Rust i napredni tipski sustavi

2. Rust - nastavak

Ivan Radiček

20. ožujka 2021.

Rješenje zadataka - prvi pokušaj

```
1   struct Rational {
2       p: isize,
3       q: isize,
4   }
5
6   fn new_rational(p : isize, q : isize) -> Rational {
7       Rational { p, q }
8   }
9
10  fn add_rationals(r1 : Rational, r2 : Rational) -> Rational {
11      let p = r1.p * r2.q + r2.p * r1.q;
12      let q = r1.q * r2.q;
13      Rational { p, q }
14  }
15
16  fn eq_rationals(r1 : Rational, r2 : Rational) -> bool {
17      r1.p == r2.p && r1.q == r2.q
18  }
```

[Igralište]

Rješenje zadataka - gdje je problem?

- $(1, 6) \neq (2, 12)$ (Rust ne zna ništa o semantici racionalnih brojeva)
- Racionalni broj treba prvo normalizirati (svesti na "*zajednički nazivnik*")
- To možemo postići tako da prvo nađemo najveći zajednički djelitelj (vidi: `https://en.wikipedia.org/wiki/Greatest_common_divisor`)

Rješenje zadataka - Euklidov algoritam

- Vidi: https://en.wikipedia.org/wiki/Euclidean_algorithm
- Rekurzija je zadnji izraz (tail-recursion; vidi: https://en.wikipedia.org/wiki/Tail_call)

```
1  fn gcd(a : isize, b : isize) -> isize {
2    if b == 0 { a }
3    else { gcd(b, a % b) }
4  }
```

[Igralište]

Rješenje zadataka

- Gdje ugraditi normalizaciju?
- Konzistentno rješenje:
  - Normalizirati kod kreiranja strukture, tj. u `new_rational`
  - Sve druge funkcije nikad direktno ne kreiraju strukturu, nego samo kroz `new_rational`

```
1  fn new_rational(p : isize, q : isize) -> Rational {
2    let d = gcd(p, q);
3    Rational { p: p / d, q: q / d }
4  }
5
6  fn add_rationals(r1 : Rational, r2 : Rational) -> Rational {
7    let p = r1.p * r2.q + r2.p * r1.q;
8    let q = r1.q * r2.q;
9    new_rational(p, q)
10 }
```

[Igralište]

Napomena: dupliciranje varijable

Čudne greške kod prevođenja

```
1  let r = new_rational(1, 3);
2  println!("{:?} + {:?} = {:?}", r, r, add_rationals(r, r))
```

[Igralište]

- Nešto sa `move`, `borrow`, . . . !?!?
- Rustovo upravljanje resursima ne dopušta "*tek tako*" dupliciranje vrijednosti zadane variajabli (osim najjednostavnijih tipova)
- O tome ćemo kasnije kada dođemo do upravljanja memorijom
- Za sada je dovoljno staviti `#[derive(Clone, Copy)]` iznad definicije tipa!

Napomena o nizu izraza (od prošli put)

- Sve (ili skoro sve) u Rustu je izraz
- Jedan od načina gradnje kompliciranijih izraza je kombinacija dva izraza (recimo e1 i e2) sa ; u novi izraz e1 ; e2
- e1 ; e2 je isto izraz (semantika?), samo ga treba postaviti unutar {...}
- U tom slučaju može doći i kao izraz uvjeta u `if-else` grananju

```
1  fn abs(x : isize) -> isize {
2    if { println!("Provjeravam: {} < 0", x); x < 0 } { -x } else { x }
3  }
```

[Igralište]

Rust nastavak

- Prošli put
  - Osnovno o programskim jezicima, sintaksi, semantici, tipskim sustavim
  - Jednostavni izrazi, `let`-izrazi, funkcije
  - N-torke, strukture (zapis)
- Plan za danas
  - Enum tip (varijacije)
  - Metode (jednostavnija sintaksa za funkcije nad strukturama)
  - Svojstva struktura (Traits)

