UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Jovan Dmitrović

MASTER IZ MATEMATIKE ILI RAČUNARSTVA ČIJI JE NASLOV JAKO DUGAČAK

master rad

Beograd, 2021.

Mentor:

dr Mika Mikić, redovan profesor
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

dr Ana Anić, vanredni profesor
University of Disneyland, Nedođija

dr Laza Lazić, docent
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: _____

Mami, tati i dedi

Naslov master rada: Master iz matematike ili računarstva čiji je naslov jako dugačak

Rezime: Apstrakt ide ovde

Ključne reči: programiranje, programski jezici

Sadržaj

Glava 1

Uvod

Glava 2

Programski jezik *Rust*

Rust je statički tipiziran jezik fokusiran na bezbednost i performanse. Od svog nastanka, ovaj jezik je dobio veliku pažnju u svetu programiranja, čemu svedoči i činjenica da je *Rust* proglašen za „omiljeni programski jezik" već petu godinu za redom u anketi koju je sprovela popularna veb-stranica *Stack Overflow* [15].

Danas se *Rust* koristi na velikom broju komercijalnih projekata. Na primer:

- U AWS servisima firme Amazon, poput *Lambda*, *EC2* i *Cloudfront* [2],
- U okviru operativnog sistema kompanije Gugl (engl. *Google*) *ChromeOS* [5],
- U određenim komponentama Majkrosoftove platforme *Azure*, uključujući i IoT sigurnosni servis *edgelet* [7],
- U registru *JavaScript* paketa *npm*, kod procedura koje prouzrokuju veliko opterećenje procesora [13],
- U Mozilinom (engl. *Mozilla*) veb-brauzeru Fajerfoks (engl. *Firefox*) [6].

2.1 Razvoj jezika *Rust*

Programski jezik *Rust* je dizajnirao Grejdon Hor (engl. *Graydon Hoare*) koji je, u to vreme, bio zaposlen u kompaniji Mozila. Hor je rad na ovom jeziku započeo 2006. godine kao svoj lični projekat, na kojem je samostalno radio naredne tri godine. Sredinom 2010. godine, u projekat se uključila i sama Mozila, koja i danas sponzoriše njegov razvoj. Pored zaposlenih Mozile, pošto je u pitanju programski jezik otvorenog koda, svoj doprinos je dalo i preko 5000 dobrovoljaca [14].

Pre nego što se Mozila priključila projektu, *Rust* je izgledao dosta drugačije nego danas. U svojoj početnoj fazi, *Rust* je bio čist funkcionalni jezik, tj. nije imao bočne efekte; takođe, postojala je i analiza promene stanja (engl. *typestate analysis*), koja je omogućavala proveru operacija koje se mogu izvoditi nad specifičnim tipom podataka pri kompiliranju. Za razliku od ove dve osobine, neka dizajnerska rešenja su ostala do danas, kao što je imutabilnost i kontrola pristupa memoriji [9].

Nedugo nakon priključivanja Mozile, Grejdon Hor napušta projekat 2012. godine. U ovom periodu *Rust* dobija svoj menadžer paketa *Cargo* zajedno sa repozitorijumom paketa `crates.io`. Proces *RFC*, inspirisan procesom *PEP* programskog jezika *Python* [16], se osniva 2014. godine u svrhu strogog kontrolisanja novina u samom jeziku.

Prva verzija programskog jezika *Rust*, odnosno *Rust* 1.0, objavljena je 2015. godine [1]. U *Rust* zajednici je formiran model izbacivanja novih verzija svakih šest nedelja. To je dinamičniji pristup u odnosu na većinu programskih jezika gde je taj period minimalno godinu dana. Ovom odlukom se stavlja akcenat na stabilnost jezika time što će svaka nova verzija biti slična svom prethodniku, dok se kod jezika sa dugim periodom između verzija očekuju velike promene, što može da šteti kompatibilnosti.

U februaru 2021. godine ozvaničen je nastanak fondacije *Rust* (engl. `Rust Foundation`), neprofitne organizacije osnovane u svrhe daljeg razvoja programskog jezika Rust [17]. Pored Mozile, sponzori fondacije *Rust* su i kompanije *Google*, *AWS*, *Huawei*, *Facebook* i *Microsoft*.

2.2 Instalacija i korišćenje sistema *Cargo*

Ukoliko se zvanična veb-stranica programskog jezika *Rust* poseti koristeći mašinu koja ima *Windows* operativni sistem, biće ponuđene instalacione datoteke za 32-bitne i 64-bitne *Windows* sisteme.

Na *GNU/Linux* operativnim sistemima potrebno je uneti sledeću komandu u terminal:

```
curl --proto '=https' --tlsv1.2 https://sh.rustup.rs -sSf | sh
```

Potvrdu da li se instalacija uspešno izvršila može se dobiti proverom verzije *Rust* kompilatora komandom:

```
rustc --version
```

Pored samog *Rust* kompilatora, u instalaciju su uključeni i sistem *Cargo* i alat *rustup*. Alat *rustup* daje mogućnost dobavljanja nove verzije *Rust*-a sa veba, kao i mogućnost deinstalacije komandama:

```
rustup update
rustup self uninstall
```

Pored toga što je menadžer paketa, *Cargo* vrši i automatizaciju prevođenja. Kreiranje novog projekta uz pomoć ovog sistema izvršava se komandom:

```
cargo new novi_projekat
```

Komandom iznad se pravi novi direktorijum `novi_projekat` koji sadrži konfiguracionu datoteku `Cargo.toml` i direktorijum `src` koji treba da sadrži izvorne datoteke obeležene ekstenzijom *.rs* i u kojem se inicijalno nalazi datoteka *main.rs*. Takođe, sa novim direktorijumom se inicijalizuje i novi *Git* repozitorijum.

Generisana `Cargo.toml` datoteka je prikazana na listingu 2.1.

```
1 [package]
2 name = "novi_projekat"
3 version = "0.1.0"
4 authors = ["jovan <jdmitrovic@gmail.com>"]
5 edition = "2018"
6
7 [dependencies]
```

Listing 2.1: Inicijalna *Cargo.toml* datoteka

U prvoj liniji koda, `[package]` označava sekciju koja opisuje paket koji je napravljen. Informacije koje se ovde nalaze su dobijene iz varijabli okruženja. Posle oznake `[dependencies]` se popisuju svi paketi koji su neophodni za rad sa novim paketom, tako da ih *Cargo* može dopremiti.

Na listingu 2.2 je prikazana `main.rs` datoteka. U njoj se, po osnovnim podešavanjima, nalazi *Hello World* program.

