

Rust peliohjelmoinnissa

Victor Bankowski, Antti Karjalainen ja Janne Pulkkinen

Seminaariraportti
HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Helsinki, 17. marraskuuta 2017

Tiedekunta — Fakultet — Faculty		Laitos — Institution — Department	
Matemaattis-luonnontieteellinen		Tietojenkäsittelytieteen laitos	
Tekijä — Författare — Author			
Victor Bankowski, Antti Karjalainen ja Janne Pulkkinen			
Työn nimi — Arbetets titel — Title			
Rust peliohjelmoinnissa			
Oppiaine — Läroämne — Subject			
Tietojenkäsittelytiede			
Työn laji — Arbetets art — Level		Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Seminaariraportti		17. marraskuuta 2017	5
Tiivistelmä — Referat — Abstract			
Tiivistelmä.			

Sisältö

1	Johdanto	1
2	Historia	1
3	Perusteet	2
3.1	Omistajuus	3
3.2	Lainaaminen ja elinajat	4
4	Vertailu C-kieliin	4
5	Seuraava juttu	5
	Lähteet	5

1 Johdanto

Rust on käännettävä ohjelmointikieli, jonka kehitystä tukee Mozilla-säätiö (Anderson et al., 2016). Mozilla käyttää kieltä uuden rinnakkaisuutta hyödyntävän Servo -internet-selainmoottorin ohjelmointiin [ja lisäksi käytetään missä?]. Käännettävänä ohjelmointikielenä C ja C++ -kielten tavoin Rust mahdollistaa suorituskyykyä ja hallittua muistin käyttöä vaativien sovellusten kehittämisen esimerkiksi sulautetuissa järjestelmissä. Edellä mainituista kielistä poiketen Rust kuitenkin estää yleisiä C-kielissä esiintyviä muistinhallintaa ja kilpatilanteita koskevia ongelmia, mahdollistaen kuitenkin vastaavan suorituskyyvyn ajettavassa ohjelmassa. Rust ratkaisee nämä ongelmat käyttämällä muistinhallinnassa omistajuuden (”ownership”) ja lainaamisen (”borrowing”) käsitteitä. Tämä estää mahdolliset virhetilanteet jo ohjelman käännösvaiheessa vaatimatta virtuaalikoneen, kääntäjän tai tulkin käyttöä ohjelman suorituksen aikana.

2 Historia

Rust-kielen kehitys alkoi vuonna 2006 Graydon Hoaren sivuprojektina, jollaisena se jatkui yli kolmen vuoden ajan¹. Mozilla-säätiö osallistui kehitykseen ensimmäisen kerran vuonna 2009 ja on tukenut ohjelmointikielen kehitystä siitä lähtien. Nykyisin kieltä kehittävä ryhmä – *The Rust Team* – jakautuu osaryhmiin, jotka vastaavat kielen eri osa-alueista. Osa-alueisiin kuuluvat esimerkiksi kääntäjän kehittäminen, kielen ominaisuuksien suunnittelu ja dokumentaatio.

Suurimpiin Rustia käyttäviin projekteihin kuuluu Mozillan kehittämä Servo -web-selainmoottori. Sen tavoitteisiin kuuluu sivun piirtämisen, HTML-datan parsimisen ja muiden web-selaimen piiriin kuuluvien tehtävien rinnakkaistaminen². Servo-projektiin kuuluva CSS-moottori Stylo on otettu käyttöön Mozilla Firefox -selaimen uusissa kehitysversioissa³.

Muihin Rust-kieltä hyödyntäviin organisaatioihin kuuluu muun muassa Dropbox ja Canonical⁴.

¹<https://www.rust-lang.org/en-US/faq.html>

²<https://hacks.mozilla.org/2017/08/inside-a-super-fast-css-engine-quantum-css-aka-stylo/>

³<https://blog.mozilla.org/blog/2017/09/26/firefox-quantum-beta-developer-edition/>

⁴<https://www.rust-lang.org/en-US/friends.html>

3 Perusteet

```
fn main() {  
    println!("Hei maailma");  
}
```

```
$ cargo run  
Hei maailma
```

Koodi 1: Tulostaa "Hei maailma"

Ohjelmointikieleen tutustuessa on tapana kirjoittaa klassinen “Hei maailma” -ohjelma joka tulostaa kyseisen lauseen. Koodi 1 on kyseisen ohjelman Rust toteutus. Kyseinen toteutus ei poikkea juurikaan muiden proseduraalisten kielten toteutuksista. Suurin ero muiden kielten toteutuksiin on se että tulostuskomento on makro. Tulostuskomento on toteutettu makrolla tekstin muotoilun helpottamiseksi. Koodissa 2 on esimerkki tästä.

```
fn main() {  
    let s_luku: i32 = 4; // Valittu reilulla napanheitolla.  
    println!("Satunnaislukusi on {}", s_luku);  
}
```

Koodi 2: Tulostaa "Hei maailma"

Rustissa muuttujat määritellään käyttäen **let** avainsanaa ja muuttujan tyyppi erotellaan kaksoipistellä. Koodissa 2 muuttuja nimeltä *s_luku* on 32-bittinen etumerkillinen kokonaisluku. Muuttujat oletusarvoisesti eivät ole muokattavissa. Muokattavat muuttujat määritellään käyttäen avainsanayhdistelmää **let mut**.

```
fn factorial(n: u64) -> u64 {  
    let mut result = 1;  
    for i in 1..(n + 1) {  
        result *= i;  
    }  
    return result;  
}
```

Koodi 3: Funktio joka laskee $n!$ iteratiivisesti.

