# Wat is de huidige status van Rust in webdevelopment?

Branco Bruyneel

18 mei 2022

# Woord vooraf

# Abstract

# Inhoudsopgave

# Lijst van figuren

# Hoofdstuk 1

# Research

## 1.1 Wat is Rust?

Rust is een gecompileerde multi-paradigma programmeertaal bedacht door Graydon Hoare en is begonnen als een project van Mozilla Research. Geïnspireerd door de programmeertalen C en C++, maar kent weinig syntactische en semantische gelijkenissen tegenover deze talen. Rust focust zich met name op snelheid, veiligheid, betrouwbaarheid en productiviteit. Dit wordt gerealiseerd door gebruik te maken van een krachtig typesysteem en een borrow checker. Hiermee kan Rust een hoog niveau van geheugenveiligheid garanderen zonder een garbage collector nodig te hebben. Rust beoogt moderne computersystemen efficiënter te benutten. Hiervoor maakt het onder meer gebruik van geheugenbeheer dat geheugen in een blok toewijst en daarnaast strikt toeziet op de stacktoewijzing. Hierdoor kunnen problemen zoals stackoverflows, bufferoverflows en niet-geïnitialiseerd geheugen voorkomen worden. Verder staat Rust ook geen null-pointers, dangling-pointers of data-races toe in veilige code.

### 1.1.1 Syntax

Voor velen die zich niet herkennen in programmeertalen zoals C++, Haskell of OCaml lijkt Rust een aparte syntax te hebben in tegenstelling tot conventionele talen. Laat ons even kijken naar een paar syntactische voorbeelden in Rust.

Hier een simpel voorbeeld dat `"Hello world!"` schrijft naar de standaard output.

```
1  fn main() {
2    println!("Hello, world!");
3  }
```

**Listing 1:** Hello, world!

Merk op dat `println!` geen functie is maar een macro. Geïnspireerd door de functionele programmeertaal Scheme, zijn macro's een manier van code schrijven dat andere code schrijft, wat bekend staat als metaprogramming. Metaprogramming is handig voor het verminderen van code dat u zelf hoeft te schrijven en onderhouden, wat ook een van de rollen is van functies. Toch verschillen macro's met functies. Zo kunnen macro's een variabel aantal parameters hebben en worden ze uitgebreid vooraleer de compiler de betekenis van de code interpreteert.

```
1   #[derive(Debug)]
2   struct Rectangle {
3     width: u32,
4     height: u32,
5   }
6
7   impl Rectangle {
8     fn square(size: u32) -> Rectangle {
9       Rectangle: {
10        width: size,
11        height: size,
12      }
13    }
14    fn area(&self) -> u32 {
15      self.width * self.height
16    }
17  }
18
19  fn main() {
20    let rect1 = Rectangle::square(4);
21    println!(
22      "The area of the rectangle is {} square pixels.",
23      rect1.area()
24    );
25  }
26
```

**Listing 2:** structs

Een `struct` in Rust is gelijkaardig als een Object in object georiënteerde programmeertalen. Het wordt gebruikt om samenhorende waarden te groeperen en optioneel kan men associated functions implementeren. Associated functions die geen self als hun eerste parameter hebben zijn geen methodes en kunnen gebruikt worden als constructors die een nieuwe instantie retourneren van de struct.

Rust heeft geen null pointers tenzij men een null pointer wil dereferentieren (dan moet die in een `unsafe` blok worden geplaatst). Als alternatief voor null maakt Rust gebruik van een `Option` type waarmee gekeken kan worden of een pointer wel `Some` of geen `None` waarde bevat. Dit kan afgehandeld worden door syntactische sugar, zoals het if let statement om toegang te krijgen tot het innerlijke type, in dit geval een string:

```
fn main() {
  let name: Option<String> = None;
  // If name was not None, it would print here.
  if let Some(name) = name {
    println!("{}", name);
  }
}
```

**Listing 3:** `Option` type

Naast de `if` en `else` controlestructuren is er ook match en `if let`. match is vergelijkbaar met een `switch` statement uit andere talen. Het neemt een waarde en test het tegen een serie van patronen. Op basis van welk patroon er overeenkomt wordt de code uitgevoerd. Patronen kunnen opgemaakt worden uit waarden, variabel namen, wildcards, en veel meer. De power van match komt van het feit dat de compiler zal bevestigen bij het compileren dat alle mogelijke gevallen zijn afgehandeld. Soms wil je niet alle gevallen expliciet afhandelen en wil je slechts een patroon afhandelen terwijl je de rest negeert. In dat geval kan je `if let` gebruiken, wat minder boilerplate code is dan match.

```
let message = match maybe_digit {
  Some(x) if x < 10 => process_digit(x),
  Some(x) => process_other(x),
  None => panic!(),
};

let config_max = Some(3u8);
if let Some(max) = config_max {
  println!("The maximum is configured to be {}", max);
}
```

**Listing 4:** `if let` & `match` operators

### 1.1.2 Geheugen

Bij vele programmeertalen hoef je geen zorgen te maken over het geheugen gebruik. Dit is mogelijk door een garbage collector te gebruiken die voortdurend zoekt naar niet langer gebruikt geheugen terwijl het programma loopt. In andere talen, moet de programmeur het geheugen expliciet toewijzen en vrijmaken. Rust gebruikt geen van beide methodes en komt met een uniek concept genaamd ownership. Hiermee kan het geheugen veiligheid garanderen zonder een garbage collector nodig te hebben.

