Rust i napredni tipski sustavi

5. Rust - još o upravljanju memorijom

Ivan Radiček

10. travnja 2021.

Plan

- Prošli put:
  - Posuđivanje (*borrow*)
  - Eksplicitne oznake života (*lifetime*)
- Danas:
  - Napomene, zaostaci, zadaci od prošli put
  - Pametni pokazivači (*smart pointers*)

Od prošli put . . .

- Prošli put smo definirali funkciju `longer` koja vraća dulji od 2 (posuđena) stringa

```
1  fn longer<'a>(x : &'a str, y : &'a str) -> &'a str {
2    if x.len() >= y.len() { x } else { y }
3  }
```

- Rust kompajler će sam "*skratiti*" argument varijable koja živi dulje na zajedničku lifetime `'a`
- (Podsjetnik: Rust će se samo praviti da argument živi kraće nego što stvarno živi jer je to dozvoljena operacija, dok obrnuto nije — slično kao u OO kod nasljeđivanja)
- No, može i eksplicitno . . .

Odnosi između lifetimea

- `'a : 'c` znači da `'a` živi **dulje ili jednako** kao `'c`

```
1  fn longer<'a : 'c, 'b : 'c, 'c>(x : &'a str,
2                                  y : &'b str) -> &'c str {
3    // ...
4  }
5
6  // ili
7
8  fn longer<'a, 'b, 'c>(x : &'a str, y : &'b str) -> &'c str
9        where 'a : 'c, 'b : 'c {
10   // ...
11 }
```

Podsjetnik: defincija strukture sa referencom

```
1  #[derive(Debug)]
2  struct Strings<'a> {
3    x : &'a str,
4    y : &'a str
5  }
6
7  impl<'a> Strings<'a> {
8    fn new(x : &'a str, y : &'a str) -> Self {
9      Strings { x, y }
10   }
11 }
12
13 fn main() {
14   let x = String::from("hello");
15   let y = String::from("world");
16   let s = Strings::new(&x, &y);
17   println!("s = {:?}", s)
18 }
```

Da li je ovo OK?

```
1   #[derive(Debug)]
2   struct Strings<'a> {
3     x : &'a str,
4     y : &'a str
5   }
6
7   /* ... */
8
9   fn main() {
10    let x = String::from("hello");
11    let z = {
12      let y = String::from("world");
13      let s = Strings::new(&x, &y);
14      s.x
15    };
16    println!("s = {:?}", z)
17  }
```

Da li je ovo OK?

```
1   #[derive(Debug)]
2   struct Strings<'a> {
3     x : &'a str,
4     y : &'a str
5   }
6
7   /* ... */
8
9   fn main() {
10    let x = String::from("hello");
11    let z = {
12      let y = String::from("world");
13      let s = Strings::new(&x, &y);
14      s.x
15    };
16    println!("s = {:?}", z)
17  }
```

- Možemo probati sa `cargo check`

Da li je ovo OK?

```
1   #[derive(Debug)]
2   struct Strings<'a> {
3     x : &'a str,
4     y : &'a str
5   }
6
7   /* ... */
8
9   fn main() {
10    let x = String::from("hello");
11    let z = {
12      let y = String::from("world");
13      let s = Strings::new(&x, &y);
14      s.x
15    };
16    println!("s = {:?}", z)
17  }
```

- Možemo probati sa `cargo check`
- !?!? Pa ne pokušavamo posuditi `y`!

Detaljnija analiza . . .

```
1   /* ... */
2   impl<'c : 'a, 'c : 'b, 'c> Strings<'c> {
3     fn new(x : &'a str, y : &'b str) -> Strings<'c> {
4       Strings { x, y }
5     }
6   }
7
8   fn main() {
9     let x = String::from("hello");
10    let z = {
11      let y = String::from("world");
12      let s = Strings::new(&x, &y);
13      s.x
14    };
15    println!("s = {:?}", z)
16  }
```

- Lifetime `s`-a može biti samo kraći od `x` i `y`, odnosno lifetime `y`-a , nazovimo ga `'b`
- Definicija `Strings`-a kaže da je lifetime `s.x`-a isti kao i od `s`, odnosno `'b`
- No, lifetime `'b` je od linija 11-13, pa `s.x` ne možemo pridružiti `z`-u