Useimmissa tapauksissa muuttujan tyyppin voi jättää merkkeamatta, koska Rust osaa päätellä sen käännösaikana. Kuitenkin funktioiden parametrien

ja palautusarvon tyypit täytyy merkata, koska Rustissa ei ole ohjelmanlaajuista tyyppipäättelyä. Funktion palautusarvon tyyppi määritellään nuolen \rightarrow jälkeen. Koodissa 3 funktion parametri n ja palautusarvo ovat 64-bittisiä etumerkittömiä kokonaislukuja.

Rustin **for**-silmukat ovat *for-each*-tyyppisiä, jossa käydään iteraattorin kaikki alkiot läpi. Esimerkiksi koodissa 3 käydään kaikki välin $[1, n + 1)$ kokonaislukuarvot läpi.

Rustissa ei tarvitse käyttää **return** avainsanaa, jos arvo palautetaan funktion lopussa. Tällöin ei myöskään merkata puolipistettä.

```
fn factorial(n: u64) -> u64 {
    if n == 0 {
        1
    } else {
        n * factorial(n-1)
    }
}
```

Koodi 4: Funktio joka laskee $n!$ rekursiivisesti.

3.1 Omistajuus

Muistinhallinta on tärkeä osa ohjelmien toimintaa. Tätä varten monissa kielissä käytetään automaattista roskienkeruuta. Tämä vähentää ohjelmoijan vastuuta, mutta samalla myös vähentää mahdollisuuksia vaikuttaa muistinhallintaan. Tietyissä suorituskyykriittisissä ongelmissa automaattinen roskienkeruu saattaa tehdä epäoptimaalisia ratkaisuja. Tällaisia ongelmia esiintyy pelimootoreissa esimerkiksi fysiikanmallinmuksessa. Tämän takia pelimootorit toteutetaan usein kielillä joissa ei muistinhallinta on manuaalista. Tämän kaltaiset kielissä jää ohjelmoijan vastuuksi varata ja vapauttaa muisti oikeaoppisesti, josta johtuen ne ovat alttiita muistihallintavirheille. Rust-ohjelmointikielessä ei käytetä automaattista roskienkeruuta, mutta siinä ei myöskään jätetä muistinhallintaa täysin ohjelmoijan vastuulle. Tämä on mahdollista omistajuuskäsitteen avulla.

Ohjelmointikielet vaativat muuttujien luomisen lisäksi niiden poistamisen. Käännetyissä kielissä tämä voi vaatia ohjelmoijalta muuttujien manuaalisen poistamista, kun niitä ei enää käytetä. Esimerkiksi C käyttää manuaalista muistinhallintaa. Useissa tulkkia käyttävissä kielissä muuttujaan kohdistuvia viittauksia lasketaan suorituksen aikana; kun muuttujaan ei enää viitata, automaattinen roskienkeräys ennen pitkää löytää käyttämättömän muuttujan ja poistaa sen. Tämä ei vaadi ohjelmoijalta yleensä lisätoita, mutta roskienkeräjä lisää ohjelman resurssivaatimuksia. Roskienkeräystä käyttäviin kielisiin kuuluvat muun muassa Python ja Java.

Rust varmistaa muuttujien muistinhallinnan käyttämällä omistamisen käsitettä, joka ei vaadi suorituksen aikaisen roskankeruun käyttöä, eikä myöskään muuttujien manuaalista poistamista. Kun muuttuja luodaan missä tahansa osassa koodia, sille määritellään omistaja muuttujan vaikutusalueen perusteella. Esimerkiksi metodissa main luodut muuttujat saavat omistajakseen kyseisen metodin. Kun kyseisen main-metodin suoritus on päättynyt, kyseisen metodin omistamat muuttujat poistetaan välittömästi ja niiden varaama muistialue palautetaan järjestelmän käyttöön. Tätä periaatetta on havainnollistettu esimerkissä 2.