Rust: tagirana unija/enum-tip

- Komplement zapisu (record): *record* opisuje komponente podatka, *enum* opisuje moguće *varijante* podatka
- Varijanta može nositi i dodatne podatke
- Knjiga: https://doc.rust-lang.org/book/ch06-01-defining-an-enum.html

Definiranje i kreiranje variajnte

```
1   #[derive(Debug)]
2   enum Shape {
3       Point,
4       Triangle(f64, f64, f64),
5       Rectangle(f64, f64),
6   }
7
8   fn main() {
9       let o1 = Shape::Point;
10      let o2 = Shape::Triangle(2.0, 3.0, 4.0);
11      let o3 = Shape::Triangle(5.0, 5.0, 5.0);
12      let o4 = Shape::Rectangle(5.0, 10.5);
13      println!("o1={:?} o2={:?} o3={:?}, o4={:?}", o1, o2, o3, o4);
14  }
```

[Igralište]

Rust: match

- Svaki kompozitni tip podatka se *koristi* na neki način . . .
- Kako se koristi `enum`? Pomoću `matcha`!
- Moramo definirati vrijednost za **svaku varijantu**!
- Knjiga: https://doc.rust-lang.org/book/ch06-02-match.html

Korištenje enuma

```
1  fn perimeter(s : Shape) -> f64 {
2    match s {
3      Shape::Point => 0.0,
4      Shape::Triangle(a, b, c) => a + b + c,
5      Shape::Rectangle(a, b) => 2.0 * a + 2.0 * b
6    }
7  }
```

[Igralište]

Rust: record umjesto n-torke u enumu

Korištenje enuma

```
1   enum Shape {
2       Point,
3       Triangle { a: f64, b: f64, c: f64 },
4       Rectangle { a: f64, b: f64 },
5   }
6
7   fn perimeter(s: Shape) -> f64 {
8       match s {
9           Shape::Point => 0.0,
10          Shape::Triangle { a, b, c } => a + b + c,
11          Shape::Rectangle { a, b } => 2.0 * a + 2.0 * b,
12      }
13  }
14
15  fn main() {
16      let o = Shape::Rectangle { a: 5.0, b: 10.5 };
17      println!("O(o)={:?}", perimeter(o));
18  }
```

[Igralište]

Rust: iscrpnost matcha varijacije

- Kompajler nas tjera da ispitamo sve variajnte varijacije!
- Kod dolje nije valjan!!

Korištenje enuma

```
1   fn perimeter(s: Shape) -> f64 {
2     match s {
3       // Shape::Point => 0.0,
4       Shape::Triangle { a, b, c } => a + b + c,
5       Shape::Rectangle { a, b } => 2.0 * a + 2.0 * b,
6     }
7   }
8
9   fn main() {
10      let o = Shape::Point;
11      println!("O(o)={:?}", perimeter(o));
12  }
```

[Igralište]

Rust: enum općenito

```
1   // Definicija
2   enum E {
3     V1 t1,
4     V2 t2,
5     ...,
6     Vn tn
7   }
8
9   // Kreiranje
10  // (gdje je Vi jedna od variajnti (1 <= i <= n), a e je tipa ti)
11  let x = Vi e;
12
13  // Korištenje
14  // (gdje je pi uzorak (pattern) koji odgovara tipu ti)
15  match x with {
16    V1 p1 => e1,
17    V2 p2 => e2,
18    ...,
19    Vn pn => en
20  }
```

Rust: uzorci (patterns)

- Može i sa `let`, isto kao i u `match`

Različite vrste uzorka

```
1  // n-torka (ugniježđena)
2  let trojka = (true, 42, (2.0, 3.0));
3  let (b, i, (p1, p2)) = trojka;
4  println!("b={}, i={}, p1={}, p2={}", b, i, p1, p2);
5
6  // n-torka i struct
7  let broj_i_tocka = (42, Point { x: 2.2, y: 3.3 });
8  let (_, Point {x, y}) = broj_i_tocka;
9  println!("x={}, y={}", x, y);
```

