts are amazing creatures.
<graydon> Five-lifecycle-phase heteroecious parasites. I mean, that's just _crazy_.
<graydon> talk about over-engineered for survival
<jonanin> what does that mean? :]
<graydon> fungi are amazingly robust
<graydon> to start, they are distributed organisms. not single cellular, but
          also no single point of failure.
```

Lees hier verder :-)

https://www.reddit.com/r/rust/comments/27jvdt/internet_archaeology_the_definitive_endall_source/

Wat zijn de voordelen?

Voordelen:

- ✔ Veel betere veiligheid mbt. geheugentoewijzing
- ✔ Gemakkelijk te paralleliseren
- ✔ Cross platform (Linux, Mac, Windows)
- ✔ Niet geschreven door één groot bedrijf (Go, Swift, C#), maar door vele *onafhankelijke* bijdragen uit een actieve gemeenschap
  - ✔ Elke keuze wordt goed afgewogen
  - ✔ Geen plotselinge veranderingen
  - ✔ Grote beschikbaarheid van diverse bibliotheken

Nadelen:

- ✗ Je moet weer een nieuwe taal leren ;-)
- ✗ Iets vreemdere syntax: C-like `sin(x);` Rust style `x.sin();`
- ✗ Je moet (opnieuw) leren nadenken over 'ownership' van variabelen

Hoe leer je een nieuwe taal?

Gewoon online proberen:

https://www.rust-lang.org

http://rustbyexample.com/hello.html

https://play.rust-lang.org

Werkt iets niet zoals je verwacht? Zoek een vergelijkbaar probleem (en vindt zo de oplossing):

http://stackoverflow.com/questions/tagged/rust

https://www.reddit.com/r/rust

Verdeel je tijd. Probeer niet in een weekend alles te begrijpen, laat je met een dagelijkse e-mail verrassen door andermans vragen:

http://stackexchange.com/filters/12504/rust

Hoe leer je een nieuwe taal (2) ?

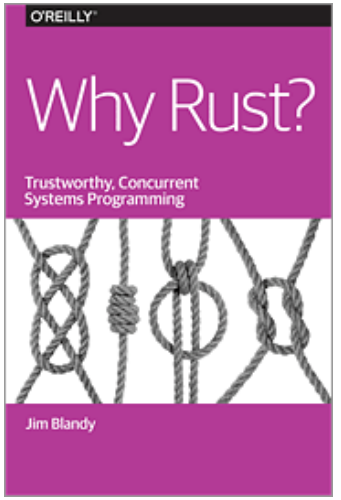
Boeken:

http://search.oreilly.com/?q=rust

De handeling “Programming Rust” verschijnt in oktober 2016,

Argumentatie waarom Rust ontwikkeld is (gratis e-boek)

http://www.oreilly.com/programming/free/why-rust.csp



Video's:

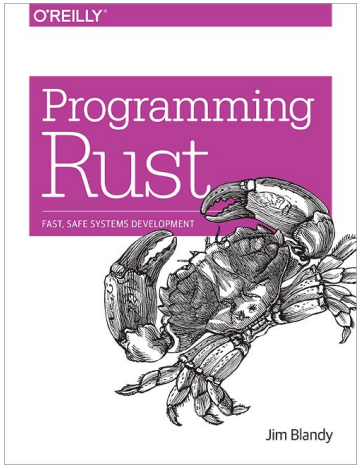
https://air.mozilla.org/introduction-to-rust/

https://air.mozilla.org/guaranteeing-memory-safety-in-rust/

https://www.youtube.com/results?search_query=rust+programming+language

https://youtu.be/bsVRuwJC15Y

https://www.youtube.com/watch?v=oD1_3iL12mU



"The most important property languages have, is changing the way how you think"

Voorbeeld 1: hergebruik variabele

```
1   fn main() {
2       let a = 3.141f64;
3       let b = 2.717f64+a;
4       a = a + 1.0;
5       println!("{}",a);
6       println!("{}",b);
7   }
```

Maar wat mag hier *niet* in Rust?

https://play.rust-lang.org

Hergebruik (variable shadowing)[rbe4.2]

```
1         fn main() {
2           let long_lb = 1;
3           {
4             let short_lb = 2;
5             println!("inner short: {}", short_lb);
6 (a)         let long_lb = 5_f32;
7             println!("inner long: {}", long_lb);
8           }
9 (b)       println!("outer short: {}", short_lb);
10(c)       println!("outer long: {}", long_lb);
11(d1)      let long_lb = 'a';
12(d2)      let long_lb = (long_lb+1).to_string();
13          println!("outer long: {}", long_lb);
14        }
```

Voorbeeld 2: functie-aanroep

```
 1   struct Complex {
 2       real: f64,
 3       imag: f64,
 4   }
 5
 6   fn add(a: Complex, b: Complex) -> Complex {
 7       Complex{ real:(a.real + b.real), imag:(a.imag +
 8   b.imag)}
 9   }
10
11   fn main() {
12       let a = Complex { real:  1.414, imag: 1.414 };
13       let b = Complex { real: -0.707, imag: 0.707 };
14       let c = add(a,b);
15       let d = add(a,c);
16       println!("({}+{}i)",c.real, c.imag);
17   }
```

Drie manieren om uit te lenen: *weggeven*

