Rust i napredni tipski sustavi

1. Uvod

Ivan Radiček

13.03.2021.

Što je i zašto Rust?

- Sistemski **programski jezik**, naglasak na performansama
  - slično kao C++
- Više paradigmi
  - Funkcijska, imperativna, ...
- Automatsko upravljanje memorijom kroz napredni **tipski sustav**
  - bez GC[1]-a
- *"The 2020 Developer Survey results are in, and once again, Rust is the number one most loved language among the 65,000 programmers who participated. Rust has taken the number one spot since 2016, showing that it's got something that the developers who use it love."* - https://stackoverflow.blog/2020/06/05/why-the-developers-who-use-rust-love-it-so-much/

[1] Garbage collection

Programski jezici

- Formalni jezik koji opisuje način upravljanja računalnim procesima
- Zadan kroz **sintaksu** (*forma*) i **semantiku** (*značenje*)

Sintaksa programskog jezika

- Izvorni kod programa je obično string (tekst, niz znakova)
- Sintaksa jezika određuje koji stringovi su valjani u jeziku

Sintaksa jednostavnog kalkulatora

- Neka je L_k jezik kalkulatora, tada L_k sadrži:
  - n (gdje je n tekstualna reprezentacija broja, npr. 3.14)
  - Ako su e_1 i e_2 u L_k, onda su i $e_1 + e_2$, $e_1 \cdot e_2$, $e_1 - e_2$, ...
- Kratko se to još može zapisati:
  $e \ := \ n \ | \ e_1 + e_2 \ | \ e_1 \cdot e_2 \ | \ e_1 - e_2 \ | \ \cdots$
- Iz toga možemo zaključiti za sljedeće stringove:
  - 3.3 (ispravan)
  - $3.14 + 21 * 42 - 1$ (ispravan)
  - $3.14 * * * 42$ (neispravan niz simbola)
- Sintaksa ne govori ništa o značenju, samo da li je niz simbola sintaktički ispravan!

Različite sintakse

- Python: https://docs.python.org/3/reference/grammar.html
- Whitespace: https://en.wikipedia.org/wiki/Whitespace_(programming_language)#Syntax
- Brainfuck: https://en.wikipedia.org/wiki/Brainfuck#Commands

Semantika programskog jezika

- Određuje **značenje** *sintaksno ispravnog* programa
- Preciznije, mapira (ispravan) niz znakova u njegovo značenje (domenu rezultata)
- Formalno bi mogli zapisati da je to funkcija: $String \to D$
- Npr. za kalkulator $D = \mathbb{R}$ (realni brojevi)
- Semantika kalkulatora je jednostavna aritmetika

Semantika programskog jezika - primjeri

- Whitespace: https://en.wikipedia.org/wiki/Whitespace_(programming_language)#Syntax
- Brainfuck: https://en.wikipedia.org/wiki/Brainfuck#Commands

Semantika programskog jezika - primjer if-then-else

$$\frac{e_1 \Rightarrow \texttt{true} \qquad e_2 \Rightarrow v_2}{\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \Rightarrow v_2} \qquad \frac{e_1 \Rightarrow \texttt{false} \qquad e_3 \Rightarrow v_3}{\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \Rightarrow v_3}$$

$$\frac{2 + 2 = 4 \Rightarrow \texttt{true} \qquad 0 \Rightarrow 0}{\texttt{if } 2 + 2 = 4 \texttt{ then } 0 \texttt{ else } 1 \Rightarrow 0}$$

Tipski sustavi

- Tipski sustavi su logički sustavi koji određuju da li je neki program ispravan u tom sustavu (ima ispravan tip)
- U većini slučajeva se to svodi na provjeru da li program proizvodi rezultat određenog tipa (broj, string, Boolean, objekt određene klase)
- Napredniji sustavi omogućuju i *"snažnije"* svojstva, npr:
  - Gornja granica kompleksnosti (vremena izvođenja) - npr. $\mathcal{O}(n^2)$
  - Ispravno korištenje memorije
  - Logička svojstva programa
  - ...

Zašto su tipski sustavi korisni?

Apsolutni broj

```
1  def f(x):
2    if x < 0: return -x
3    elif x > 0: return x
```

- Koji je rezultat $f(0) + 1$?
- Najjednostavniji tipski sustav bi mogao uhvatiti ovu grešku
- Ovu grešku je lagano uočiti - ali što sa aplikacijom od 100k linija koda gdje lanac pozivanja funkcija više redova veličine?
- No, treba ipak priznati da za male skripte tipovi baš i nisu korisni . . .
- Tj. najkorisniji su kod velike organizacije koda

Python program sa tipovima (restrikcija tipa)

```
1  def f():
2    if True:
3      return 1 + 1
4    else:
5      return 1 + "one"
```

- Gdje je problem?