[Igralište]

Rust: još uzoraka

- Postoji jako mnogo opcija
- Knjiga: https://doc.rust-lang.org/book/ch18-03-pattern-syntax.html
- Nećemo sve sada obraditi, neke ćemo još susretati tijekom predavanja

Brojevi i više slučajeva zajedno

```
1  let x = 5;
2
3  match x {
4      0 | 1 => println!("nula ili jedan"),
5      2..=6 => println!("dva do šest"),
6      7 => println!("sedam"),
7      _ => println!("bilo što drugo!")
8  }
```

[Igralište]

Rust: još uzoraka

- Postoji jako mnogo opcija
- Knjiga: https://doc.rust-lang.org/book/ch18-03-pattern-syntax.html
- Nećemo sve sada obraditi, neke ćemo još susretati tijekom predavanja

Dodatni uvjeti

```
1  match p {
2      Point { x, y } if x >= 0.0 && y >= 0.0 => x + y,
3      _ => 0.0
4  }
```

[Igralište]

Rust: neki korisni enum tipovi - Boolean

- Prisutan u svim jezicima
- Čak toliko da ima svoji poseban `match`: `if-then-else`
  - `match` možemo promatrati kao generalizaciju `if-then-elsea` (+ pattern matching, vrijednosti)
- Iako je u svakome jeziku ugrađeni tip, a ne definiran kao `enum`

```
1  enum Bool {
2    True,
3    False
4  }
```

Rust: neki korisni enum tipovi - option

- Koristan za operacije koje mogu vratiti vrijednost ili ništa
- Zamjenjuje `null`/`None` koji se koristi u jezicima kao C, Python, ...
- No, kompajler nas tjera da provjerimo da li je nešto `None` ili `Some` (ima vrijednost)
- Ugrađeno u standardnu biblioteku Rusta (vidi: https://doc.rust-lang.org/std/option/)

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

Rust: korištenje optiona

```
1  fn div(a : isize, b : isize) -> Option<isize> {
2    if b == 0 { None }
3    else { Some (a / b) }
4  }
5
6  fn print_div_result(a : isize, b : isize) {
7    let r = div(a, b);
8    match r {
9      None => println!("Nema rezultata"),
10     Some (n) => println!("Rezultat je: {}", n)
11   }
12 }
13
14 fn main() {
15   print_div_result(10, 5);
16   print_div_result(1000, 1000);
17   print_div_result(42, 0)
18 }
```

[Igralište]

Null References: The Billion Dollar Mistake

In his 2009 presentation "Null References: The Billion Dollar Mistake", **Tony Hoare**, the inventor of `null`, has this to say:

I call it my billion-dollar mistake. At that time, I was designing the first comprehensive type system for references in an object-oriented language. My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.

Rust: što ako nam treba samo rezultat?

- Rust nema iznimke (exception), samo `macro panic!("<text greške>")` koji zaustavlja program (ili thread)

```
1  fn main() {
2    let res = match div(10, 5) {
3      Some(x) => x,
4      None => panic!("Dijeljenje s nulom!")
5    };
6    println!("Rezultat: {}", res)
7  }
```

[Igralište]

- Može se positići i s `.unwrap()` i `.expect("<text greške>")`

Rust: još enum tipova - rezultat tip

- Koristan za operacije koje mogu vratiti vrijednost ili **grešku**
- Omogućuje rukovanje pogreškama bez korištenja iznimki (exception)
- Nekad nespretno za korištenje, ali Rust ima dosta sintakse za pomoć kod korištenja
- Postoji definicija u standardnoj bibiloteci (vidi https://doc.rust-lang.org/std/result/)

```
1  enum Result<T, E> {
2      Ok(T),
3      Err(E)
4  }
```