Vooraleer we verder gaan is het belangrijk dat we de twee begrippen genaamd stack en heap begrijpen. De twee datastructuren maken deel uit van het geheugen en zijn beschikbaar voor uw code om te gebruiken tijdens runtime, maar ze zijn op verschillende manieren gestructureerd. Wat maakt dat de ene zorgt voor snellere dataopslag dan de andere.

De stack slaat waarden op in de volgorde waarin hij ze krijgt en verwijdert de waarden in de omgekeerde volgorde. Dit wordt aangeduid als last in, first out. Alle gegevens die op de stack worden opgeslagen moeten een bekende, vaste grootte hebben. Gegevens waarvan de grootte op het moment van compileren onbekend is of die van grootte kunnen veranderen, moeten in plaats daarvan op de heap worden opgeslagen.

De heap is minder georganiseerd: als je gegevens op de heap zet, vraag je een bepaalde hoeveelheid ruimte aan. De memory allocator vindt een lege plek in de heap die groot genoeg is, markeert die als in gebruik, en geeft een pointer terug, dat is het adres van die locatie. Dit proces wordt “allocating on the heap” genoemd en wordt soms afgekort als gewoon allocating. Het ”pushen”van waarden op de stack wordt niet beschouwd als allocating. Omdat de pointer naar de heap een bekende, vaste grootte heeft, kun je de pointer op de stack opslaan, maar als je de eigenlijke gegevens wilt hebben, moet je de pointer volgen.

Het efficiëntste is dus om data naar de stack weg te schrijven dan naar de heap, omdat de allocator nooit hoeft te zoeken naar een plaats om nieuwe gegevens op te slaan. Die plaats is altijd bovenaan de stack. Het alloceren van ruimte op de heap vergt meer werk, omdat de allocator eerst een ruimte moet vinden die groot genoeg is om de gegevens op te slaan en dan de boekhouding moet doen om de volgende allocatie voor te bereiden.

Rust heeft dus een voorkeur om zijn variabelen weg te schrijven naar de stack. Maar zoals gezegd kan je niet elke variabele wegschrijven naar de stack, daarom is er een onderscheid tussen simpele en complexe types. Bij simpele types ken je de grootte voor het compileren, daarmee worden ze dan ook opgeslagen op de stack. In tegenstelling tot complexe types waarbij de grootte kan veranderen, worden ze opgeslagen in de heap.

De simpele types zijn:

- Integer
- Floating-point
- Boolean
- Character
- Tuple
- Array (ze hebben dus een vaste grootte in Rust)

Nu de begrippen zijn opgeklaard kunnen we kijken naar het ownership systeem. Het systeem bestaat uit drie regels:

1. Elke waarde in Rust heeft een variabele die de owner wordt genoemd
2. Er kan maar een owner per keer zijn
3. Wanneer de owner buiten scope gaat, zal de waarde worden verwijderd

Deze regels worden gecontroleerd bij het compileren. Als een van de regels wordt overtreden zal het programma niet compileren.

Laat ons eens kijken naar een simpel voorbeeld.

```
1  {
2    let s = String::from("hello"); // s is geldig vanaf deze lijn
3    // doe iets met s
4  } // hier eindigt de scope, en s is niet langer geldig
```

**Listing 5:** ownership

Als we de regels volgen, wordt de variabele s owner over de string literal `hello`. De variabele `s` blijft geldig zolang hij binnen de scope wordt aangeroepen. Op het einde van de scope zal Rust de `drop` functie uitvoeren en dus het geheugen terug vrijgeven. Dit is de basis van hoe het ownership systeem werkt in Rust. Dit was een zeer simpel voorbeeld en in de realiteit komen er natuurlijk nog wat eigenaardigheden bij kijken.

### 1.1.3 Ecosysteem

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

## 1.2 Wat is WebAssembly & Hoe werkt het?

Om interactieve webapps te creëren gebruik je tegenwoordig javascript. Ondanks de succesvolle inspanningen van browsermakers om hun javascript-engines in elke versie weer wat efficiënter te maken, is dat voor veel toepassingen nog niet genoeg. Google dan kwam in 2011 met Native Client (NaCI), een sandbox voor het efficiënt en veilig uitvoeren van gecompileerde C en C++ code in de browser, onafhankelijk van het besturingssysteem van de gebruiker. Het bracht prestaties en low-level controle van native code naar moderne webbrowsers, zonder de veiligheid en portabiliteit van het web op te offeren. [1]

Mozilla wilde de platformonafhankelijkheid van javascript echter niet verlaten, en begon daarom in 2013 aan een andere aanpak: asm.js, een subset van javascript die browsers heel efficiënt kunnen uitvoeren. Je compileert dan een webapp uit een taal zoals C naar asm.js, en je browser voert dit dan als gewone javascript uit.

