Zbornik radova Fakulteta tehničkih nauka, Novi Sad

UDK: 4.9
DOI: https://doi.org/10.24867/33BE08Bajat

MEĐUREPREZENTACIJE IZVORNOG KODA RUSTC KOMPAJLERA

INTERMEDIATE REPRESENTATIONS OF THE SOURCE CODE IN THE RUSTC COMPILER

Aleksa Bajat, *Fakultet tehničkih nauka, Novi Sad*

Oblast – ELEKTROTEHNIČKO I RAČUNARSKO INŽENJERSTVO

Kratak sadržaj – *Ovaj rad istražuje arhitekturu prednjeg kraja (engl. frontend) Rust kompajlera (rustc), sa fokusom na ključnu ulogu koju međureprezentacije izvornog koda igraju u procesu prevođenja. Analizirane su faze od leksičke analize i parsiranja do generisanja Apstraktnog Sintaksnog Stabla (ASS), preko Međureprezentacije Visokog Nivoa (MVN/HIR), Tipizirane MVN (TMVN/THIR), do Međureprezentacije Srednjeg Nivoa (MSN/MIR). Rad objašnjava kako svaka od ovih reprezentacija omogućava ključne funkcionalnosti Rust jezika, uključujući proveru tipova, dijagnostiku grešaka, inkrementalno kompajliranje, proveru pozajmljivanja (borrow checking) i pripremu za dalju optimizaciju i generaciju koda u LLVM-u. Cilj je da se prikaže kako generisanje infrastrukture doprinosi garancijama memorijske bezbednosti i visokim performansama Rust programskog jezika.*

Ključne reči: *Rust, rustc, kompajler, frontend, međureprezentacije, ASS, MVN, TMVN, MSN, LLVM, analiza koda, optimizacija*

Abstract – *This paper explores the architecture of the Rust compiler's (rustc) frontend, focusing on the crucial role of intermediate representations (IRs) of source code in the compilation process. Phases from lexical analysis and parsing to the generation of the Abstract Syntax Tree (AST), through the High-Level Intermediate Representation (HIR), Typed HIR (THIR), to the Mid-Level Intermediate Representation (MIR) are analyzed. The paper explains how each of these representations enables key features of the Rust language, including type checking, error diagnostics, incremental compilation, borrow checking, and preparation for further optimization and code generation in LLVM. The aim is to demonstrate how generation of compiler infrastructure contributes to Rust's guarantees of memory safety and high performance.*

Keywords: *Rust, rustc, compiler, frontend, intermediate representations, AST, HIR, THIR, MIR, LLVM, code analysis, optimization*

NAPOMENA:
Ovaj rad proistekao je iz master rada čiji mentor je bio dr Dunja Vrbaški, doc.

1. UVOD

Programski jezik *Rust* je stekao značajnu popularnost zahvaljujući svom fokusu na memorijsku bezbednost bez upotrebe sakupljača smeća (engl. *garbage collector*) i visokim performansama koje pariraju jezicima kao što su *C* i *C++*. Ove karakteristike čine ga idealnim za razvoj sistemskog softvera, operativnih sistema, veb servera i drugih aplikacija gde su performanse i bezbednost kritične [1]. Postizanje ovih ciljeva zahteva sofisticiran kompajler, rustc, čiji prednji kraj (frontend) igra ključnu ulogu u analizi, validaciji i transformaciji izvornog koda.

Prednji kraj rustc kompajlera koristi niz međureprezentacija (engl. *Intermediate Representations)* kako bi postepeno transformisao izvorni kod u formu pogodnu za dalje faze kompajliranja, uključujući optimizaciju i generaciju koda od strane *LLVM* bekenda [13]. Svaka međureprezentacija služi specifičnoj svrsi, omogućavajući različite vrste analiza i transformacija. Razumevanje ovih reprezentacija je ključno za razumevanje kako *Rust* postiže svoje garancije.