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

Listing 2.2: Inicijalna *main.rs* datoteka

Cargo prevodi projekat komandom `cargo build` a prevodi ga i pokreće `cargo run`. Korišćenjem komande `cargo check` može se proveriti da li se kôd kompilira, bez generisanja izvršne datoteke, što je korisno jer je ova opcija efikasnija od korišćenja pomenute `build` komande. Kompiliranjem projekta se pravi nova putanja `target/debug`, gde će se generisati izvršne datoteke.

Kompilator ima dva profila: `dev` profil, koji se koristi za prevođenje koda tokom razvoja, i `release` profil za prevođenje završne verzije programa, spremne za isporuku. Razlika između profila ogleda se u nivou optimizacije izvršnog koda: koristeći `dev` profil, kompilator će koristiti minimalan nivo optimizacije zarad bržeg prevođenja, dok se za `release` profil projekat prevodi sa maksimalnim nivoom optimizacije zarad dobijanja najboljih performansi programa. Podrazumevano ponašanje je da se koristi `dev` profil pokretanjem komande `cargo build`, dok se prevođenje u `release` profilu izvršava komandom:

```
cargo build --release
```

2.3 Osnovne karakteristike jezika

U nastavku će biti opisane bitne karakteristike programskog jezika *Rust*, koje nisu nužno u okviru jednog stila programiranja. Zastupljeno je nekoliko programskih paradigmi:

- Imperativna,
- Objektno-orijentisana, gde se umesto klasa koriste svojstva (engl. *traits*),
- Generička, u vidu generičkih tipova,
- Funkcionalna, u vidu iteratora i zatvorenja [11],
- Konkurentna [10].

Promenljive i konstante

Promenljive se mogu definisati korišćenjem ključne reči `let`, čime se podrazumeva da je takva promenljiva zapravo imutabilna, tj. ona se ne može menjati. Imutabilnost promenljivih omogućava kompilatoru da prepozna razne vrste grešaka

već u fazi kompilacije. Ipak, Rust dozvoljava i definisanje mutabilnih promenljivih korišćenjem ključnih reči `let mut`.

Pored promenljivih, *Rust* dozvoljava i definisanje konstanti upotrebom ključne reči `const`. Konstante se razlikuju od imutabilnih promenljivih po tome što se mogu definisati u bilo kom opsegu, uključujući i globalni, i po tome što konstante samo mogu imati vrednost konstantnog izraza, ali ne i vrednost izvršavanja funkcije.

U okviru jezika dozvoljeno je i tzv. sakrivanje (engl. *shadowing*). Sakrivanje je ponovno definisanje promenljivih. Za razliku od korišćenja ključne reči `mut`, koja omogućava promenu vrednosti promenljive, prilikom sakrivanja je moguće promeniti tip promenljive koja se ponovo definiše.

Tipovi

Programski jezik *Rust* je statički tipiziran jezik, ali ne zahteva pisanje tipa uz svaku promenljivu, osim ako je to neophodno; primer je korišćenje funkcije `parse`, koja je data na listingu 2.3:

```
1 fn main() {
2     let num = "10".parse().expect("Nije unet broj!");
3     println!("Unet broj je: {}", num);
4 }
```

Listing 2.3: Korišćenje funkcije `parse`

Ukoliko se pokuša prevođenje koda sa listinga 2.3, *Cargo* će pokazati grešku:

```
error[E0282]: type annotations needed
 --> src/main.rs:2:9
  |
2 |     let num = "10".parse().expect("Nije unet broj!");
  |         ^^^ consider giving `num` a type
```

Ova greška se javlja zbog toga što funkcija `parse` prima generičke parametre, te je kompilatoru neophodna informacija kojeg je tipa promenljiva `num`.

Rust sadrži četiri vrste prostih tipova: cele brojeve, brojeve zapisane u pokretnom zarezu, karaktere i Bulove konstante *true* i *false*. Celi brojevi mogu biti označeni ili neoznačeni. Označeni brojevi su predstavljeni tipovima `i8`, `i16`, `i32`, `i64` i `i128`, gde broj posle karaktera `i` predstavlja veličinu tipa u bitovima. Neoznačeni brojevi su predstavljeni analogno označenim, s tim da oni počinju karakterom `u`.

Postoje i tipovi `isize` i `usize` čija veličina zavisi od arhitekture. Analogon tipovima `float` i `double` iz programskog jezika *C* su tipovi `f32` i `f64`. Tip `char` je veličine četiri bajta, gde su karakteri predstavljeni *Unicode* vrednostima.

Osnovni složeni tipovi u programskom jeziku *Rust* su torke i nizovi. Torke se predstavljaju navođenjem liste tipova, razdvojene zarazima u okviru zagrada. Primer sa listinga 2.4 ilustruje upotrebu torki:

```
fn main() {
    let koordinate: (i32, f32, i32) = (1, 2.0, 3);
    let (x, y, z) = koordinate;

    println!("x: {}, y: {}", x, koordinate.1);
}
```

Listing 2.4: Inicijalizacija i upotreba torki

Torke mogu sadržati vrednosti različitog tipa, a može im se pristupiti ili korišćenjem novih promenljivih, ili korišćenjem `.i` sintakse, čime se pristupa elementu na i-toj poziciji, gde se elementi torke broje počevši od 0.

Nizovi predstavljaju kolekciju vrednosti istog tipa i, poput torki, fiksne su veličine. Obeležavaju se sa uglastim zagradama, unutra kojih su elementi razdvojeni zarezima.

Niske se u programskom jeziku *Rust* javljaju u dva oblika: u obliku literala i u obliku niske promenljive dužine (u *Rust*-u se one zovu *string*, odnosno *String*). Literalima se veličina zna pre kompilacije i njihov sadržaj se ne može promeniti, pa se one mogu čuvati na steku, dok se niske promenljive dužine moraju čuvati na hipu.

Funkcije

Funkcije se definišu ključnom reči `fn`, navođenjem imena funkcije za kojim sledi lista parametara i njihovih tipova u zagradama, razdvojenih zarezima. Nakon liste parametara, moguće je napisati tip povratne vrednosti posle oznake `->`. Ukoliko povratni tip izostane, kompilator će ga automatski izvesti. Na kraju se nalazi telo funkcije između vitičastih zagrada. Primer funkcije koja izračunava zbir kvadrata dva broja je dat u okviru listinga 2.5.

```
fn zbir_kvadrata(x: i32, y: i32) -> i32 {
  x*x + y*y
```

```
}
```

Listing 2.5: Funkcija koja izračunava zbir kvadrata

Pored standardnog korišćenja ključne reči `return` kao oznake za povratnu vrednost funkcije, funkcija će vratiti vrednost poslednje naredbe koja se ne završava delimiterom.