Het voordeel van asm.js is dat het gewoon al in alle webbrowsers werkte, maar Mozilla botste tegen de snelheidsgrenzen van javascript aan. Omdat javascript een tekstformaat heeft, vraagt het parsen veel rekenkracht, zeker op mobiele toestellen met een wat zwakkere processor. En zo werd in 2015 WebAssembly (afgekort Wasm) geboren, een binair instructieformaat voor een stack-gebaseerde virtuele machine in je webbrowser. Het is ontworpen als een overdraagbaar compilatiedoel voor programmeertalen, waardoor het gebruik op het web mogelijk wordt voor client- en servertoepassingen. [2] [3]

### 1.2.1 Doel

Het is dus geen nieuwe programmeertaal, maar een binair formaat voor uitvoerbare programma's. Het wordt gecreëerd als een open standaard binnen de W3C WebAssembly Community Group met de volgende doelstellingen:

- Snel, efficiënt en overdraagbaar - wasm code kan op bijna-native snelheid worden uitgevoerd op verschillende platforms door gebruik te maken van gemeenschappelijke hardware mogelijkheden.
- Leesbaar en foutopspoorbaar - wasm is een lage assembleertaal, maar het heeft een menselijk leesbaar tekstformaat (aan de specificatie wordt nog gewerkt) waarmee code met de hand kan worden geschreven, bekeken en foutopsporing mogelijk is.
- Veilig - wasm is gespecificeerd om te worden uitgevoerd in een veilige, sandboxed omgeving. Net als andere webcode, zal het de browser's same-origin en permissies beleid afdwingen.
- Maak het web niet kapot - wasm is zo ontworpen dat het goed samengaat met andere webtechnologieën en achterwaartse compatibiliteit behoudt.

### 1.2.2 Hoe werkt het?

Nu we weten wat wasm op een hoog niveau inhoudt, is het ook goed om eens praktisch te kijken hoe het precies werkt. Er zijn namelijk een aantal opties voor het compileren naar wasm:

- C/C++ applicatie omzetten naar wasm met Emscripten
- wasm rechtstreeks op assembly niveau schrijven of genereren
- een Rust applicatie schrijven en wasm als compilatie target gebruiken
- AssemblyScript gebruiken, wat vergelijkbaar is met Typscript en compileert naar een wasm binary

In deze bachelorproef werd het technisch onderzoek uitgevoerd in Rust. Met zijn kleine runtime, betrouwbaar en rijk typesysteem is het een van de populairste keuzes voor het

bouwen van webapps met WebAssembly. Dus laat ons even kijken naar een voorbeeld hoe we vanuit Rust javascript functies kunnen gebruiken en andersom.

Een van de moeilijkste onderdelen van het werken met WebAssembly is om verschillende soorten waarden in en uit functies te krijgen. Dat komt omdat WebAssembly momenteel slechts twee types kent: integers en floating point getallen.

Dit betekent dat je niet zomaar een string in een WebAssembly functie kunt stoppen. In plaats daarvan moet je een aantal stappen doorlopen om een string voor te stellen als getallen. Als je complexere types hebt, zul je zelfs een ingewikkelder proces hebben om de gegevens heen en weer te sturen. Gelukkig bestaat er de library wasm-bindgen die deze stappen voor ons doet. Met een paar annotaties aan je Rust code, zal het automatisch de code maken die nodig is (aan beide kanten) om complexere types te laten werken.

## 1.3 Welke front- & backend frameworks zijn er ter beschikking?

## 1.4 Hoe bouw je een web frontend in Rust?

## 1.5 Hoe bouw je een API in Rust?

## 1.6 Is Rust productie klaar?

# Hoofdstuk 2

# Abstract

Samenvatting of abstract (mag in het Engels): MAX 1 halve A4-pagina:

Je beantwoordt in de samenvatting kort en bondig een viertal vragen:

- Wat is de onderzoeksvraag?
- Wat was jouw onderzoek?
- Welke elementen spelen een grote rol (zowel positief als negatief) bij de evaluatie van het onderzoek?
- Welke elementen zijn belangrijk bij jouw advies?
- Het besluit wordt kort samengevat.

# Hoofdstuk 3

# Inhoudsopgave

# Hoofdstuk 4

# Figurenlijst

In de figurenlijst staan alle figuren die je in je bachelorproef gebruikt opgesomd. Wanneer je van de functie 'Bijschrift invoegen...' in Word gebruik maakt, kun je de lijst automatisch genereren.

## Hoofdstuk 5

# Lijst met afkortingen

# Hoofdstuk 6

# Verklarende woordenlijst