УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА У НОВОМ САДУ

Марко Ердељи

Транспилација из *Python*-а у *Rust*: Студија случаја

ДИПЛОМСКИ РАД
- Основне академске студије -

Нови Сад, 2024

Образац **Q2.НА.15-04** - Издање 1

Садржај

1 Увод

Транспилација (енгл. *transpilation*)[1] представља значајан корак у оптимизацији и унапређењу перформанси софтвера. У савременом софтверском инжењерству, употреба трансформација изворног кода у изворни (енгл. *source - to - source transformation*)[2] показала се као кључна за постигнуће ефикасније извршне верзије програма. Ова техника отвара пут ка дубљем разумевању структуре и перформанси софтвера, омогућавајући програмерима да директно утичу на трансформацију изворног кода према специфичним захтевима извршења [1].

Први транспајлери су развијени у касним 70 - им и раним 80 - им годинама прошлог века. Године 1978, Intel је предложио аутоматски преводилац кода за конверзију 8 - битних програма у њихове еквивалентне 16 - битне програме [2]. XLT86™ је предложен 1981. године као преводилац асемблерског језика са 8080 на 8086, са циљем да аутоматски трансформише ASM тип фајлове у A86 тип фајлове [3].

Овај рад фокусира се на транспилацију између два програмска језика високог нивоа (енгл. *high - level programming languages*)[3] , *Python* - а и *Rust* - а, истражујући процесе, изазове и резултате овог инжењерског задатка. Кроз детаљну анализу алата и техника које су коришћене, циљ је да се истражи како транспилација може да унапреди перформансе и одрживост софтверских решења у контексту машинског учења.

У уводном делу рада биће детаљно истражена техника транспилације, са посебним фокусом на процес транспилације *Python* - а у *Rust*. Циљ истраживања је дубље разумевање техника транспилације, као кључне за оптимизацију и унапређење перформанси софтвера. Биће анализирани аспекти *Python* - а и *Rust* - а, са фокусом на њихове предности и ограничења у контексту перформанси, безбедности и скалабилности.

Даље, истраживаће се постојећи транспајлери (енгл. *transpilers*)[4], који се користе за *Python* и *Rust* транспилацију, са детаљном анализом алата и техника. Фокус ће бити на документацији процеса транспилације, укључујући кораке за постављање и коришћење транспајлера, као и приказивање примера кода пре и после транспилације. Посебна пажња биће посвећена студији случаја кроз примере мањих пројеката у области машинског учења, што ће омогућити дубље разумевање изазова, решења и потенцијалних унапређења.

На крају, у дискусији ће се сумирати главни налази истраживања, дискутовати предности и мане транспилације, као и дати предлози за будућа истраживања која би могла унапредити ефикасност и примену транспајлера у практичним софтверским решењима.

[1]Процес превођења између програмских језика високог нивоа.

[2]Техника која укључује конверзију између различитих језика високог нивоа без међупроцеса са машинским кодом.

[3]Програмски језици високог нивоа су програмски језици који су дизајнирани да буду разумљиви људима и омогућавају апстракцију сложених операција.

[4]Алати за транспилацију кода из једног програмског језика у други.

2 Теоријска Позадина

2.1 Дефиниција и значај транспајлера

Хирцел и коаутори [4] дефинишу процес транспилације као ону у којој се софтвер пише на изворном језику, али се компајлира и извршава у различитом програмском језику. Чабер и коаутори [5] објашњавају да је транспилација метод генерисања кода у коме дође до превода из једног програмског језика високог нивоа у други програмски језик ниског нивоа. Разлози зашто је транспилација значајна:

- **Миграција (енгл. *Migration)***[5] : Омогућава пренос legacy кода на модерније програмске језике, чиме се унапређује одржавање и распространање апликација [6].
- **Компатибилност**: Генерисање кода који је компатибилан са старијим верзијама програмских језика, док развојници користе нове функционалности доступне у модернијим верзијама [7].
- **Преусмеравање вештина програмирања**: Омогућава адаптацију програмера или тимских преференција транспилацијом кода у језик који је погоднији за разумевање или вештине тима [8].
- **Побољшање перформанси**: Унапређење перформанси апликација транспилацијом делова кода у језик који има бољи компајлер или је боље оптимизован за циљну платформу [6].