2. PUT IZVORNOG KODA U RUSTC KOMPAJLERU

Proces kompajliranja u rustc frontendu započinje leksičkom analizom i parsiranjem, nastavlja se kroz nekoliko nivoa međureprezentacija gde se vrše ključne analize i transformacije, a završava se generacijom Međureprezentacije Srednjeg Nivoa (MSN) koja se potom prosleđuje *LLVM*-u za dalju optimizaciju i generaciju mašinskog koda. U tipično struktuiranom kompajleru svaka od međureprezentacija predstavlja poseban program i posebno izvršavanje. Ovakvi kompajleri se nazivaju kompajleri zasnovani na prolascima - arhitektura cevi. *Rust* kompajler je u tranziciji između kompajlera zasnovanog na prolascima i kompajlera zasnovanog na potražnji. Kompajler zasnovan na potražnji koristi upite nad izvornim kodom i simultono izvršava velike celine kompajliranja. Izvršavanje upita je memoizovano. Samo prvi poziv upita izvršava komputacije, dok je svaki naredni keširan. Sve međureprezentacije nakon ASS se zasnivaju na sistemu upita, tj. samo lekser i parser rade po principu arhitekture cevi. Dugoročni cilj je refaktorizacija kompajlera tako da celokupan proces radi na osnovu sistema upita.

2.1. OD TOKENA DO APSTRAKTNOG SINTAKSNOG STABLA (ASS)

Prvi korak je leksička analiza, gde rustc_lexer, kao lekser niskog nivoa, čita izvorni kod kao niz karaktera, grupiše ih u lekseme (npr. ključne reči, identifikatori, operatori) i proizvodi tok tokena. Svaki token obično sadrži tip i, opciono, vrednost. *Rust*-ov lekser je ručno implementiran, što omogućava finu kontrolu nad procesom tokenizacije i rezultira detaljnijim i korisnijim porukama o greškama.

Nakon toga, *rustc_parse::lexer*, lekser višeg nivoa, dodatno obrađuje ove tokene. U ovoj fazi, vrši se interniranje simbola (engl. symbol interning), gde se stringovi poput identifikatora smeštaju u posebnu memorijsku oblast (arenu) tako da svaki jedinstveni string ima samo jednu kopiju. Ovo optimizuje upotrebu memorije i ubrzava poređenje simbola. Takođe se koriste strukture kao što su Span i SpanData za praćenje lokacije svakog tokena u izvornom kodu, što je ključno za preciznu dijagnostiku grešaka. Pre samog parsiranja, vrši se i rezolucija zagrada, gde se tok tokena strukturira u stabla tokena (*TokenTree*), olakšavajući kasniju obradu.

Parser zatim koristi ovaj obrađeni tok tokena da generiše Apstraktno Sintaksno Stablo (ASS ili AST), koje predstavlja hijerarhijsku strukturu izvornog koda [3]. ASS verno odražava strukturu korisničkog koda, eksplicitno prikazujući, na primer, prvenstvo operatora. Koren ASS-a je obično Crate struktura, dok su osnovni gradivni blokovi *Item*-i (npr. funkcije, strukture, moduli), koji sadrže atribute, jedinstveni identifikator (*NodeId*) i *Span*. Iako je ASS osnova za dalje analize, nije direktno pogodan za sve operacije zbog svoje bliskosti originalnom izvornom kodu. U ovoj fazi se vrši i ekspanzija makroa – moćne *Rust*-ove karakteristike koja omogućava metaprogramiranje ("kod koji generiše kod") i smanjuje potrebu za ponavljajućim kodom. Takođe, započinje inicijalna rezolucija imena, gde kompajler pokušava da poveže imena korišćena u kodu sa njihovim definicijama, koristeći koncept "rebara" (engl. *ribs*) za upravljanje vidljivošću imena u različitim opsezima (engl. *scopes*). Pre prelaska na sledeću reprezentaciju, ASS se validira, posebno nakon ekspanzije makroa koji mogu generisati sintaktički nekompletan ili neispravan kod.