Rust: korištenje rezultat tipa

```rust
enum DivError {
    ZeroDivision,
    IntegerDivision
}

fn whole_div(a: isize, b: isize) -> Result<isize, DivError> {
    if b == 0 {
        Result::Err(DivError::ZeroDivision)
    } else if a % b != 0 {
        Result::Err(DivError::IntegerDivision)
    } else {
        Result::Ok(a / b)
    }
}

fn print_div_result(a: isize, b: isize) {
    let r = whole_div(a, b);
    match r {
      Result::Ok(n) => println!("Rezultat je: {}", n),
      Result::Err(err) => match err {
        DivError::ZeroDivision => println!("Dijeljenje s nulom"),
        DivError::NonIntegerDivision => println!("Cijelo dijeljenje nije moguće"),
      },
    }
}
```

[Igralište]

Rust: još rezultata

- `std::fs::read_to_string` (vidi: https://doc.rust-lang.org/std/fs/fn.read_to_string.html) - čita file u string ili vraća grešku
- Vraća `std::io::Result<String>` = `std::result::Result<String, std::io::Error>`
- Sa `Result` tipom isto možemo koristiti `unwrap` i `expect`
- Napomena: koristimo `use` umjesto cijelog naziva funkcije

```
1  use std::fs::read_to_string;
2
3  fn main() {
4    // Probaj i "/etc/shadow"
5    let content = read_to_string("/etc/passwd").unwrap();
6    // let content = read_to_string("/etc/passwd").expect("Ne mogu otvoriti file!");
7    println!("Sadržaj: {}", content)
8  }
```

[Igralište]

Rust: propagiranje greske

- `main` može vratiti i `Result<(), E>`, pa status procesa ovisi o rezultatu, a greška se može ispisati korisniku
- Na ovaj način se i općenito greška propagira sa funckije koja može vratiti grešku pozivatelju

```
1   fn main() -> Result<(), std::io::Error> {
2     let content = read_to_string("/etc/bla");
3     match content {
4       Err(err) => Err(err),
5       Ok(content) => {
6         println!("Sadržaj: {}", content) ;
7         Ok(())
8       }
9     }
10  }
```

[Igralište]

Rust: eksplicitni `return`

- Gornji kod možemo i pojednostaviti

```
1  fn main() -> Result<(), std::io::Error> {
2    let content = read_to_string("/etc/bla");
3    let content = match content {
4      Err(err) => return Err(err),
5      Ok(cnt) => cnt
6    };
7    println!("Sadržaj: {}", content) ;
8    Ok(())
9  }
```

[Igralište]

Rust: ? operator

- Gornji kod možemo i **još** pojednostaviti
- Slična sintaksa postoji i recimo u C#-u
- Može se upotrijebiti i na `Option<T>` tipu

```
1   fn main() -> Result<(), std::io::Error> {
2     let content = read_to_string("/etc/bla")?;
3     println!("Sadržaj: {}", content) ;
4     Ok(())
5   }
```

[Igralište]

Rust: metode i pridružene funkcije

- Popularna *sintaksa* u OO jezicima
- Ako je prvi argument `self`, onda definirana "*metoda*" i poziva se s `.`
- Inače "*pridružena*" funkcija i poziva se s `::`
- Knjiga:
  https://doc.rust-lang.org/book/ch05-03-method-syntax.html

```
1   impl Rational {
2     fn new(p : isize, q : isize) -> Rational {
3       ...
4       Rational { p, q }
5     }
6
7     fn add(self, other : Rational) -> Rational {
8       ...
9       Rational::new(p, q)
10    }
11  }
12
13  fn main() {
14    let r1 = Rational::new(4, 3);
15    let r2 = Rational::new(5, 2);
16    println!("Rezultat: {:?}", r1.add(r2))
17  }
```

[Igralište]