2.2 Структура и фазе транспилације

Архитектура транспајлера може се поделити на два дела – фронтенд (енгл. *frontend*) и бекенд (енгл. *backend*). Фронтенд преводи изворни језик у интермедијалну репрезентацију. Бекенд ради са интерном репрезентацијом да би произвео код на излазном језику [9].

2.2.1 Фронтенд транспајлера

Фронтенд транспајлера укључује неколико фаза које се секвенцијално извршавају. Структура је приказана на слици 1 [10].

- **Лексичка анализа (енгл. *lexical analysis*)**: Анализира ниску карактера која је представљена и дели је на токене који су легални чланови вокабулара језика у којем је програм написан.
- **Синтаксна анализа (парсирање) (енгл. *syntactic analysis/parsing*)**: Процесира секвенцу токена и производи међурепрезентацију, као што су стабло парсирања или секвенцијални међурепрезентацијски код.

[5] Пренос старог кода на новије програмске језике.

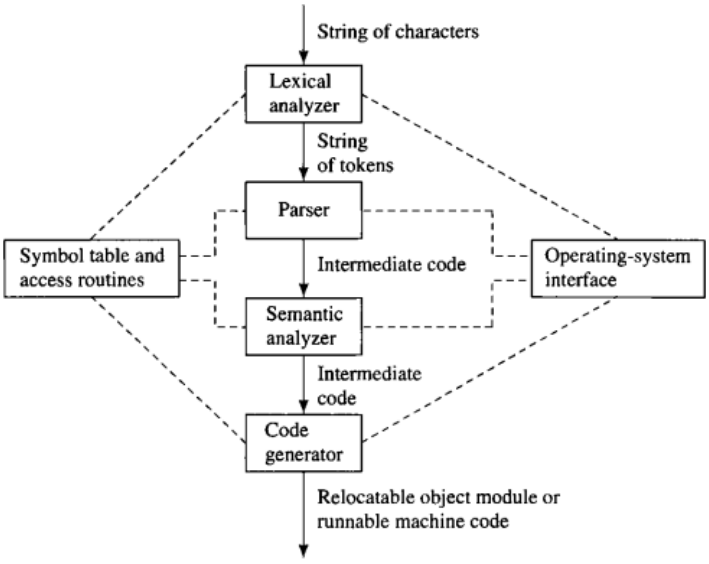
- **Семантичка провера (енгл. *semantic checking*)**: Врши проверу програма за статичку - семантичку исправност, утврђујући да ли програм задовољава статичке - семантичке особине захтеване од изворног језика.

- **Генерисање кода (енгл. *code generation*)**: Трансформише међурепрезентацију у еквивалентни код у циљном језику или другом облику прилагодљивом за циљну платформу.

Транспајлери могу бити имплементирани као једнопролазни или вишепролазни системи, слично као и класични компајлери. Једнопролазни транспајлери пружају бржу компилацију али може бити теже постићи високу ефикасност генерисаног кода. Вишепролазни приступи омогућавају бољу оптимизацију и квалитетнији генерисани код, али уз трошак дужег времена компилације.

Као и код компајлера, након што се транспилација заврши, програми или њихови делови обично морају бити повезани (линковани) како би се међусобно повезали и са потребним библиотечким рутинама, те учитани и премештени у меморију ради извршавања.

За многе високонивне језике, четири фазе могу бити комбиноване у један пролаз кроз изворни програм како би се произвео брз једнопролазни компајлер (енгл. *compiler*) [6]. Такав компајлер може бити сасвим одговарајућ за повремене кориснике, где је циљ обезбедити брз циклус измена, компилације и дебаговања. Међутим, такав компајлер генерално није у могућности да произведе веома ефикасан код. Алтернативно, фазе лексичке и синтаксне анализе могу бити комбиноване у пролаз који производи табелу симбола и неки облик међурепрезентације.



Слика 1: Структура *frontend* - а традиционалног транспајлера

[6]Компајлер је програм који може да прочита програм написан на једном језику и преведе га у еквивалентни програм на другом језику. За разлику од транспајлера, који преводи програм са једног језика на други у истом нивоу апстракције [11].