2.2. MEĐUREPREZENTACIJA VISOKOG NIVOA (MVN ili HIR)

ASS se zatim "snižava" (engl. *lowering*) u Međureprezentaciju Visokog Nivoa (MVN) [4]. MVN je apstraktnija reprezentacija koja pojednostavljuje mnoge konstrukcije iz ASS-a; na primer, for petlje se prevode u while let konstrukcije, if let izrazi u match izraze, a impl Trait u parametrima funkcija u odgovarajuće generičke argumente. Svrha ovog snižavanja je da se smanji broj različitih sintaksnih formi (sintaksni šećer) sa kojima kasnije faze kompajlera moraju da rade, čime se pojednostavljuje analiza. HIR Crate struktura sadrži informacije o celokupnom kodu paketa, uključujući i delove koji možda nisu direktno pozvani, što je važno za analizu eksternih biblioteka. HIR je ključan za sistem upita (engl. *query system*) rustc-a. Sistem upita funkcioniše kao baza podataka kompajlera gde se rezultati različitih analiza (upita) keširaju (memoizacija). Svaki upit je funkcija koja mapira jedinstveni ključ (npr. identifikator stavke) na rezultat (npr. tip stavke, telo funkcije u MVN-u). Ako se upit ponovo pozove sa istim ključem, vraća se keširani rezultat umesto ponovnog izračunavanja. Ovo je temelj inkrementalnog kompajliranja: prilikom izmene koda, samo oni upiti čije su zavisnosti promenjene moraju ponovo da se izvrše. Sistem upita zahteva da provajderi upita budu determinističke funkcije i održava direkcioni aciklični graf (DAG) poziva upita kako bi se izbegli ciklusi. Za stabilnu identifikaciju čvorova grafa između različitih kompajliranja koriste se *DefPath* i *DefPathHash* strukture, kao i "otisci prsta" (engl. *fingerprints*) za efikasno poređenje stanja. Na MVN-u se vrše mnoge semantičke analize, kao što su rezolucija osobina (engl. *trait resolution*) i provere koherentnosti.

2.3. TIPIZIRANA MEĐUREPREZENTACIJA VISOKOG NIVOA (TMVN ili THIR)

Nakon MVN-a, generiše se Tipizirana Međureprezentacija Visokog Nivoa (TMVN). TMVN obogaćuje MVN informacijama o tipovima za svaki izraz i deo koda unutar tela funkcija; dakle, TMVN se generiše samo za izvršni kod [5]. TMVN je efemerna reprezentacija, što znači da se delovi generišu "na zahtev" i ne čuvaju se kompletno u memoriji tokom celog procesa, čime se smanjuje memorijski otisak kompajlera. U TMVN-u, mnoge implicitne operacije iz izvornog koda postaju eksplicitne: automatska referenciranja i dereferenciranja, pozivi metoda na osobinama, i preklopljeni operatori se prevode u eksplicitne pozive funkcija. Takođe, uništenje opsega (engl. scope destruction) je eksplicitno predstavljeno.

TMVN služi kao osnova za dve važne provere:

1. **Provera bezbednosti (engl. *check_unsafety*)**: Algoritam analizira unsafe kontekste, proverava da li se unsafe operacije (npr. dereferenciranje sirovog pokazivača) pozivaju van unsafe bloka, i da li unsafe blokovi zaista sadrže unsafe kod (ako ne, generiše se upozorenje - *lint*).
2. **Provera iscrpnosti šablona (engl. *pattern matching exhaustiveness*):** Za konstrukcije kao što su match, if let, while let, let else, pa čak i za argumente funkcija koji koriste destrukturiranje, TMVN omogućava proveru da li su svi mogući slučajevi pokriveni. Pored iscrpnosti, proverava se i "korisnost" svake grane, kako bi se detektovale nedostižne (redundantne) grane koda.