Kontrola toka

Rust podržava klasične imperativne načine kontrole toka sa `if`, `while` i `for` komandama. Postoji par dodatnih mogućosti koje *Rust* takođe nudi:

- Ukoliko je potrebno napisati beskonačnu petlju, nije neophodno koristiti `while` naredbu sa uvek tačnim uslovom, već se može koristiti ključna reč `loop`, kao u listingu 2.6, gde se *loop* koristi u svrhe nalaženja najvećeg zajedničkog delioca dva broja,
- I `while` i `for` se mogu naći u okviru `let` naredbe, s tim da se u telu petlje mora naći `break` naredba uz koju se dopisuje povratna vrednosti kao u primeru 2.6, analogno korišćenju `return` naredbe u funkcijama,
- Iteriranje kroz kolekciju pomoću `for` petlje se može vršiti ili preko indeksa trenutnog elementa, ili pomoću `iter` funkcije. Za iteriranje od kraja kolekcije, na `iter` se nadovezuje funkcija `rev` kao u listingu 2.7.

```
fn main() {
    let mut a: u32 = 65342;
    let mut b: u32 = 23456;

    let gcd = loop {
        if b == 0 || a == b {
            break a;
        }

        if a == 0 {
            break b;
        }

```

```
14          if a > b {
15              a -= b;
16          } else {
17              b -= a;
18          }
19      };
20
21      println!("Najveci zajednicki delilac a i b je {}", gcd);
22  }
```

Listing 2.6: Korišćenje naredbe *loop* i petlje u okviru *let* naredbe

```
1  fn main() {
2      let a = [1, 2, 3];
3
4      for elem in a.iter().rev() {
5          println!("{}", elem);
6      }
7  }
```

Listing 2.7: Korišćenje `iter` i `rev` funkcija

Vlasništvo

Koncept **vlasništva** je novina u odnosu na druge programske jezike. Ova osobina dozvoljava programskom jeziku *Rust* da funkcioniše bez sakupljača otpadaka. Za razliku od sakupljača otpadaka, sistem vlasništva ne utiče ni na koji način na izvršavanje programa, jer se postupanje prema skupu pravila vlasništva proverava prilikom prevođenja.

Sistem vlasništva se može svesti na tri pravila:

1. Svaka vrednost u programskom jeziku *Rust* ima promenljivu koja je poseduje.
2. U jednom trenutku, za svaku vrednost, postoji tačno jedan vlasnik.
3. Kada promenljiva završi svoj životni vek, tada se vrednost koju ta promenljiva poseduje automatski briše iz memorije.

Primer funkcionisanja ovog sistema se može dati uz pomoć pomenutih niski promenljive dužine, odnosno tipa `String`: ako se, kao u listingu 2.8, pokuša ulančavanje pokazivača na istu vrednost, `Rust` kompilator će prikazati sledeću grešku:

```
error[E0382]: borrow of moved value: `niska1`
 --> src/main.rs:6:32
  |
2 |     let niska1 = String::from("Hello World!");
  |         ------ move occurs because `niska1` has type `String`, which
does not implement the `Copy` trait
3 |     let niska2 = niska1;
  |                  ------ value moved here
...
6 |     println!("Prva niska: {}", niska1);
  |                                ^^^^^^ value borrowed here after move
```

Kompilator ukazuje na promenu poseda vrednosti na koju pokazuje `niska1`, čime promenljiva `niska1` gubi mogućnost manipulisanja vrednošću nad kojom je imala vlasništvo, time poštujući pravilo da vrednost pripada samo jednoj promenljivoj.

```
1 fn main() {
2     let niska1 = String::from("Hello World!");
3     let niska2 = niska1;
4 
5     println!("Druga niska: {}", niska2);
6     println!("Prva niska: {}", niska1);
7 }
```

Listing 2.8: Prenos vlasništva između promenljivih

Može se uočiti da je kopiranje promenljive `niska1` plitko. Duboko kopiranje promenljivih se odvija pomoću metoda `clone` koja je implementirana za tip `String`.

Kada se vrednost promenljive šalje kao parametar funkcije, vlasništvo te vrednosti prelazi u posed te funkcije, odnosno, promenljiva iz pozivaoca se ne može koristiti nakon pozivanja funkcije. Međutim, pozvana funkcija može *vratiti* vlasništvo nad promenljivom tako što iskoristi tu promenljivu kao povratnu vrednost: time se vlasništvo te promenljive vraća pozivaocu te funkcije. Promenljiva koja

je vraćena može biti bilo koja promenljiva nad kojom ta funkcija ima vlasništvo, uključujući i promenljive koje ne potiču odatle (dakle, i sami parametri funkcije).

Pozajmljivanje

Proces davanja i preuzimanja vlasništva može da bude nepogodan u nekim slučajevima; zbog toga, postoji i opcija **pozajmljivanja** (engl. *borrowing*). Pozajmljivanje omogućava funkcijama da pozajme vrednosti promenljivih, koje se moraju vratiti vlasniku po završetku izvršavanja te funkcije, što znači da se pozajmljene promenljive mogu kasnije koristiti. Promenljive se pozajmljuju koristeći simbol `&`, što je ilustrovano na listingu 2.9, gde će izvršavanje koda ispisati: `Niska „Hello world" ima 2 reci`.

```
fn main() {
    let s = String::from("Hello world");

    println!("Niska \"{}\" ima {} reci", s, broj_reci(&s));
}

fn broj_reci(s: &String) -> usize {
    s.matches(" ").count() + 1
}
```

Listing 2.9: Pozajmljivanje vrednosti promenljivih

Na sličan način se definišu i reference na promenljive. Vrednosti pozajmljenih promenljivih se ne mogu menjati preko referenci definisanih na ovaj način; u te svrhe se koriste mutabilne reference koje se obeležavaju sa `&mut`. Za razliku od referenci u drugim programskim jezicima, može postojati samo jedna mutabilna referenca za jednu promenljivu u istom dosegu u kojem ne sme postojati ni imutabilna referenca te promenljive.