Precizniji opis lifetima

- Problem: izgubili smo preciznost, jer ne razlikujemo lifetime `s.x`-a i `s.y`-a (pa moramo uzeti kraći)
- Rješenje: pratiti lifetime `s.x`-a i `s.y`-a odvojeno

```
1  #[derive(Debug)]
2  struct Strings<'a, 'b> {
3    x: &'a str,
4    y: &'b str,
5  }
6
7  impl<'a, 'b> Strings<'a, 'b> {
8    fn new(x: &'a str, y: &'b str) -> Strings<'a, 'b> {
9      Strings { x, y }
10   }
11 }
```

Više lifetimea u strukturi

- Cijeli `struct` živi kao i najkraći od dijelova (`'a` i `'b` gore)
- Stoga ovo dolje nije ispravno (iako kompajler daje čudnu grešku)

```
1  fn main() {
2    let x = String::from("hello");
3    let s = {
4      let y = String::from("world");
5      Strings::new(&x, &y)
6    };
7    println!("s.x = {:?}", s.x)
8  }
```

Više o vježbama

- Nema baš puno zanimljivih vježbi za mladi jezik kao što je Rust
- Postoji projekt `rustlings`:
  https://github.com/rust-lang/rustlings
  - Razvrstano po temama od kojih smo većinu već i prošli
  - Pokušajte riješiti čim više bez pomoći (i pitajte za pomoć kad baš zapne)
- No, najbolja vježba će nam biti korištenje jezika za pravi projekt

A sada nešto potpuno drugačije

- Funkcija uzima neki argument koji ima Debug trait i ispisuje ga

```
1  use std::fmt::Display;
2
3  fn print_me<T : Display>(x : T) {
4    println!("Printing: {}", x)
5  }
6
7  fn main() {
8    print_me(42);
9    print_me(true);
10   print_me("Hello world")
11   // Ovo ne: print_me((42, true))
12 }
```

Pa još malo *dinamičnije*

```
1  use std::fmt::Display;
2
3  fn print_me<T : Display>(x : T) {
4    println!("Printing: {}", x)
5  }
6
7  fn main() {
8    let x = if 42 < 42 { true } else { 5 };
9    print_me(x)
10 }
```

Pa još malo *dinamičnije*

```
1  use std::fmt::Display;
2
3  fn print_me<T : Display>(x : T) {
4    println!("Printing: {}", x)
5  }
6
7  fn main() {
8    let x = if 42 < 42 { true } else { 5 };
9    print_me(x)
10 }
```

- Možemo pokušati sa `x : Display` ...

Pa još malo *dinamičnije*

```
1  use std::fmt::Display;
2
3  fn print_me<T : Display>(x : T) {
4    println!("Printing: {}", x)
5  }
6
7  fn main() {
8    let x = if 42 < 42 { true } else { 5 };
9    print_me(x)
10 }
```

- Možemo pokušati sa `x : Display` ...
- ili `fn print_me(x : dyn Display)` i `x : dyn Display` ...

Pa još malo *dinamičnije*

```
1  use std::fmt::Display;
2
3  fn print_me<T : Display>(x : T) {
4    println!("Printing: {}", x)
5  }
6
7  fn main() {
8    let x = if 42 < 42 { true } else { 5 };
9    print_me(x)
10 }
```

- Možemo pokušati sa `x : Display` ...
- ili `fn print_me(x : dyn Display)` i `x : dyn Display` ...
- no, na kraju, kompajler će reći da `dyn T` ima veličinu koja nije poznata kod kompilacije. Poznato!?

Lista

```
1  enum List<T> {
2    Nil,
3    Cons { head: T, tail: List<T> }
4  }
```

- Isti (ili sličan) problem . . .
- Jedno od rješenja koje komapjler nudi je `Box` . . .