2.2.2 Бекенд транспајлера

Бекенд транспајлера се бави оптимизацијом кода. Ево како неке од оптимизационих техника које се користе у бекенду транспајлера могу да се примене [10]:

- **Замена скаларних референци на низове:** Транспајлери могу трансформисати приступе низовима у скаларним променљивама[7] где је то могуће, како би се смањило оптерећење приступа меморији и побољшала перформанса у циљном језику.
- **Интеграција процедура:** Спајање више процедура или функција у јединицу може смањити *overhead* позива функција[8], посебно у језицима где се семантика позива функција значајно разликује.
- **Оптимизација репних позива, укључујући елиминацију репне рекурзије:** Оптимизација рекурзивних позива функција ради избегавања прекорачења стека и побољшања перформанси је од суштинског значаја приликом транспилације, поготово приликом конверзије језика као што су Python у Rust.
- **Замена скаларних агрегата:** Слично замени скаларних референци на низове, ова оптимизација се фокусира на оптимизацију приступа агрегатним структурама података[9] (као што су структуре или класе), разбијајући их на појединачне компоненте ради побољшања перформанси у циљном језику.
- **Ретка константна пропагација услова:** Пропагација константи кроз условне изразе ради поједностављења израза и потенцијалног елиминације непотребних грана може унапредити ефикасност генерисаног кода током транспилације.
- **Интерпроцедурална пропагација константи:** Проширивање пропагације константи преко граница функција помаже у инлинирању и оптимизацији кода преко граница модула[10], унапређујући перформансе у циљном језику.
- **Специјализација и клонирање процедура:** Прилагођавање функција или процедура на основу њихових контекста употребе ради оптимизације специфичних случајева који се јављају током транспилације, чиме се побољшава укупна ефикасност кода.

[7] Скаларна променљива је појединачна променљива која чува једну вредност, за разлику од низова који могу садржати више елемената.

[8] *Overhead* позива функција је додатни ресурс (меморија и процесорско време) потребан за позивање функција.

[9] Агрегатне структуре података су комплексни подаци који садрже више различитих елемената или поља.

[10] Модул је организациона јединица у софтверу која садржи повезане функције, податке или друге ресурсе. Проширивање пропагације константи преко граница модула значи да се константе могу пропагирати и оптимизовати преко функција које припадају различитим модулима, што може побољшати перформансе и ефикасност генерисаног кода током транспилације.

- **Елиминација мртвог кода:** Уклањање недоступних или непотребних сегмената кода током транспилације осигурава да генерисани код буде компактан и ефикасан у циљном језику.

Ове технике се прилагођавају карактеристикама и ограничењима како изворног, тако и циљног језика. Док неке ниске оптимизације као што су машински идиоми или планирање инструкција могу бити мање применљиве због високог нивоа транспилације, технике које се фокусирају на семантичко очување и унапређење перформанси су кључне за ефикасно функционисање транспајлера.

2.3 *Python*

Python је опште - наменски програмски језик који постоји већ дуго: Гвидо ван Росум, креатор *Python* - а, почео је да развија овај језик још 1990. године. Језик је стабилан и веома високог нивоа, динамичан (енгл. *dynamic*)[11] , објектно - оријентисан (енгл. *object - oriented*)[12] и портабилан. *Python* ради на свим главним хардверским платформама и оперативним системима [12].

Python нуди високу продуктивност у свим фазама животног циклуса софтвера: анализа, дизајн, прототиповање, кодирање, тестирање, отклањање грешака, подешавање, документација, имплементација и одржавање. Популарност *Python* - а је константно расла током година. Данас је познавање *Python* - а предност за сваког програмера, јер је *Python* присутан у свим нишама и има корисне улоге у било ком софтверском решењу.

Python пружа мешавину елеганције, једноставности, практичности и моћи. Продуктивност кодирања са *Python* - ом расте захваљујући његовој конзистентности и регуларности, богатој стандардној библиотеци и алатима који су лако доступни за њега. *Python* је лак за учење, па је погодан за оне који су нови у програмирању, а истовремено је довољно моћан за најсофистицираније стручњаке.