U *Rust*-u je nemoguće napraviti "zalutalu" referencu (engl. *dangling reference*), tj. referencu koja referiše na prostor u memoriji koji je već oslobođen. Prilikom prevođenja koda, kompilator će dati preporuku za korišćenje statičke reference uz pomoć sintakse `&'static`. Ukoliko bi se pokušalo prevođenje koda iz listinga 2.10, kompilator ne bi dozvolio prevođenje bez `static` oznaka. Konvencija je da imena promenljivih sa statičkim životnim vekom počinju velikim početnim slovom.

```
fn main(){
```

```
    println!("{}", num());
}

fn num() -> &'static u32 {
    static N: u32 = 1234;

    return &N;
}
```

Listing 2.10: Korišćenje promenljivih sa statičkim životnim vekom

Strukture

Definisanje struktura se vrši na uobičajen način, ali *Rust* nudi mogućnosti koje skraćuju sintaksu, kao u listingu 2.11.

```
struct Student {
    ime: String,
    indeks: String,
    email: String
}

fn kreiraj_studenta(ime: String, indeks: String) -> Student {
    Student {
        ime,
        email: format!("{}@matf.bg.ac.rs", indeks),
        indeks
    }
}

fn main() {
    let student1 = kreiraj_studenta(
        String::from("Ana"),
        String::from("mi15123")
    );
    let student2 = Student {
        indeks: String::from("mr14321"),
        ..student1
    };
```

```
24 }
```

Listing 2.11: Kreiranje novih struktura

Sa listinga 2.11 se može primetiti kako se uz pomoć oznake `..` mogu definisati nove instance strukture gde se imena polja poklapaju sa imenima parametara funkcije. U istom primeru će instanca `student2` imati isto ime kao i instanca `student1`, ali će polje `indeks` imati drugačiju vrednost.

Strukture se mogu i definisati putem torki, s tim da njihova polja neće imati imena, već će im se pristupati `.` notacijom. Strukture definisane na ovaj način ne moraju nužno imati polja, već im se struktura i ponašanje može postaviti putem interfejsa.

Da bi se implementirali metodi u okviru struktura, koristi se ključna reč `impl` kao u listingu 2.12.

```
1 impl Student {
2     fn promena_mejla(&mut self, novi_email: String) {
3         self.email = novi_email;
4     }
5 }
```

Listing 2.12: Definisanje metoda

Mogu se implementirati i metodi koji ne uzimaju instancu kao argument; njima se pristupa pomoću `::` notacije.

Enumeratori

Enumeratori u programskom jeziku *Rust* imaju dodatne mogućnosti u odnosu na enumeratore u drugim programskim jezicima: svaki "tip" u enumeratoru može se posmatrati kao torka vrednosti. Ovime se, na primer, postiže ponašanje enumeratora `Option` koji premošćava izostanak vrednosti `null`. `Option` je definisan na sledeći način:

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

Listing 2.13: Definisanje enumeratora

Ono što `Option` nudi jeste mogućnost izostanka neke vrednosti, ali da taj izostanak ne može uticati na integritet programa. Pre nego što se koristi, vrednost tipa `Option` se mora otpakovati (engl. *unwrap*), odnosno, mora se utvrditi da li je vrednost generičkog tipa `T` izostala. Otpakivanje se vrši `match` naredbom kao u listingu 2.14.

```
fn main() {
    let a: Option<i8> = Some(10);

    match a {
        Some(x) => println!("Vrednost promenljive a je: {}", x),
        None => println!("Promenljiva a nema vrednost!"),
    }
}
```

Listing 2.14: Korišćenje `match` naredbe

Ukoliko se promenljiva ne otpakuje, kompilator će ispisati odgovarajuću grešku. Slično, greška će se ispisati i kada naredba `match` ne pokrije sve moguće vrste enumeratora, gde je moguće koristiti `_` notaciju za obeležavanje podrazumevane grane.

Strukture podataka promenljivog sadržaja

Pored torki i nizova, *Rust* ima podršku za strukture podataka kojima se sadržaj može menjati: to su vektori i heš mape.

Vektori

Vektori su predstavljeni tipom `Vec<T>`, gde je `T` generički tip podataka sadržanih u vektoru. Novi vektor se inicijalizuje funkcijom `Vec::new` ili korišćenjem naredbe `vec!` kao u listingu 2.15. U istom listingu su prikazane funkcije dodavanja odnosno brisanja elemenata u vektoru: `push`, odnosno `remove`. Poziv funkcije `remove(i)` briše element koji se nalazi na poziciji i u vektoru. Da bi elementi vektora mogli menjati njihovu unutrašnju strukturu, mora se dodati ključna reč `mut` prilikom inicijalizacije. Vektorima koji su definisani bez te ključne reči se i dalje mogu dodavati ili brisati elementi, ali naknadne promene tih elemenata nisu dozvoljene.

Pravila vlasništva su i dalje na snazi, pa se prilikom iteriranja kroz petlju koriste reference na elemente vektora. Da bi se te reference mogle menjati, prvo ih je potrebno dereferencirati operatorom `*`.

```
fn main() {
    let mut v: Vec<i32> = vec![4, 77, 10, 923];
    v.push(50);
    v.push(33);

    v.remove(2);

    for elem in &mut v {
        *elem *= 10;
        print!("{} ", elem);
    }
}
```

Listing 2.15: Inicijalizacija, dodavanje i brisanje elemenata iz vektora

Kao i u nizu, u vektoru se moraju naći elementi istog tipa. Međutim, ovo ograničenje se može prevazići korišćenjem enumeratora, pošto je kompilatoru neophodno samo da ima informaciju koji se tipovi mogu naći u vektoru da bi se mogla rezervisati adekvatna količina memorije.

Heš mape

Za razliku od vektora, deo standardne biblioteke koja implementira heš mape se mora ručno uključiti u program naredbom `use`. Funkcija `insert` će uneti odgovarajući par u mapu, ali će postaviti novu vrednost, bez obzira da li se ključ već nalazi u mapi.

Korišćenjem funkcije `entry`, može se dobiti enumerator `Entry` koji ukazuje da li vrednost za dati ključ postoji u mapi. Ako se na `Entry` nadoveže funkcija `or_insert`, ona će uneti odgovarajući par u mapu samo ako taj ključ ne postoji u mapi. Nakon toga, `or_insert` će vratiti referencu na par koji je napravljen, ili koji je nađen u mapi, kao u listingu 2.16. U istom listingu se koristi naredba `println!` sa argumentom `{:?}`, koja ispisuje odgovarajuće informacije za debagovanje; u ovom slučaju, biće ispisano `{„Pas": 7, „Macka": 3}`.

```
fn main() {
    use std::collections::HashMap;
```

```
3
4       let mut m = HashMap::new();
5
6       m.insert(String::from("Pas"), 2);
7       m.insert(String::from("Macka"), 3);
8
9       let p = m.entry(String::from("Pas")).or_insert(4);
10      *p += 5;
11
12      println!("{:?}", m);
13  }
```

Listing 2.16: Inicijalizacija heš mape i korišćenje `insert` i `or_insert` funkcija

Generički tipovi i interfejsi

U sekciji 2.3, pomenut je enumerator `Option` koji predstavlja omotač za vrednosti bilo kog tipa, da bi se izbeglo korišćenje nedostajućih vrednosti. Da bi se `Option` mogao koristiti na taj način, moraju se koristiti generički tipovi, koji se navode između znakova `<` i `>`, kao u definiciji enumeratora `Option` u listingu 2.13.