Rust: metode i &

- Moguće da trebamo i & ispred tipa ili `self`
- Ignorirati ćemo i staviti ako treba, a diskutirati kasnije kada ćemo pričati o memoriji

```
1   impl Rational {
2     fn new(p : isize, q : isize) -> Rational {
3       ...
4       Rational { p, q }
5     }
6
7     fn add(&self, other : &Rational) -> Rational {
8       ...
9       Rational::new(p, q)
10    }
11  }
12
13  fn main() {
14    let r1 = Rational::new(4, 3);
15    let r2 = Rational::new(5, 2);
16    println!("Rezultat: {:?}", r1.add(&r2))
17  }
```

[Igralište]

Rust: osobine tipova ili `traits`

- Omogućuje apstrakciju nekog dijela ponašanja skupine tipova
- Slično kao **interface** u OO jezicima
- Knjiga: https://doc.rust-lang.org/book/ch10-02-traits.html

Definicija traita

```
1  trait Shape {
2    fn perimeter(&self) -> f64;
3    fn area(&self) -> f64;
4  }
```

Rust: implementacija traita za neki tip

Definicija traita

```
1   #[derive(Debug, Clone, Copy)]
2   struct Square {
3     x : f64
4   }
5
6   impl Square {
7     fn new(x : f64) -> Option<Square> {
8       if x > 0.0 { Some (Square { x }) }
9       else { None }
10    }
11  }
12
13  impl Shape for Square {
14    fn perimeter(&self) -> f64 {
15      4.0 * self.x
16    }
17
18    fn area(&self) -> f64 {
19      self.x * self.x
20    }
21  }
```

[Igralište]

Rust: kako koristiti `trait`?

Definicija `traita`

```
1   fn print_shape(s: impl Shape + Debug) {
2     println!("s={:?}, O(s)={}, P(s)={}", s, s.perimeter(), s.area());
3   }
```

ili

```
1   fn print_shape<T : Shape + Debug>(s: T) { ... }
```

ili

```
1   fn print_shape<T>(s: T) where T : Shape + Debug { ... }
```

[Igralište]

Rust: generički tip

- `fn f<T>(...) { ... }` je **generička** funkcija (za bilo koji tip podataka `T`)
- U Rustu kompajler **implementira** generičke funkcije za svaki tip za koji se koriste (možda slično kao C++ template?) - to se naziva i **monomorphization**
- Generički kod **ne zna ništa o generičkim tipovima**!

Zamjena mjesta u n-torki/paru

```
1  fn swap<T1, T2>(p : (T1, T2)) -> (T2, T1) {
2    let (x, y) = p;
3    (y, x)
4  }
```

[Igralište]

Rust: generički tip

- Generički kod ne zna ništa o generičkim tipovima!
- Što može implementirati funkcija dolje?

u

```
1  fn f<T>(x : T) -> T {
2    ???
3  }
```

Rust: generički tip

- Generički kod ne zna ništa o generičkim tipovima!
- Što može implementirati funkcija dolje?
- **Samo i jedino funkciju identiteta**

```
1  fn id<T>(x : T) -> T {
2    x
3  }
```

[Igralište]

Rust: razlika `impl` i generičkih tipova

- Sa generičkim tipovima možemo *finije* kontrolirati tipove nego sa `impl`
- Koja je razlika između dolje navedenih tipova funkcija?

```
fn id(x : impl X + Y, y : impl X + Y) -> ...
```

vs.

```
fn id<T1 : X + Y, T2 : X + Y(x : T1, y : T2) -> ...
```

vs.

```
fn id<T : X + Y>(x : T, y : T> -> ...
```

Rust: razlika `impl` i generičkih tipova

- Sa generičkim tipovima možemo *finije* kontrolirati tipove nego sa `impl`
- Koja je razlika između dolje navedenih tipova funkcija?

```
fn id(x : impl X + Y, y : impl X + Y) -> ...
```

vs.

```
fn id<T1 : X + Y, T2 : X + Y(x : T1, y : T2) -> ...
```

vs.

```
fn id<T : X + Y>(x : T, y : T> -> ...
```

- Prvi i drugi način su identični
- Treći način *zahtjeva* da su `x` i `y` istog tipa `T` (u prvom i drugom slučajevu mogu biti različiti tipovi sve dok oba implementiraju `X` i `Y`)
- Nije moguće sa `impl`