Alokacija memorije na heapu

- Tip `Box<T>` definira pokazivač na heap
- Inicijaliziramo sa `Box::new`
- Vrijednost tipa `Box<T>` dereferenciramo pomoću `*`

```
1  fn get_value(x: Box<String>) -> String {
2    // Možemo li izvući vrijednost od posuđenoga Box-a?
3    *x
4  }
5
6  fn main() {
7    let x = Box::new(5);
8    let y = Box::new("Hello world!".to_string());
9    let n = *x;
10   let s = get_value(y);
11   println!("n={}, s={:?}", n, s)
12 }
```

Za što je koristan `Box`

- Kada neki tip zauzima puno memorije, a prenosi ga se često kao argument — tada ga je efikasnije staviti na heap, i okolo prosljeđivati samo referencu
- Kod tipova podataka koji nemaju poznatu veličinu, kao što je npr. `dyn T` i rekurzivni tipovi podataka (za sada lista, ali ćemo ih još vidjeti kod našeg projekta)

Dinamički traitovi

Ovo sad funkcionira

```
1   use std::fmt::Display;
2
3   fn print_me<T: Display>(x: T) {
4     println!("Printing: {}", x)
5   }
6
7   fn main() {
8     let x : Box<dyn Display> =
9       if 42 <= 42 { Box::new(true) }
10      else { Box::new(5) };
11    print_me(x)
12  }
```

- U definiciji funkcije može i `x : Box<dyn Display>`

Error trait

- Korisno za npr. propagiranje proizvoljnih grešaka (vidi trait `Error`: https://doc.rust-lang.org/std/error/trait.Error.html)

Ovo sad funkcionira

```
1   use std::{error::Error, fmt::Debug};
2   use std::fmt::{self, Display};
3
4   #[derive(Debug)]
5   enum MyError { Unknown, Code(u8) }
6
7   impl Display for MyError {
8     fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
9       Debug::fmt(&self, f)
10    }
11  }
12
13  impl Error for MyError { }
14
15  fn check_code(c : isize) -> Result<(), Box<dyn Error>> {
16    if c == 0 { Ok(()) }
17    else if c > 0 && c <= 255 { Err(Box::new(MyError::Code(c as u8))) }
18    else { Err(Box::new(MyError::Unknown)) }
19  }
20
21  fn main() {
22    let x = 42;
23    println!("Code is: {:?}", check_code(x))
24  }
```

Definiranje rekurzivnih tipova podataka na heapu

- Box<T> je samo pokazivač na memoriju na heapu, pa ima i poznatu veličinu u memoriji (koju?)
- Stoga, možemo definirati rekurzivni tip podataka kao što je lista

```
1   #[derive(Debug)]
2   enum List<T> {
3     Nil,
4     Cons { head: T, tail: Box<List<T>> }
5   }
6
7   impl<T> List<T> {
8     // Pomoćna funkcija za kreiranje liste
9     fn cons(head : T, tail : List<T>) -> List<T> {
10      List::Cons { head, tail: Box::new(tail) }
11    }
12  }
13
14  fn main() {
15    // Koja je razlika između List::Cons i List::cons?
16    // A koja između List<T> i Box<List<T>>?
17    let list = List::cons(1, List::cons(2, List::cons(4, List::Nil)));
18    println!("{:?}", list);
19  }
```

Vježba: funkcije nad listom

```
impl<T> List<T> {
  fn len(&self) -> usize {
    // Vraća dužinu liste
  }

  fn tail(self) -> Option<List<T>> {
    // Vraća tail liste ako postoji (lista nije prazna)
  }

  fn to_vec(self) -> Vec<T> {
    // Pretvara listu u vektor
    // Vektor se kreira sa: let v : Vec::new()
    // A koristi sa: v.push(x)
  }
}
```

Funkcije nad listom

```
1   impl<T> List<T> {
2     fn len(&self) -> usize {
3       match self {
4         List::Nil => 0,
5         // Kojeg je tipa tail ovdje?
6         // Zašto tail.len() funkcionira?
7         List::Cons { head: _, tail } => 1 + tail.len(),
8       }
9     }
10
11    fn tail(self) -> Option<List<T>> {
12      match self {
13        List::Nil => None,
14        // _ označava da nećemo koristiti head
15        // (inače se javi upozorenje da ga ne koristimo)
16        List::Cons { head: _, tail } => Some(*tail)
17      }
18    }
19  }
```

Funkcije nad listom (nastavak)

```
1   impl<T> List<T> {
2     fn populate_vec(self, v : &mut Vec<T>) {
3       match self {
4         List::Nil => (),
5         List::Cons { head, tail } => {
6           v.push(head);
7           tail.populate_vec(v)
8         }
9       }
10    }
11
12    fn to_vec(self) -> Vec<T> {
13      let mut v = Vec::new();
14      self.populate_vec(&mut v);
15      v
16    }
17  }
```