Prilikom definisanja metoda u okviru strukture podataka, kada struktura koristi jedan ili više generičkih tipova, moguće je implementirati metod samo za jedan tip, kao u listingu 2.17, gde se rastojanje između tačaka može izračunati kada su koordinate zapisane u pokretnom zarezu.

```
1   struct Tacka<T> {
2       x: T,
3       y: T
4   }
5
6   impl Tacka<f32> {
7       fn dist(&self, t: &Tacka<f32>) -> f32 {
8           ((self.x - t.x).powi(2) + (self.y - t.y).powi(2)).sqrt()
9       }
10  }
11
12  fn main() {
13      let t1 = Tacka {
```

```
14          x: 32.0,
15          y: 40.0
16      };
17      let t2 = Tacka {
18          x: 13.0,
19          y: 51.0
20      };
21
22      println!("Rastojanje izmedju tacaka je: {}", t1.dist(&t2));
23  }
```

Listing 2.17: Korišćenje generičkog tipa prilikom definisanja strukture

Korišćenje generičkih tipova ne prouzrokuje pad performansi, zato što će se prilikom prevođenja koda izvršiti monomorfizacija. Monomorfizacija podrazumeva pretvaranje generičkih tipova u konkretne tipove koji se koriste u programu, time garantujući da će prevedeni kôd biti podjednako efikasan kao i kôd koji ne koristi generičke tipove.

Interfejsi (engl. *traits*) predstavljaju način definisanja zajedničkog ponašanja struktura i enumeratora. U listingu 2.18 je definisan novi interfejs `Objekat2D` koji obavezuje objekte da sadrže implementaciju funkcije `povrsina`.

```
1  use std::f32::consts::PI;
2
3  trait Objekat2D {
4      fn povrsina(&self) -> f32;
5  }
6
7  struct Krug {
8      r: f32,
9  }
10
11 impl Objekat2D for Krug {
12     fn povrsina(&self) -> f32 {
13         self.r * self.r * PI
14     }
15 }
```

Listing 2.18: Definisanje novog interfejsa

U okviru definicije interfejsa se može naći i podrazumevana implementacija metoda. U tom slučaju se metode ne moraju implementirati za pojedinačne strukture podataka.

Generički tipovi i interfejsi se mogu koristiti zajedno tako da se obezbedi ograničenje konkretnih tipova koji se koriste u definiciji funkcija ili struktura. U listingu 2.19 je definisana funkcija koja kao argument uzima samo tipove koji implementiraju interfejs `Objekat2D`.

```
fn povrsina_objekta<T: Objekat2D>(objekat: &T) {
    println!("Povrsina objekta je: {}", objekat.povrsina());
}
```

Listing 2.19: Definisanje funkcije sa ograničenjem generičkih parametara

Za definisanje ograničenja generičkih tipova se može koristiti sintaksička pomoć u vidu ključne reči `where`. U listingu 2.20 je prepravljena funkcija iz listinga 2.19 tako da generički tip mora implementirati interfejs `Display`, čime se obezbeđuje mogućnost korišćenja parametra funkcije u okviru naredbe `println!`. Bitno je napomenuti da se više različitih interfejsa ograničenja deklariše oznakom `+`.

```
fn povrsina_objekta<T>(objekat: &T)
    where T: Objekat2D + Display
{
    println!("Povrsina objekta {} je: {}", objekat, objekat.povrsina());
}
```

Listing 2.20: Definisanje funkcije sa ograničenjem višestrukih generičkih parametara

Životni vek referenci

Svaka referenca u programskom jeziku *Rust* ima svoj životni vek, čak iako on nije eksplicitno naveden. U sekciji 2.3 su pomenute zalutale reference, odnosno reference koje referišu na već oslobođeni prostor. U istoj sekciji je predstavljen i jedan od načina na koji se problem zalutalih referenci može zaobići: korišćenjem statičkog životnog veka. Drugi način rešavanja ovog problema jeste korišćenjem eksplicitno navedenih životnih vekova kao u listingu 2.21. Životni vekovi referenci se obeležavaju sa apostrofom i identifikatorom životnog veka posle znaka `&` i pre tipa reference. Prilikom definisanja funkcije se navode između karaktera `<` i `>` zajedno sa generičkim tipovima.

```rust
fn duza_rec<'a>(s1: &'a str, s2: &'a str) -> &'a str {
    let w1 = s1.split_whitespace()
               .next()
               .unwrap_or("");

    let w2 = s2.split_whitespace()
               .next()
               .unwrap_or("");

    if w1.len() > w2.len() {
        w1
    } else {
        w2
    }
}

fn main() {
    let s1 = String::from("Prva recenica");
    let s2 = String::from("Druga recenica");

    println!("Duza rec je: {}", duza_rec(&s1, &s2));
}
```

Listing 2.21: Definisanje funkcije čiji parametri imaju ekplicitno navedene životne vekove

Nije uvek neophodno eksplicitno deklarisati životni vek svake reference prilikom definisanja funkcije, zato što postoje **pravila spajanja životnih vekova** (engl. *lifetime elision rules*). U pravila spajanja se ubrajaju tri pravila po kojima kompilator automatski određuje životni vek reference:

1. Svaki parametar funkcije koji je referenca dobija jedinstveni identifikator životnog veka.

2. Ako postoji samo jedan parametar funkcije sa životnim vekom i povratna vrednost je takođe referenca, onda će i povratna vrednost imati isti životni vek kao i taj parametar.

3. Ako postoji više parametara funkcije koji su reference, ali jedan od njih je `&self` ili `&mut self` sa životnim vekom `'a`, onda će povratna vrednost, ukoliko je ona referenca, imati životni vek `'a`.

U listingu 2.21 funkcija `duza_rec` ima navedene životne vekove jer kompilator nije u mogućnosti da izvede sve životne vekove samo iz pravila spajanja. Da životni vekovi nisu eksplicitno napisani, oba parametra funkcije bi imala jedinstveni životni vek, a životni vek povratne vrednosti bi ostao nepoznat jer se nijedno pravilo spajanja ne odnosi na njega.