Python је језик веома високог нивоа. То значи да *Python* користи виши ниво апстракције, концептуално даљи од основне машине, него класични компајлирани језици као што су *C*, *C++* и *Fortran*, који се традиционално називају језици високог нивоа. *Python* је једноставнији, бржи за обраду (и за људске мозгове и за програмске алате) и регуларнији од класичних језика високог нивоа. Ово омогућава високу продуктивност програмера и чини *Python* атрактивним алатом за развој. Добри компилатори за класичне компилиране језике могу генерисати бинарни машински код који ради брже од *Python* кода [12].

2.3.1 Зашто користити *Python*?

Постоји неколико разлога зашто користити Python [13]:

[11] Језик који подржава динамичко типизирање, што значи да се типови података могу одредити и изменити током извршавања програма.

[12] Објектно - оријентисан програмски језик омогућава организацију кода у објекте, који су инстанце класа. Ови објекти могу садржати податке и функције.

- **Портабилност:** *Python* ради на готово свим оперативним системима, укључујући *Linux/Unix*, *Windows*, *Mac*, *OS2* и друге.
- **Интеграција:** *Python* се може интегрисати са *COM*, *.NET* и *CORBA* објектима. Постоји *Jython* имплементација која омогућава коришћење *Python* - а на било којој *Java* платформи. *IronPython* је имплементација која даје *Python* програмерима приступ *.NET* библиотекама. *Python* такође може садржати обавијен *C* или *C++* код.
- **Лакоћа коришћења:** Врло је лако брзо се упознати са *Python* - ом и почети писати програме. Јасна и читљива синтакса чини апликације једноставним за креирање и отклањање грешака.
- **Моћ:** Стално се развијају нови проширења за *Python*, као што су приступ базама података, аудио/видео уређивање, графички кориснички интерфејс, веб развој и тако даље.
- **Динамичност:** *Python* је један од најфлексибилнијих језика. Лако је бити креативан са кодом и решавати дизајнерске и развојне проблеме.
- **Отвореност кода (енгл. *open source*)**[13] **:** *Python* је језик отвореног кода, што значи да се може слободно користити и дистрибуирати.

2.3.2 Мане *Python* - а

Python има неколико недостатака [14]:

- **Увећана потрошња меморије:** Типови података у *Python* - у троше значајно више меморије у поређењу са сличним типовима у другим језицима као што је *C*. На пример, цео број у *Python* - у троши 28 бајтова, док у *C* - у троши само 4 бајта. Ово је углавном због метаподатака које *Python* одржава за сваки објекат, укључујући бројаче референци и динамичке информације о типовима.
- **Одлагање сакупљања отпада (енгл. *garbage collection*[14]):** *Python* користи сакупљача отпада, што може одложити ослобађање меморије и повећати потрошњу меморије у поређењу са језицима који користе другачије управљање меморијом.
- **Прекомерна потрошња ресурса:** *Python* - ова висока потрошња меморије и перформанси могу бити проблематични, нарочито у критичним деловима кода који су захтевни за меморију и перформансе.

[13] Односи се на софтвер чији изворни код је доступан јавности за коришћење, измену и дистрибуцију. Корисници могу слободно да прегледају, модификују и деле софтвер, што промовише транспарентност и сарадњу.

[14] Метод управљања меморијом који аутоматски ослобађа меморију коју више није могуће користити.

- **Недостатак детаљних информација:** Постојећи алати за профилисање (енгл. *profiling*[15]) *Python* - а можда не пружају детаљне информације о перформансама, јер не решавају специфичне изазове окружења извршења *Python* - а.

- **Оптимизација преко нативних библиотеках (енгл. *native libraries*[16]):** Због трошкова перформанси и меморије у *Python* - у, често је неопходно користити пакете високих перформанси са нативним имплементацијама (нпр. *NumPy*, *SciKit-Learn*, *TensorFlow*) за оптимизацију кода. Ово значи да чист *Python* код можда није довољан за апликације које су критичне за перформансе.

2.4 *Rust*

Rust је модерни системски програмски језик који је развила Mozilla Research, а његов главни креатор је Грејдон Хоар. Развој *Rust* - а је почео 2010. године, са циљем да обезбеди високу безбедност и перформансе у системском програмирању. *Rust* је постао званично стабилан са издавањем верзије 1.0 у мају 2015. године [15].