Duplikacija list

- Kako iz posuđene liste kreirati novu listu nad kojom imamo vlasništvo? Klonirati cijelu strukturu!
- Što se događa u memoriji u ovoj funkciji?

```
1  impl List<isize> {
2    // Nema parametra kod impl jer funkcija nije generička
3    // nego za specifičan tip (isize)
4    fn clone_list(&self) -> Self {
5      match self {
6        List::Nil => List::Nil,
7        List::Cons { head, tail } => List::cons(head, tail.clone_list())
8      }
9    }
10 }
```

- Ovo funkcioniramo (trenutno) samo za `T : Copy` (kao što je recimo `isize` u primjeru)
- No, može se generalizirati - `Clone` trait

Implementacija kloniranja nad listom

- Vidi: https://doc.rust-lang.org/std/clone/trait.Clone.html
- Sada funkcionira za bilo koji `T : Clone`

```
1  impl<T : Clone> Clone for List<T> {
2    fn clone(&self) -> Self {
3      match self {
4        List::Nil => List::Nil,
5        List::Cons { head, tail } => List::cons(head.clone(), *tail.clone())
6      }
7    }
8  }
```

Korištenje iste pod-liste više puta

Naravno, ovo neće funkcionirati

```
1  fn main() {
2    let l = List::cons(4, List::cons(5, List::cons(6, List::Nil)));
3    let l1 = List::cons(2, l);
4    // Koristimo l nakon što je promijenio vlasnika
5    let l2 = List::cons(1, l);
6    println!("l1 = {:?}, l2 = {:?}", l1, l2);
7  }
```

- Funkcionira ako kloniramo `l` na liniji 3 . . .

Korištenje iste pod-liste više puta - kloniranje

Ovo funkcionira

```
1  fn main() {
2    let l = List::cons(4, List::cons(5, List::cons(6, List::Nil)));
3    let l1 = List::cons(2, l.clone());
4    let l2 = List::cons(1, l);
5    println!("l1 = {:?}, l2 = {:?}", l1, l2);
6  }
```

- Ali . . . ?

Korištenje iste pod-liste više puta - kloniranje

Ovo funkcionira

```
1  fn main() {
2    let l = List::cons(4, List::cons(5, List::cons(6, List::Nil)));
3    let l1 = List::cons(2, l.clone());
4    let l2 = List::cons(1, l);
5    println!("l1 = {:?}, l2 = {:?}", l1, l2);
6  }
```

- Ali . . . ?
- Ako je `l` jako dugačka lista, kopirati ćemo je (sa cijelom memorijom) na liniji 3
- Da li je moguće da `l1` i `l2` dijele istu memoriju?

Dijeljenje iste memorije

- `std::rc::Rc<T>` je pokazivač na memoriju na heapu koja se može dijeliti (vidi: https://doc.rust-lang.org/std/rc/struct.Rc.html)
- No, uz podatke, zapisan je još i **broj** *dijelitelja* ili **referenci**:
  - Kod svakog novog referenciranja (kloniranja) taj broj se poveća
  - Kada neka varijabla više ne koristi podatak taj broj se smanjuje
  - Kada broj dosegne nulu - memorija se dealocira
- Zove se još i **reference counting**
  - Neki jezici koriste RC kao metodu garbage-collectiona (CPython)
- Korištenje:
  - `Rc::new(x)` kreira novi RC pokazivač
  - `Rc::clone(x)` kreira novu referencu (može i `x.clone()`, ali moguće zamijeniti za metodu nad podacima u memoriji)
  - `Rc::strong_count(x)` vraća broj referenci (može i `x.strong_count()`)
- `Rc` se ne može koristiti sa threadovima, ali `std::sync::Arc` je ekvivalent u svijetu thredova

Primjer RC-a

```
1  use std::rc::Rc;
2
3  fn print_count<T>(x: &Rc<T>) {
4    println!("count = {}", Rc::strong_count(x))
5  }
6
7  fn main() {
8    let x = Rc::new(5);
9    print_count(&x);
10   {
11     let y = Rc::clone(&x);
12     print_count(&x);
13     print_count(&y);
14     let y = *y; // Može i bez ovoga
15     println!("y = {}", y)
16   }
17   print_count(&x);
18 }
```

RC lista

- Zadatak: definirati tip i metode `List<T>` pomoću `Rc` umjesto `Box`