Glava 3

Prilagodljiva radiks-stabla

Razvoj računarske memorije je doveo do situacije da se baze podataka mogu u potpunosti smestiti u radnu sistemsku memoriju. Ovakve baze podataka se nazivaju **memorisane baze podataka** (engl. *in-memory databases*), a njihovi predstavnici su *Apache Ignite*, *VoltDB* i *Redis*. Korišćenjem radne umesto sekundarne memorije dobijaju se značajna poboljšanja u vidu smanjenja odziva, jer odziv sistemske memorije ima za šest redova veličine brže vreme odziva od sekundarne memorije bazirane na magnetnim diskovima.

Za memorisane baze podataka je neophodna struktura podataka koja podržava upite dometa (engl. *range queries*) ali da istovremeno optimalno koristi procesor, pošto vreme odziva više nije problem. Strukture koje se tradicionalno koriste u bazama podataka jesu stabla i druge drvolike strukture. Međutim, problem je u njihovoj neefikasnosti, jer funkcije dodavanja i brisanja elemenata u najboljem slučaju ima asimptotsko vreme izvršavanja $\mathcal{O}(n \log n)$. U ove svrhe su stvorene nadograđene varijante stabala, kao što je struktura podataka FAST (engl. ***F**ast **A**rchitecture **S**ensitive **T**ree*) [8] koja se oslanja na SIMD instrukcije koje omogućavaju paralelno izračunavanje na delu memorije. FAST je efikasniji prilikom pretrage podataka koji su već uneti, ali gubi na robusnosti, pošto se podaci ne mogu efikasno dodavati niti brisati.

Pored stabala i drugih drvolikih struktura, u ove svrhe se koriste i heš mape. Nasuprot stablima, heš mape su efikasne za sve osnovne operacije (dodavanje, brisanje, pretraživanje), sa prosečnom vremenskom složenošću $\mathcal{O}(1)$. Uprkos efikasnosti heš mapa, one nisu dobro rešenje za baze podataka zbog upita dometa. U heš mapama podaci nisu nužno sortirani ili grupisani na bilo koji način, što čini upite dometa, koji dohvataju više podataka sa sličnim ključem, neefikasnim.

Takođe, povećavanje kapaciteta heš mape je skupo i podrazumeva reorganizaciju čitave mape, sa linearnom vremenskom složenošću.

Da bi se ispunili zahtevi memorisanih baza podataka, pojavljuju se tzv. **radiks-stabla** (engl. *radix tree*). Radiks-stabla su drvolika struktura koja, za razliku od stabala pretrage koja porede ključeve, koristi reprezentaciju ključa u binarnom sistemu brojeva. Svaki unutrašnji čvor stabla predstavlja, na primer, jedan bajt ključa, dok se u listovima nalaze vrednosti. Ovom metodom skladištenja sličnih ključeva u istom podstablu rezultuje vremenskom složenošću koja ne zavisi od broja elemenata unetih u strukturu niti od veličine same strukture, već od dužine ključa reprezentovanog u binarnom sistemu. Iako je vremenska složenost linearna, osobina da efikasnost ne zavisi od broja unetih elemenata je pogodna za primenu u memorisanim bazama podataka. Prednost radiks-stabala je i u činjenici da ona, za razliku od stabala pretrage, ne zahtevaju rebalansiranje zarad održavanja visine stabla.

Problem koji se javlja kod radiks-stabala jeste neoptimalno korišćenje memorije. Ukoliko svaki unutrašnji čvor predstavlja jedan bajt ključa, to znači da svaki unutrašnji čvor može imati do 2^8, odnosno 256 pokazivača na decu. Kako svaki čvor neće imati maksimalan broj dece, rezervisanje memorijskog prostora za najgori slučaj rezultuje prekomernim korišćenjem memorije u prosečnom slučaju. Na IEEE konferenciji 2013. godine su predstavljena prilagodljiva radiks-stabla, skraćeno nazvana **ART** [12] (engl. ***A****daptive* ***R****adix* ***T****ree*) kao potencijalno novo rešenje, zato što ona dinamički prilagođavaju veličinu unutrašnjih čvorova u odnosu na broj dece.

3.1 Struktura prilagodljivih radiks-stabala

Unutrašnji čvorovi radiks-stabla ne moraju nužno memorisati jedan bajt ključa. Broj bitova koji se reprezentuje se naziva **opseg** (engl. *span*). To znači da, u svom osnovnom obliku, radiks-stabla imaju visinu koja zavisi od dužine ključa k i od opsega s, i ona iznosi $\lceil \frac{k}{s} \rceil$.

Odabir opsega ima uticaja i na vremensku i na memorijsku složenost radiks stabla. Ukoliko je opseg mali, onda je potencijalni broj dece po unutrašnjem čvoru manji, pa je memorija bolje iskorišćena, ali visina stabla je velika, te su operacije na stablu sporije. I suprotno važi: ukoliko je opseg veliki, onda je visina stabla mala i operacije se vrše brzo, ali je memorijska složenost prevelika. Implementacije

radiks-stabala GPT [3] i LRT [4] koriste četiri, odnosno šest bitova opsega.

ART ima opseg od osam bita, odnosno jedan bajt. Da bi se sprečila prekomerna upotreba memorije, koriste se četiri različita tipa unutrašnjih čvorova u odnosu na maksimalan broj pokazivača na decu:

- `Node4` je najjednostavniji oblik unutrašnjeg čvora i sastoji se od dva vektora dužine 4 za bajtove ključa i pokazivače. Vektor ključeva održava sortirani poredak, zbog mogućnosti korišćenja upita dometa.
- `Node16` koristi dva vektora dužine 16. Na tipu `Node4` se koristi binarna pretraga zato što je vektor na kome se pretraga vrši relativno male veličine, ali nije adekvatno rešenje za vektor od 16 elemenata. Zbog toga se koriste pomenute SIMD instrukcije koje su hardverski podržane od strane procesora.
- `Node48` ne može koristiti SIMD instrukcije zato što su one hardverski ograničene po broju bitova koji vektor može imati. Stoga se koristi vektor ključeva koji sadrži maksimalnih 256 elemenata. Vektor ključeva čuva indekse odgovarajućih elemenata u vektoru pokazivača koji ima kapacitet od 48 elemenata.
- `Node256` koristi samo jedan vektor koji sadrži pokazivače, dok se bajtovi ključeva indirektno čuvaju kao indeksi vektora pokazivača. Ovaj tip koristi više memorije od `Node48`, zato što pokazivači zauzimaju više memorije od čuvanja indeksa koji nisu veći od jednog bajta.