2.4. MEĐUREPREZENTACIJA SREDNJEG NIVOA (MVN ili HIR)

Konačna reprezentacija u frontendu je Međureprezentacija Srednjeg Nivoa (MIR). Uvođenje MSN je bilo motivisano potrebom za preciznijom kontrolom nad tokom izvršavanja, lakšim sprovođenjem Rust-specifičnih optimizacija i pojednostavljivanjem procesa dokazivanja memorijske bezbednosti, što je bilo teško postići direktnim prevođenjem sa MVN-a ili TMVN-a na LLVM IR. MSN je eksplicitna reprezentacija kontrolnog toka (control flow graph - CFG) programa [5]. CFG se sastoji od osnovnih

blokova (engl. basic blocks), gde svaki blok predstavlja niz naredbi koje se izvršavaju sekvencijalno, a poslednja naredba u bloku, terminator (engl. terminator), određuje u koji sledeći blok (ili blokove) će se preći. MSN koristi LIFO stek za smeštanje argumenata funkcija, lokalnih promenljivih i privremenih vrednosti. Memorijske lokacije na steku se identifikuju preko "mesta" (engl. places, npr. _1 za prvu lokalnu promenljivu, _0 za povratnu vrednost), a pristupi poljima struktura ili dereferenciranje se predstavljaju kao "projekcije" (npr. _1.polje, *_1). Izrazi koji generišu vrednosti nazivaju se "desne vrednosti" (RValues).

U neoptimizovanom MSN, eksplicitno se koriste iskazi StorageLive i StorageDead kako bi se označio početak i kraj životnog veka svake lokalne promenljive, što kompajleru daje jasnu informaciju o tome kada je memorija za datu promenljivu validna i kada se može bezbedno osloboditi.

Na MSN se vrši ključna analiza za Rust: analiza pozajmljivanja (*borrow checking*). Ovo uključuje implementaciju Neleksičkih Životnih Vekova (*Non-Lexical Lifetimes - NLL*). Tradicionalni leksički životni vekovi su vezani za opsege u kodu i mogu biti previše restriktivni. *NLL*, s druge strane, analizira stvarnu upotrebu pozajmica unutar grafa kontrole toka, omogućavajući preciznije i fleksibilnije određivanje tačnog trajanja svake pozajmice, što često rezultira prihvatanjem koda koji bi sa leksičkim životnim vekovima bio odbijen. MSN takođe služi za Rust-specifične optimizacije; na primer, kompleksne petlje ili match izrazi mogu biti dodatno pojednostavljeni (npr. for petlja se preko *loop* { *match* ... } sa *goto* naredbama može transformisati u efikasnije switch konstrukcije) pre nego što se kod prosledi *LLVM* bekendu za dalje, generalnije optimizacije i konačnu generaciju mašinskog koda.

3. ZNAČAJ MEĐUREPREZENTACIJA

Svaka od navedenih međureprezentacija u rustc frontendu ima svoju specifičnu i ključnu ulogu:

- **ASS:** Predstavlja vernu reprezentaciju izvornog koda, služi kao osnova za makro ekspanziju i inicijalnu rezoluciju imena. Njegova bliskost izvornom kodu omogućava precizne poruke o sintaksnim greškama.
- **MVN:** Kao apstrakcija nad ASS-om, pojednostavljuje strukturu koda i ključna je za rad sistema upita, omogućavajući semantičke analize i inkrementalno kompajliranje.
- **TMVN:** Obogaćujući HIR tipovima, omogućava detaljnu proveru tipova, proveru bezbednosti unsafe koda i iscrpnosti i korisnosti šablona, što su ključne komponente Rust-ove pouzdanosti.
- **MSN:** Kao eksplicitni graf kontrolnog toka, fundamentalan je za analizu pozajmljivanja (borrow checking) i implementaciju Neleksičkih Životnih Vekova (NLL), što direktno doprinosi memorijskoj bezbednosti. Takođe, omogućava Rust-specifične optimizacije.

Ova višefazna arhitektura, gde svaka IR rešava specifičan skup problema, omogućava rustc-u da efikasno sprovodi kompleksne analize neophodne za garantovanje memorijske bezbednosti i generisanje efikasnog koda, što su dve ključne prednosti Rust jezika. Modularnost pristupa olakšava razvoj i održavanje kompajlera, dok sistem upita dodatno doprinosi ukupnoj efikasnosti kroz memoizaciju i inkrementalno kompajliranje. Precizne dijagnostičke poruke su takođe rezultat mogućnosti da se greške lociraju u odgovarajućoj fazi i na odgovarajućem nivou apstrakcije.