Plan predavanja

1. Osnove programiranja u Rustu (bez memorije)
2. Programiranje u Rustu sa upravljanjem memorijom
3. Definicija semantike malog programskog jezika (LISP sintaksa)?
4. Pregled osnovnih tipskih sustava
5. Neki primjeri naprednijih tipskih sustava

Rust početak

- Rust *"igralište"* (playground): https://play.rust-lang.org
- Rust knjiga / dokumentacija: https://doc.rust-lang.org/book/

Hello world

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

[Igralište]

Rust: aritmetički izrazi

```
1  fn main() {
2    println!("Rezultat: {}", 80 / 2 + 2)
3  }
```

[Igralište]

Rust: varijable / let-izraz

- Najjednostavnija apstrakcija
- Služi za de-duplikaciju i bolju preglednost

```
1  fn main() {
2      let x = 6;
3      println!("Rezultat: {}", x * x + x)
4  }
```

[Igralište]

Rust: varijable / let-izraz (nastavak)

```
1  fn main() {
2      let x = 6;
3      if x > 0 {
4          let x = 3;
5          println!("x = {}", x);
6      } else {
7          println!("x = {}", x);
8      }
9      println!("x = {}", x);
10 }
```

[Igralište]

- Koji je rezultat?

Rust: mutacija memorije

```
1  fn main() {
2      let mut x = 6;
3      if x > 0 {
4          x = 3;
5          println!("x = {}", x);
6      } else {
7          println!("x = {}", x);
8      }
9      println!("x = {}", x);
10 }
```

[Igralište]

- A sada?
- Mutacije ćemo izbjegavati kad god je to moguće!
- Zašto? Omogućuju promjene koje mogu imati posljedice u dalekim dijelovima programa . . . teško za pratiti

Rust: grananje

- If-else: jednostavna vrsta grananja (true/false)
- Postoji i "*naprednija*" vrsta grananja sa općenitijim tipom podatka od Booleana

```
1  fn main() {
2    if 2 + 2 == 4 { println!("OK!") }
3    else { println!("Nije OK!") }
4  }
```

[Igralište]

Općenito if-else

```
1  if c { e1 } else { e2 }
```

- c - izraz uvjeta (Boolean)
- e_1, e_2 - izrazi grananja, odabir ovisi o c (oba izraza moraju biti istog tipa)

Rust: sve je izraz

- U Rustu je skoro sve izraz (pa i npr. if-else grananje)

```
1  fn main() {
2      let x = -42;
3      let abs_x = if x < 0 { -x } else { x };
4      println!("x={}, |x|={}", x, abs_x)
5  }
```

[Igralište]

Rust: apstrakcija funkcijama

- Jedan od najvažnijih apstrakcijskih alata
- Slično kao matematička funkcija
- Napomena: U Rustu funkcija ne treba eksplicitni **return**, već je zadnji izraz u funkciji i njen rezultat

```
1  fn abs(x : isize) -> isize {
2    if x < 0 { -x } else { x }
3  }
4
5  fn main() {
6    let broj = -42;
7    println!("x={}, |x|={}", broj, abs(broj))
8  }
```

[Igralište]

Rust: jednostavna definicija funkcije

Definicija (jednostavna) funkcije u Rustu

```
1  fn f(x1 : t1, x2 : t2, ..., xn : tn) -> tr {
2     e
3  }
```

- f - ime funkcije
- x_i - ime i-tog parametra
- t_i - tip i-tog parametra
- t_r - tip rezultata funkcije
- e - je izraz tipa t_r koji definira rezultat funkcije
- `isize` je tip cijelog broja čija veličina ovisi o arhitehturi (https://doc.rust-lang.org/book/ch03-02-data-types.html#integer-types)

Rust: kombinacija više izraza

Odvajanje izraza sa ;

```
1  fn poly(x : f64, y : f64) -> f64 {
2    println!("x = {}, y = {}", x, y) ;
3    2.5 * x * x + 1.5 * y + 0.5
4  }
5
6  fn main() {
7    let r = poly(2.1, 3.1);
8    println!("Rezultat: {}", r)
9  }
```

[Igralište]

- ; odvaja 2 ili više izraza
- Rezultat kombinacije je vrijednost zadnjeg izraza
- Obično svi osim zadnjeg izraza nemaju rezultat
- tj. `unit` (*slično kao* `void`) tipa, ali o tome kasnije