Ukoliko čvor popuni svoj kapacitet, onda se on transformiše u naredni čvor po veličini.

3.2 Memorijska optimizacija unutrašnjih čvorova

Osnovni oblik radiks-stabla podrazumeva korišćenje jednog unutrašnjeg čvora za reprezentaciju jednog bajta ključa. Time se ključevi implicitno čuvaju u samoj strukturi stabla, tj. put od korena stabla do lista predstavlja parcijalni ključ tog lista. Međutim, samo korišćenje jednog unutrašnjeg čvora za svaki bajt ključa ukazuje na neoptimalno iskorišćenu memoriju.

Da bi se memorijska iskorišćenost optimizovala, koristi se metoda **sažimanja puteva** (engl. *path compression*) koja udružuje unutrašnje čvorove koji imaju samo jedno dete sa tim detetom. Korišćenjem ove metode se gubi svojstvo implicitno sačuvanih parcijalnih ključeva, pa se oni moraju čuvati eksplicitno. Postoje dva metoda čuvanja parcijalnih ključeva:

- **Pesimističan**: svaki unutrašnji čvor ima odgovarajući vektor u kome se čuvaju bajtovi parcijalnih ključeva.
- **Optimističan**: svaki unutrašnji čvor čuva informaciju o dužini parcijalnog ključa. Prilikom pretraživanja stabla preskače se dužina parcijalnog ključa u ključu pretrage i pretraga se nastavlja dalje. Kad pretraga stigne do lista, tad se ceo ključ proverava.

Pesimističan metod daje brzu proveru korektnosti putanje, ali sa sobom nosi fragmentaciju memorije. Nasuprot tome, optimističan metod nema problem sa fragmentacijom, ali pretraga može da preduzme pogrešan korak, gde se pretraga mora vratiti unazad, što otežava implementaciju i ima negativan uticaj na performanse. Postoji i hibridan način čuvanja parcijalnih ključeva, gde u unutrašnjim čvorovima čuva fiksni broj bajtova parcijalnog ključa. Sve preko unapred određenog broja bajtova tog vektora ne memoriše, već se koristi optimističan metod čuvanja.

3.3 Algoritmi pretrage i umetanja

U svom radu iz 2013. godine, autori prilagodljivih radiks-stabala su objavili i algoritme pretrage i umetanja. Algoritam pretrage je dat na listingu 3.1.

```
1  search(node, key, depth):
2      if node == NULL
3          return NULL
4      if isLeaf(node)
5          if leafMatches(node, key, depth)
6              return node
7          return NULL
8      if checkPrefix(node, key, depth) != node.prefixLen
9          return NULL
10     depth += node.prefixLen
```

```
11      next = findChild(node, key[depth])
12      return search(next, key, depth + 1)
```

Listing 3.1: Algoritam pretrage strukture podataka ART

Funkcija `search` ima tri parametra: trenutni čvor pretrage, ključ pretrage i dubinu trenutnog čvora. Kako je ovaj algoritam rekurzivan, prvo se proverava bazni slučaj kada je trenutni čvor ili prazan ili list, gde se rezultat vraća samo ukoliko je čvor list i ključ u tom listu se poklapa sa ključem pretrage. U drugom slučaju, ako je pretraga trenutno u unutrašnjem čvoru, prvo se proverava poklapanje parcijalnog prefiksa u čvoru u liniji 5 i pretraga se završava ako dođe do nejednakosti parcijalnih ključeva. Zatim se koriguje dubina pretrage, zato što parametar dubine se odnosi na broj poređenih bajtova u tom trenutku. Na kraju, nalazi se idući čvor funkcijom `findChild` prosleđivanjem trenutnog čvora i odgovarajućeg bajta i pretraga se nastavlja rekurzivno. Algoritam funkcije `findChild` je dat u listingu 3.2.

```
1  findChild(node, byte):
2      if node.type == Node4
3          for (i = 0; i < node.count; i = i + 1)
4              if node.key[i] == byte
5                  return node.child[i]
6          return NULL
7      if node.type == Node16
8          for (i = 0; i < node.count; i = i + 1)
9              key = _mm_set1_epi8(byte)
10             cmp = _mm_cmpeq_epi8(key, node.key)
11             mask = (1 << node.count) - 1
12             bitfield = _mm_movemask_epi8(cmp) & mask
13             if bitfield
14                 return node.child[ctz(bitfield)]
15             else
16                 return NULL
17     if node.type == Node48
18         if node.childIndex[byte] != EMPTY
19             return node.child[node.childIndex[byte]]
20         else
21             return NULL
22     if node.type == Node256
```

```
23          return node.child[byte]
```

Listing 3.2: Algoritam pronalaska idućeg čvora pretrage

Naredni čvor se nalazi na četiri različita načina, u odnosu na tip trenutnog čvora koji je poslat kao parametar funkciji `findChild`:

- **Node4**: naredni čvor se nalazi pretragom svih elemenata vektora ključeva. U ovom slučaju je moguće koristiti i binarnu pretragu, s tim da je poboljšanje zanemarljivo, a niz se, u tom slučaju mora održavati u sortiranom poretku.

- **Node16**: u ovom slučaju se koriste SIMD instrukcije. Prvo se, funkcijom `_mm_set1_epi8` inicijalizuje SIMD vektor od 128 bita, odnosno, 16 elemenata veličine jednog bajta. Svi elementi tog vektora biće jednaki bajtu koji je unet kao parametar. Posle inicijalizacije se porede svi elementi novog vektora i vektora ključeva trenutnog čvora funkcijom `_mm_cmpeq_epi8`. Kao rezultat se dobija vektor iste dužine, ali čiji elementi su binarne vrednosti `0` i `11111111`, u zavisnosti da li su elementi na odgovarajućoj poziciji jednaki ili ne. U liniji 11 se pravi maska zato što vektor elemenata nije nužno pun, ali postoji mogućnost poklapanja bajta pretrage sa bajtovima koji nisu u upotrebi, zato što SIMD vektor mora biti inicijalizovan u potpunosti prilikom njegovog definisanja. Maska je veličine 32 bita, pa se vektor `cmp` šalje funkciji `_mm_movemask_epi8` koja vraća 32-bitnu vrednost gde je svaki element vektora (ukupno ih je 32) reprezentovan svojim najznačajnijim bitom. Kada se primeni maska na dobijenu vrednost, dobija se 32-bitna vrednost `bitfield`, a ključ se nalazi u mestu kojem odgovara bit sa vrednošću 1. Da bi se našao indeks prvog bita sa vrednošću 1, koristi se funkcija `ctz` (engl. ***c**ount **t**railing **z**eros*) koja broji nule na kraju vrednosti zapisane u binarnom sistemu.