```
fn myfunction (vec: Vec<i32>) { ... }
```

In de body van deze functie:

- Mag je de meegegeven variabele lezen
- Mag je de variabele wijzigen
- Mag je de variable vernietigen

Na het verlaten van deze functie *wordt* de variabele vernietigd (tenzij…)

Andere threads kunnen de variabele **niet** zien!

Drie manieren… : *alleen lezen* uitlenen

```
fn myfunction (vec: &Vec<i32>) { ... }
```

In de body van deze functie:

- Mag je de meegegeven variabele lezen
- Mag je de variabele **niet** wijzigen
- Mag je de variable **niet** vernietigen

Na het verlaten van deze functie *blijft* de variabele behouden

Andere threads mogen deze variabele ook lezen! (maar niet overschrijven of overnietigen)

Drie manieren… : *lezen en schrijven*

```
fn myfunction (vec: &mut Vec<i32>) { ... }
```

In de body van deze functie:

- Mag je de meegegeven variabele lezen
- Mag je de variabele wijzigen
- Mag je de variable vernietigen

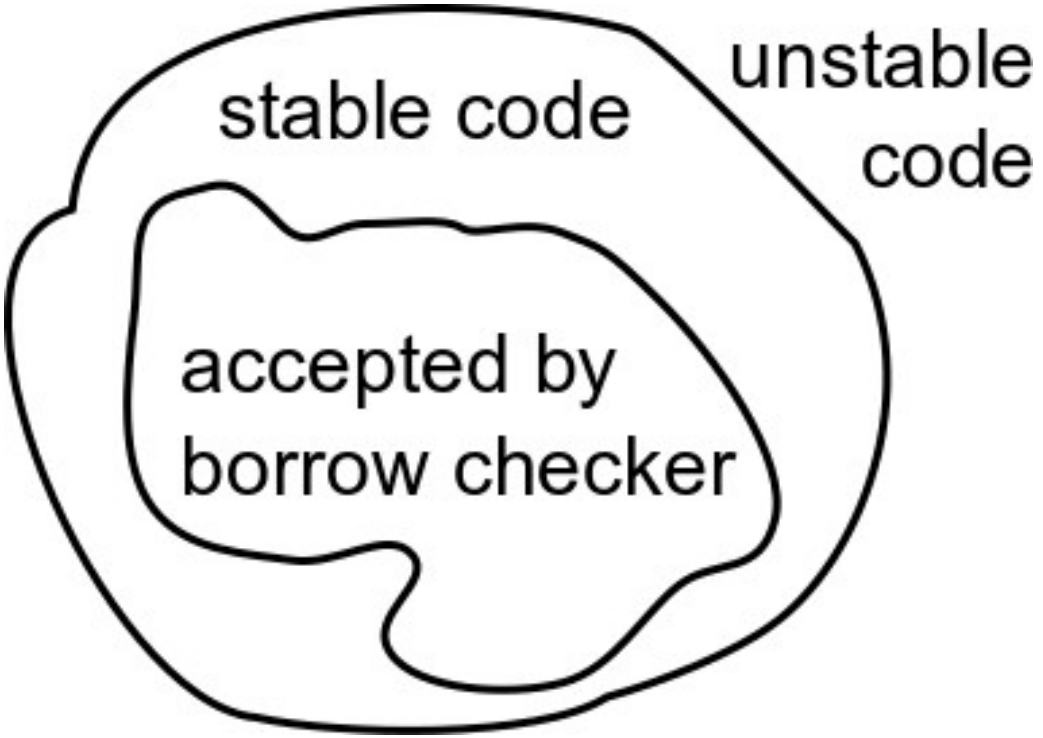
Na het verlaten van deze functie *blijft* de variabele bestaan (tenzij je 'm in de functie vernietigd hebt)

Andere threads kunnen de variabele (tijdelijk) niet zien!

Borrow checker

- Muteerbaarheid wordt in compile time gecontroleerd. Kost dus geen runtime performance.
- Aliases naar niet meer bestaande objecten (dangling pointers) worden ook gefilterd.
- Onge-initialiseerde pointers… mogen ook niet!
- Geen buffer overflows, controle in run-time (zie verder op)
- Dankzij deze geheugenveiligheid → gemakkelijk te paralleliseren

Niet te beperkend?



- De borrow checker is beperkt
- Bij een volgende release weer ietsje groter bereik
- Als je het echt zeker weet `unsafe {…}`

Voorbeeld 3: Controle op initialisatie