Rust: rekurzija

- Rekurzivne funkcije su često elegantnije od iterativnih i dozvoljavaju *čišće* izražavanje ideje
- Treba paziti na performanse (iako je to često nebitno, a i kompajler može pomoći)

Rekurzivna definicija funkcije $x!$

```
 1  fn fact(x : isize) -> isize {
 2    if x <= 1 { 1 }
 3    else { x * fact(x - 1) }
 4  }
 5
 6  fn fact_iter(x : isize) -> isize {
 7    let mut res = 1;
 8    for i in 1..=x {
 9      res = res * i
10    }
11    res
12  }
13
14  fn main() {
15    println!("5! = {}", fact(5));
16    println!("5! = {}", fact_iter(5));
17  }
```

[Igralište]

Rust: apstrakcija pomoću kombinacije podataka

- Sljedeći način gradnje apstrakcije su kompozitni podaci
- Do sada smo vidjeli samo jednostavne (integer, float, Boolean)
- Najjednostavnija kombinacija je par ili n-torka (pair, tuple)

Korištenje para (n-torke)

```
1  fn add_points(a : (f64, f64),
2                b : (f64, f64)) -> (f64, f64) {
3    let x = a.0 + b.0;
4    let y = a.1 + b.1;
5    (x, y)
6  }
7
8  fn main() {
9    let a = (1.1, 2.2);
10   let b = (3.3, 4.4);
11   println!("Rezultat: {:?}", add_points(a, b))
12 }
```

[Igralište]

Rust: kreiranje i korištenje kombinacije podataka

- Načelno, za svaku kompoziciju podataka postoji:
  - Način na koji se kompozicija kreira iz jednostavnijih podataka
  - Način na koji se kompozicija *"koristi"* (i razbija opet na jednostavnije podatke od kojih je kreirana)
- Na primjeru n-torke:

Kreiranje i korištenje n-torke

```
1  // N-torka se kreira nabrajanjem elemenata
2  let n_torka = (v1, v2, ..., vn)
3
4  // Koristi se može na 2 načina:
5  // 1. Odabirom (projekcijom) pojedinog elementa
6  let x = n_torka.i // i mora biti broj 0 <= i < n
7
8  // 2. Razbijanjem na pojedinačne djelove
9  let (v1, v2, ..., vn) = n_torka
```

Rust: problemi n-torke

- Gore smo točku (u prostoru) modelirali kao par (x, y)
- Par 2 broja nam ne govori ništa detaljnije o prirodi te vrijednosti
- Na isti način bi mogli modelirati i razlomak, a točka i razlomak ipak nisu ista stvar!
- Iako ih računalo interno isto tretira
- Zapis (record/struct): zasebno definirani tip sa više (imenovanih!) elemenata

Rust: zapis / struct / record

Knjiga: https://doc.rust-lang.org/book/ch05-01-defining-structs.html,

https://doc.rust-lang.org/book/ch05-02-example-structs.html

Definiranje zapisa

[Igralište]

```
1   #[derive(Debug)] // Ignorirati za sada - potrebno za println!
2   struct Point {
3     x : f64,
4     y : f64
5   }
6
7   fn add_points(a : Point, b : Point) -> Point {
8     let x = a.x + b.x;
9     let y = a.y + b.y;
10    Point { x, y } // Napomena: Skraceno kada se imena podudaraju
11  }
12
13  fn main() {
14    let a = Point { x: 1.1, y: 2.2 };
15    let b = Point { x: 3.3, y: 4.4};
16    println!("Rezultat: {:?}", add_points(a, b))
17  }
```

Rust: kreiranje i koristenje zapisa

Definiranje zapisa

```
1   #[derive(Debug)]
2   struct Record {
3     x1 : t1,
4     x2 : t2,
5     ...,
6     xn : tn
7   }
8
9   // 1. Kreiranje:
10  let r = Record { x1: v1, x2: v2, ..., xn: vn }
11
12  // 2. Koristenje
13  let vi = r.xi // gdje je xi gore definirano ime (za 1 <= i <= n)
```

[Igralište]

Rust: derive & Debug

- Što znači `#[derive(Debug)]` iznad definicije novoga tipa?
- `derive` je makro-naredba koja automatski generira neka svojstva (trait) na tipu podataka
- `Debug` je svojstvo (trait) koje omogućuje generiranje debug-prikaza podataka
- Ukratko: `#[derive(Debug)]` je potrebno da bi se vrijednost tipa mogla ispisati sa `println!("{:?}", x)`

Rust: sljedeći put

- Enum tip
- Moduli i vidljivost (public / private)
- Svojstva (trait)
- Memorija (i pravi dio Rust tipskog sustava)