3.2 Lainaaminen ja elinajat

Rust-kielen muistinhallintaan kuuluu omistajuuden lisäksi kaksi merkittävää alakäsitettä: lainaaminen ja elin aika. Lainaaminen tarkoittaa muuttujan viittauksen antamista väliaikaisesti johonkin toiseen skoppiin; esimerkiksi toiseen metodiin. Useimmista kielistä poiketen Rust kuitenkin määrittää invariantin, joka tarkastetaan käännösvaiheessa: muuttujalla voi olla joko useita muuttumattomia (vain-luku) viittauksia, yksi muuttava (luku ja kirjoitus) viittaus, mutta ei kummankin tyyppistä viittauksia samanaikaisesti. Tällä estetään virhetilanteet, jossa yksi tai useampi taho lukee muuttujaa samaan aikaan kun sitä muutetaan. Kielen standardikirjastossa on saatavilla erilaisia synkronointialkioita, jotka sallivat esimerkiksi ”yksi kirjoittaja, usea lukija” -malliset tilanteet.

4 Vertailu C-kieliin

C++ ja C-ohjelmointikielien ovat Rustin tavoin käännettäviä ohjelmointikieliä, jotka soveltuvat suorituskykyä vaativien sovellusten kehittämiseen. Näihin kuuluvat muun muassa käyttöjärjestelmät, sulautetut järjestelmät ja pelimoottorit, joista esimerkiksi *Unity* ja *Unreal Engine* on kirjoitettu C++-kieltä käyttäen. Rustille on saatavilla pelimoottorikirjastoja kuten *Piston*, mutta luontityökaluja sisältävää pelinkehityskokonaisuutta ei toistaiseksi ole saatavilla Rust-kielille. (*Are we game yet? - Rust*, s.a.)

Rust-lähdekoodin kääntämiseen käytetään `rustc` -työkalua, joka myös on kirjoitettu Rust-kielillä. Kääntäjä hyödyntää LLVM-kääntäjäinfrastruktuurin työkaluja, ja hyötyy siten kyseistä projektia koskevista suorituskykyparannuksista. (*Frequently Asked Questions - The Rust Programming Language*, s.a.) C++ -kielestä poiketen Rustille ei ole kuitenkaan saatavilla vaihtoehtoisia kääntäjiä. Laajassa käytössä oleviin C++ -kielen kääntäjiin kuuluu muun muassa LLVM-projektin *Clang*, *GNU GCC* ja *Intel C++ Compiler*.

C++ -kieli ja sitä edeltävä C-kieli ovat yleisiä käyttöjärjestelmien ohjelmointikielinä: *Microsoft Windows* on ohjelmoitu C++ -kielillä, *FreeBSD* on ohjelmoitu käyttäen sekä C++ ja C-kieliä ja *Linux* käyttää C-kieltä. Rust olisi muistinhallinnan puolesta otollinen käyttöjärjestelmän ytimen

ohjelmointiin, jossa muistinhallinta-virheillä voi olla vakavia seurauksia järjestelmän vakauden ja tietoturvallisuuden kannalta. *Redox* on Rust-kielillä kirjoitettu mikroydin-rakennetta hyödyntävä käyttöjärjestelmä. (*The Redox Operating-System*, s.a.) *Redox* on vielä aikaisessa kehitysvaiheessa eikä siten sovellu jokapäiväiseen käyttöön.

Koska Rust-kieleen ei kuulu ajonaikaista virtuaalikonetta tai muuta tulkia, Rust-koodia on C-kielten tavoin mahdollista hyödyntää korkeamman tason kielissä kuten Pythonissa tai Rubyssa. Tässä tapauksessa sovelluksen tehokasta suorituskkyä ja muistinkäyttöä vaativat osat voidaan kirjoittaa Rustilla. (*Rust Inside Other Languages - Rust*, s.a.) C-kielillä kirjoitettuja kirjastoja on myös mahdollista käyttää Rustilla toteuttamalla kirjastolle sidonnat, jotka käyttävät kirjaston funktioita FFI (*Foreign Function Interface*) -rajapinnan kautta. C-kielillä kirjastoja ei siis tarvitse uudelleenkirjoittaa, jotta niitä voidaan hyödyntää Rust-kielessä. Tässä tapauksessa ohjelmoinjan tulee luoda kirjastolle turvalliset sidonnat, jotka estävät mahdolliset virhetilanteet esimerkiksi muistinhallinnan osalta.

5 Seuraava juttu

Lähteet

Anderson, B., Bergstrom, L., Goregaokar, M., Matthews, J., McAllister, K., Moffitt, J., & Sapin, S. (2016). Engineering the servo web browser engine using rust. Teoksessa *Proceedings of the 38th international conference on software engineering companion* (s. 81–89). New York, NY, USA: ACM. doi: 10.1145/2889160.2889229

Are we game yet? - rust. (s.a.). Lainattu 2017-10-26, saatavilla <http://arewegameyet.com/categories/engines.html>

Frequently asked questions - the rust programming language. (s.a.). Lainattu 2017-10-26, saatavilla <https://www.rust-lang.org/en-US/faq.html#how-fast-is-rust>

The redox operating-system. (s.a.). Lainattu 2017-10-26, saatavilla https://doc.redox-os.org/book/overview/what_redox_is.html

Rust inside other languages - rust. (s.a.). Lainattu 2017-11-01, saatavilla <https://doc.rust-lang.org/1.2.0/book/rust-inside-other-languages.html>