```
fn add(self, other: Fraction) -> Fraction {
    let m;
    ....
    if self.polarity == other.polarity
    {
        m = self.mantissa + other.mantissa;
        ....
    }
    else
    {
        ....
        if tmp2 < tmp3
        {
            ....
        }
        else
        {
            ....
        m = 0;
        }
        ...
    }
    let mut result = Fraction { polarity:p, mantissa:m, numer:n, denom:d };
    result.remove_gcd();     // Do reduction if possible.
    return result;
}
```

Voorbeeld4: multi-assignment

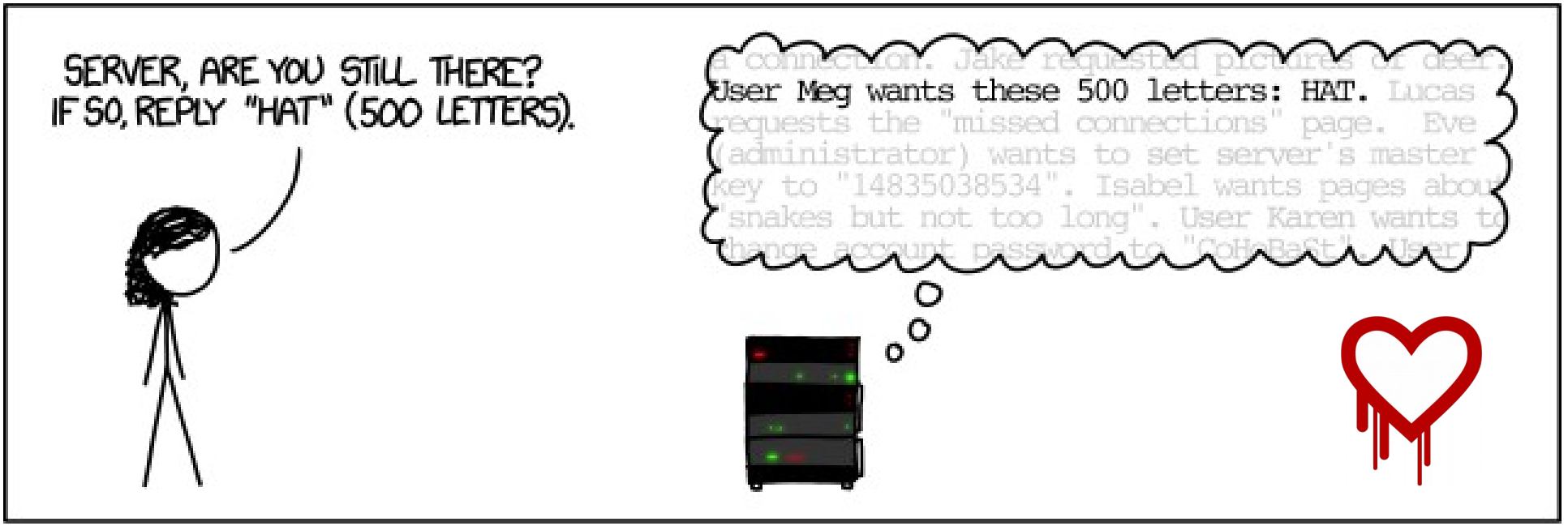
```
struct Point {
    x: f64,
    y: f64,
}

struct Rectangle {
    p1: Point,
    p2: Point,
}

fn area(r: &Rectangle) -> f64 {
    let Point { x: x1, y: y1 } = r.p1;
    let Point { x: x2, y: y2 } = r.p2;
    ((x1 - x2) * (y1 - y2)).abs()
    }

fn main() {
    let q = Rectangle {
        p1: Point {x: 2.0, y: 4.0},
        p2: Point {x: 3.0, y: 1.0},
    };
    let a = area(&q);
    println!("(Oppervlak is {})",a);
}
```

Is dit te voorkomen?



https://xkcd.com/1354/

“There is a moral consequence for memory safety...”

Voorbeeld 5: Iterator

```
fn main ()
{
  let array = [1u32,7,2,4,8,3]; // zonder index te printen

  for i in array.iter() {
    println!("> {}", i);
  }

  let array_len = array.len();// eerste poging
  for i in 0..array_len {
    println!("[{}]> {}", i, array[i]);
  }

  for it in array.iter().enumerate() { // Rust's style!
    let (i, a) = it;
    println!("[{}]> {}", i, a);
  }
}
```

Nog meer…

- Fraai voorbeeld hoe je iets parallels programmeert:

http://doc.rust-lang.org/book/dining-philosophers.html

- Vergelijking snelheid Rust en C++

http://cantrip.org/rust-vs-c++.html

“Snel” en “traag” in orde van 1.3 tot 2x, geen 100x

Conclusie

- Rust is een nieuwe programmeertaal met een snel groeiende gebruikers gemeenschap.
- Rust heeft diverse mechanismen om veel voorkomende fouten uit te sluiten:
  - Controle op mutability en aliasing (dangling pointers)
  - Check op initialisatie in `if`, `match` (some, none) etc.
  - Voorkomt bereiksfouten met Array/Vector iterators etc.
- Makkelijk te paralleliseren,… handig als zelf je mobiele telefoon 4 cores heeft
- Default is veilig, maar als je wilt kan het `unsafe`
- Nog meer:
  - Goede interfaces met andere talen FFI
  - Cargo (make+depencies+..), macro's?