4. BEZBEDNO NEBEZBEDAN

Analiza je provedena 2023. godine da se 12% paketa u *Rust* ekosistemu direktno oslanja na nestabilne funkcionalnosti, dok je 44% paketa indirektno zavisilo od nestabilnih funkcionalnosti da bi se kompajliralo. Procentualna zavisnost prema nestabilnim funkcionalnostima je visoka ali analiza nije sprovedena da se izračuna procenat masivno korišćenih paketa sa direktnim ili tranzitivnim zavisonstima, kao i procenat nestabilnih funkcionalnosti na kojima se ovakvi paketi zasnivaju [9].

Kapije funkcionalnosti su mehanizam na osnovu kog se kontroliše vidljivost funkcionalnosti u određenom skupu alata (*stable*, *beta*, *nightly*). Funkcionalnost može biti prihvaćena, nestabilna, nezavršena ili obrisana. Kapije funkcionalnosti se ne brišu fizički iz koda, već uz adekvatan opis služe kao perzistentno obrazloženje odluke da funkcionalnost nije podobna za razvoj.

Odobravanje funkcionalnosti je rigorozan ali transparentan proces. Svaka značajna promena koja nije refaktorizacija ili dokumentovanje mora proći kroz sledeće faze:

1. **Zahtev za komentare (RFC):** Proces započinje kreiranjem **RFC dokumenta** koji detaljno obrazlaže svrhu nove funkcionalnosti i njen opšti dizajn. Ovaj dokument je javan i podložan diskusiji od strane Rust tima i celokupne zajednice. Odobrenje RFC-a daje zeleno svetlo za početak razvoja, ali ne garantuje konačno prihvatanje.
2. **Razvoj i testiranje:** Nakon odobrenja, funkcionalnost se implementira i detaljno testira.
3. **Proces stabilizacije:** Kada je funkcionalnost razvijena i testirana bez značajnih primedbi, pokreće se formalni zahtev za stabilizaciju, koji se sastoji iz četiri ključna dela:
    - **Ažuriranje dokumentacije:** Dokumentacija se premešta iz interne "nestabilne knjige" (*Unstable Book*) u zvaničnu dokumentaciju za korisnike (*Rust Reference*).
    - **Stabilizacioni izveštaj:** Kreira se izveštaj koji sadrži primere korišćenja, linkove ka dokumentaciji i testove koji pokrivaju granične slučajeve.
    - **Period finalnog komentara (*FCP*):** Tim ponovo pregleda ceo predlog kako bi se postigao konačni konsenzus.
    - **Stabilizacioni *Pull Request*:** Ukoliko je konsenzus pozitivan, kreira se finalni

zahtev za povlačenjem (*pull request*). Njegov cilj je da tehnički omogući funkcionalnost u stabilnoj verziji jezika, uklanjajući oznaku nestabilnosti i grešku koja sprečava njeno korišćenje van noćne (*nightly*) verzije kompajlera.

4. **Beta i stabilna verzija:** Nakon uspešne stabilizacije, funkcionalnost postaje dostupna korisnicima u **beta verziji** za finalno testiranje, pre nego što konačno postane deo sledeće **stabilne distribucije** Rust-a.

4. ZAKLJUČAK

Međureprezentacije izvornog koda u prednjem kraju rustc kompajlera čine složen, ali visoko efikasan sistem. Kroz pažljivo dizajnirane faze transformacije i analize – od ASS-a, preko MVN-a i TMVN-a, do MSN-a – kompajler postepeno prevodi, proverava i optimizuje korisnički kod. Ova slojevita arhitektura je temelj na kojem Rust gradi svoje jedinstvene prednosti: garantovanu memorijsku bezbednost bez sakupljača smeća, konkurentnost bez rizika od "data races", i visoke performanse uporedive sa C i C++.