Rust: operacije kroz obilježja

- Kako u Rustu definirati zbrajanje (pomoću +) nad definiranim tipom podataka?
- Pomoću `std::ops::Add` (vidi: https://doc.rust-lang.org/std/ops/trait.Add.html)
- Vidi općenito operacije: https://doc.rust-lang.org/std/ops/index.html

Zbrajanje dvije točke

```
1  impl Add for Point {
2    type Output = Point; // Definicija tipa rezultata (ne mora nužno biti Point)
3
4    fn add(self, rhs: Point) -> Point {
5      let x = self.x + rhs.x;
6      let y = self.y + rhs.y;
7      Point { x, y }
8    }
9  }
```

[Igralište]

Rust: još malo operacija

- Možemo i kombinirati tipove podataka

Zbrajanje dvije točke

```
1   impl Mul<f64> for Point {
2     type Output = Point;
3
4     fn mul(self, rhs: f64) -> Point {
5       let x = self.x * rhs;
6       let y = self.y * rhs;
7       Point { x, y }
8     }
9   }
```

[Igralište]

Rust: tipovi kod traitova

Definicija Add traita

```
1  trait Add<Rhs = Self> {
2    type Output;
3
4    pub fn add(self, rhs: Rhs) -> Self::Output;
5  }
```

- `Self` je tip za koji implementiramo `trait`
- Za bilo koji tip možemo:
  - Implementirati više verzija varirajući tip `Rhs`
  - Za svaku varijaciju mora biti točno jedan tip `Output` (associated type)

Rust: obilježja koja nemaju implementaciju

- Jednakost u matematici:
  - $x = x$ (refleksivna)
  - ako $x = y$, onda i $y = x$ (simetrična)
  - ako $x = y$ i $y = z$, onda i $x = y$ (tranzitivna)
- `PartialEq` je obilježje koje treba implementirati za == operaciju, i kompajler podrazumijeva da je jednakost *simetrična* i *tranzitivna*, ali ne i *refleksivna* (vidi: https://doc.rust-lang.org/std/cmp/trait.PartialEq.html)
- `Eq` je obilježje za koje ne treba ništa implementirati, ali daje nam do znanja da je jednakost još i *refleksivna* (vidi: https://doc.rust-lang.org/std/cmp/trait.Eq.html)

Rust: još o obilježjima

- `Eq` zahtjeva `PartialEq`, tj. `Eq : PartialEq`
- Npr. `f64` je `PartialEq`, ali ne i `Eq` (zbog `NaN` vrijednosti)
- Neki tipovi, kao `HashMap` (diskutirati ćemo kasnije) zahtjevaju `Eq` (`PartialEq` nije dovoljno)
  - Tj. pomažu programeru da ne koriste tip koji nije kompatibilan!
- Ima još takvih tipova, npr. `PartialOrd` i `Ord` (vidi: https://doc.rust-lang.org/std/cmp/trait.PartialOrd.html)
- Primjer `Eq` i `PartialEq` za `Point`: [Igralište]
- Primjer tipa gdje `Eq` ne vrijedi: [Igralište]

Rust: još resursa o `traitovima`

- Skoro sve u Rustu modelirano pomoću `traita`
- Operacije: https://doc.rust-lang.org/book/appendix-02-operators.html
- Obilježja koje se mogu implementirati automatski/izvesti: https://doc.rust-lang.org/book/appendix-03-derivable-traits.html