- **Node48**: indeks narednog čvora je upisan u vektoru `childIndex` koji ima 256 elemenata, za svaku vrednost bajta pretrage. Ukoliko vrednost za taj ključ nije uneta, u vektor `childIndex` se piše fiksirana vrednost `EMPTY` koja je veća od 48 (maksimalnog broja elemenata u nizu potomaka).

- **Node256**: koristi se niz maksimalne veličine od 256 elemenata gde se bajt ključa direktno koristi za indeksiranje.

Algoritam za umetanje novih parova ključ-vrednost u stablo je dat na listingu 3.3, dok je algoritam brisanja parova, po rečima autora, simetričan ovom algoritmu. Ponovo je u pitanju rekurzivni algoritam, te se prvo obrađuje bazni slučaj, odnosno kad se praznom podstablu dodaje list. Ako je čvor kome se dodaje element list, onda se pravi novi unutrašnji čvor funkcijom `makeNode4` koji će da sadrži čvor koji se dodaje kao i stari list. Pre toga, mora se utvrditi dužina prefiksa u petlji u liniji 8.

Ukoliko stablo nije prazno, niti je u pitanju list, onda je prosleđeni čvor unutrašnji. Ako prefiks unutrašnjeg čvora ne odgovara traženom ključu, onda dolazi do račvanja stabla na dve grane gde se ponavlja slična procedura kao i kod stabla koje sadrži samo jedan list. Na kraju, ako se prefiksi poklapaju, onda se traži sledeći čvor gde bi se pretraga mogla nastaviti. Ako naredni čvor u pretrazi postoji, onda se pretraga rekurzivno nastavlja. U suprotnom, pretraga se zaustavlja i novi list se dodaje trenutnom unutrašnjem čvoru. Trenutni unutrašnji čvor može biti popunjen, pa se njegov kapacitet mora proširiti pretvaranjem trenutnog unutrašnjeg čvora u sledeći čvor u hijerarhiji funkcijom `grow` (iz `Node4` u `Node16`, iz `Node16` u `Node48` ili iz `Node48` u `Node256`).

```
1  insert (node, key, leaf, depth):
2   if node == NULL
3      replace(node, leaf)
4      return
5   if isLeaf(node)
6      newNode=makeNode4()
7      key2=loadKey(node)
8      for (i = depth; key[i] == key2[i]; i = i + 1)
9          newNode.prefix[i - depth] = key[i]
10     newNode.prefixLen = i - depth
11     depth = depth + newNode.prefixLen
12     addChild(newNode, key[depth], leaf)
13     addChild(newNode, key2[depth], node)
14     replace(node, newNode)
15     return
16  p=checkPrefix(node, key, depth)
17  if p != node.prefixLen
18     newNode = makeNode4()
19     addChild(newNode, key[depth + p], leaf)
```

```
20      addChild(newNode, node.prefix[p], node)
21      newNode.prefixLen = p
22      memcpy(newNode.prefix, node.prefix, p)
23      node.prefixLen = node.prefixLen-(p + 1)
24      memmove(node.prefix,node.prefix + p + 1,node.prefixLen)
25      replace(node, newNode)
26      return
27  depth = depth + node.prefixLen
28  next = findChild(node, key[depth])
29  if next
30      insert(next, key, leaf, depth + 1)
31  else
32      if isFull(node)
33          grow(node)
34  addChild(node, key[depth], leaf)
```

Listing 3.3: Algoritam umetanja novih elemenata u stablo

Glava 4

Razrada

Glava 5

Zaključak

Literatura

[1] *Announcing Rust 1.0*. on-line at: https://blog.rust-lang.org/2015/05/15/Rust-1.0.html. 2015.

[2] Matt Asay. *Why AWS loves Rust, and how we'd like to help*. on-line at: https://aws.amazon.com/blogs/opensource/why-aws-loves-rust-and-how-wed-like-to-help/. 2020.

[3] Matthias Boehm i dr. „Efficient in-memory indexing with generalized prefix trees". U: *Datenbanksysteme für Business, Technologie und Web (BTW)*. Ur. Theo Härder i dr. Bonn: Gesellschaft für Informatik e.V., 2011, str. 227–246.

[4] Jonathan Corbet. *Trees I: Radix trees*. on-line at: https://lwn.net/Articles/175432/. 2006.

[5] *crosvm*. on-line at: https://opensource.google/projects/crosvm.

[6] Manish Goregaokar. *Fearless Concurrency in Firefox Quantum*. on-line at: https://blog.rust-lang.org/2017/11/14/Fearless-Concurrency-In-Firefox-Quantum.html. 2017.

[7] *IoT Edge Security Daemon edgelet*. on-line at: https://github.com/Azure/iotedge/tree/master/edgelet.

[8] Changkyu Kim i dr. „FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs". U: *Proceedings of the 2010 ACM SIGMOD Intenational Conference on Management of data* (2010), str. 339–350.

[9] Steve Klabnik. *The History of Rust*. on-line at: https://dl.acm.org/doi/10.1145/2959689.2960081. 2016.

[10] Steve Klabnik i Carol Nichols. *Fearless Concurrency*. on-line at: https://doc.rust-lang.org/book/ch16-00-concurrency.html. 2019.

[11] Steve Klabnik i Carol Nichols. *Functional Language Features: Iterators and Closures*. on-line at: https://doc.rust-lang.org/book/ch13-00-functional-features.html. 2019.

[12] Viktor Leis, Alfons Kemper i Thomas Neumann. „The adaptive radix tree: ARTful indexing for main-memory databases". U: *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 2013, str. 38–49. DOI: 10.1109/ICDE.2013.6544812.

[13] *Rust Case Study: Community makes Rust an easy choice for npm*. on-line at: https://www.rust-lang.org/static/pdfs/Rust-npm-Whitepaper.pdf.

[14] *Rust contributors*. on-line at: https://thanks.rust-lang.org/.

[15] *Stack Overflow Developer Survey*. on-line at: https://insights.stackoverflow.com/survey/2020. 2020.

[16] Barry Warsaw i dr. *PEP 1 – PEP Purpose and Guidelines*. on-line at: https://www.python.org/dev/peps/pep-0001/. 2001.

[17] Ashley Williams. *Hello World!* on-line at: https://foundation.rust-lang.org/posts/2021-02-08-hello-world/. 2021.

Biografija autora

Jovan Dmitrović (*Gornji Milanovac, 17.11.1995.*)