Svaka izmena ili proširenje *Rust* kompajlera podleže rigoroznom i transparentnom procesu prihvatanja. Kroz mehanizam Zahteva za komentare (RFC) i višefaznu stabilizaciju, zajednica osigurava da se jezik razvija na kontrolisan način, čuvajući integritet i bezbednosne garancije koje kompajler pruža.

Razumevanje ovih internih mehanizama i uloge svake međureprezentacije je od velikog značaja, ne samo za dalji razvoj i unapređenje samog kompajlera (npr. poboljšanje vremena kompajliranja ili uvođenje novih optimizacija), već i za dublje razumevanje ponašanja Rust programa i efikasnije korišćenje naprednih mogućnosti jezika.

5. LITERATURA

[1] MSRC, “A proactive approach to more secure code | MSRC Blog | Microsoft Security Response Center,” Microsoft.com, Jul. 16, 2019. https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/ (pristupljeno u septembru 2024.)

[2] “Parsing” Rochester.edu, 2024. https://www.cs.rochester.edu/u/nelson/courses/csc_173/grammars/parsing.html#:~:text=Recursive%2Ddescent%20parsing%20is%20one,non%2Dterminal%20with%20a%20procedure (pristupljeno u septembru 2024.)

[3] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Boston, MA, USA: Addison-Wesley, 2006.

[4] “Getting Started - Rust Compiler Development Guide,” Rust-lang.org, 2024. https://rustc-dev-guide.rust-lang.org/getting-started.html (pristupljeno u Septembru 2024.)

[5] “Introduction - The Rust Reference,” Rust-lang.org, 2015. https://doc.rust-lang.org/reference/introduction.html (pristupljeno u septembru 2024.)

[6] “Meet Safe and Unsafe - The Rustonomicon,” Rust-lang.org, 2024. https://doc.rust-lang.org/nomicon/meet-safe-and-unsafe.html (pristupljeno u septembru 2024)

[7] “What are editions? - The Rust Edition Guide,” Rust-lang.org, 2024. https://doc.rust-lang.org/edition-guide/editions/ (pristupljeno u oktobru 2024.)

[8] “The Unstable Book - The Rust Unstable Book,” Rust-lang.org, 2024. https://doc.rust-lang.org/unstable-book/index.html (pristupljeno u oktobru 2024.)

[9] Li, Chenghao, et al. “Demystifying Compiler Unstable Feature Usage and Impacts in the Rust Ecosystem.” 26 Oct. 2023. arXiv, (pristupljeno u oktobru 2024.)

[10] Bugden W, Alahmar A. Rust: The programming language for safety and performance. arXiv preprint arXiv:2206.05503. 2022 Jun 11.

[11] P. Mainardi, “The Rising Threat of Software Supply Chain Attacks: Managing Dependencies of Open Source projects,” Linuxfoundation.eu, Aug. 15, 2023. https://linuxfoundation.eu/newsroom/the-rising-threat-of-software-supply-chain-attacks-managing-dependencies-of-open-source-p (pristupljeno u oktobru 2024.)

[12] “The Architecture of Open Source Applications (Volume 1)LLVM,” Aosabook.org, 2024. https://aosabook.org/en/v1/llvm.html (pristupljeno u novembru 2024.)

[13] “The LLVM Compiler Infrastructure Project,” Llvm.org, 2024. https://llvm.org/ (pristupljeno u novembru 2024.)

Kratka biografija:

Aleksa Bajat rođen je 2001. godine u Novom Sadu. Završio je prirodno-matematički smer na engleskom jeziku u gimnaziji ”Jovan Jovanović Zmaj” 2019. godine. Tokom sve četiri godine gimnazije uspešno je pohađao ”Centar za mlade talente” kompanije Schneider Electric. Godine 2019. upisao je Fakultet Tehničkih Nauka u Novom Sadu, gde je ispunio sve obaveze i položio sve ispite predviđene studijskim programom sa prosečnom ocenom 9.03.
kontakt: aleksabajat15@gmail.com