Rust је познат по својој јединственој комбинацији брзине и безбедности. Он подржава парадигму власништва (енгл. *ownership*)[17], што омогућава програмерима да пишу сигуран и високо ефикасан код. Језик је истовремено статички типизирани (енгл. *statically-typed*)[18] и подржава више парадигми програмирања, укључујући објектно - оријентисано и функционално програмирање [16].

2.4.1 Зашто користити *Rust*?

Постоји неколико разлога зашто користити Rust [17]:

- **Безбедност:** *Rust* је дизајниран са фокусом на сигурност меморије. Његов систем власништва спречава уобичајене грешке као што су употреба после ослобађања меморије и конкурентне трке података.

- **Перформансе:** *Rust* пружа перформансе на нивоу са *C* и *C++* захваљујући концепту апстракције са нултим трошковима, што значи да можете писати висококвалитетни код без компромиса у брзини.

- **Екосистем:** *Rust* има богат екосистем алата и библиотека, укључујући *Cargo*, који управља пројектима, грађењем, тестирањем и дистрибуцијом пакета. Уз *Cargo*, *Rust* заједница развија и подржава велики број корисних библиотека за различите домене.

[15] Процес праћења и анализе перформанси програма.

[16] Библиотеке написане у другим језицима као што су C или C++ које могу побољшати перформансе.

[17] Власништво је концепт који обезбеђује безбедно управљање меморијом без потребе за аутоматским сакупљањем отпада, што елиминише читаву класу грешака везаних за меморију као што су сегментационе грешке и цурење меморије.

[18] Језик у којем се типови свих израза одређују у време компајлирања.

- **Конкурентност (енгл. Concurrency)**[19]**:** *Rust* омогућава писање безбедног конкурентног кода кроз свој систем за позајмљивање и правила за дељење података.

- **Ефикасност:** *Rust* избегава потребу за сакупљањем смећа захваљујући статичкој анализи времена живота објеката, што побољшава ефикасност рада програма.

- **Популарност:** *Rust* је стално на врху листе најомиљенијих програмских језика према Stack Overflow истраживању, што указује на задовољство програмера његовом употребом.

2.4.2 Мане *Rust* - а

Rust има неколико недостатака [19]:

- **Недостатак зрелости:** *Rust* је релативно млад језик. Ово значи да језик можда није довољно тестиран у стварним сценаријима и да још увек пролази кроз значајне промене.

- **Мања потражња за програмерима:** За разлику од *C* и *C++*, за које је лакше пронаћи и запослити програмере, постоји мања потражња за *Rust* програмерима на тржишту, што може отежати проналажење талентованих инжењера.

- **Мањи број алата и библиотека:** Иако *Rust* има значајан екосистем алата и библиотека, још увек није на нивоу старијих језика као што су *C* и *Python*. Ово може ограничити брзину развоја и доступност решења за специфичне проблеме.

- **Комплексност система власништва и позајмљивања:** Систем власништва и позајмљивања (енгл. *borrowing*)[20] у *Rust* - у, иако обезбеђује сигурност, може бити сложен и захтевати веће време учења за нове програмере.

- **Недостатак ручног управљања меморијом:** Иако *Rust* избегава многе грешке које се јављају због управљања меморијом, у ретким случајевима када је ручно управљање меморијом потребно, програмери се морају ослонити на небезбедне блокове (енгл. *unsafe blocks*)[21], што може бити изазовно и ризично.

[19]Конкурентност се односи на способност извршавања више задатака или операција истовремено унутар истог програма или система [18].

[20]Позајмљивање је концепт у *Rust* - у који омогућава променљивим да позајмљују референце на податке без промене власништва над тим подацима.

[21]Небезбедни блокови у *Rust* - у омогућавају програмерима да деактивирају неке од безбедносних провера *Rust* - а, као што су провере позајмљивања. То омогућава директно управљање меморијом, позивање небезбедних функција или метода, приступ и измену глобалних променљивих, имплементацију небезбедних особина и приступ пољима у унијама.

3 Преглед литературе

4 Литература

[1] D. B. Loveman, “Program Improvement by Source - to - Source Transformation,” J. ACM, vol. 24, no. 1, pp. 121–145, Jan. 1977, doi: 10.1145/321992.322000.

[2] Intel. MCS - 86 Assembly Language Converter Operating Instructions for ISIS - II Users. Technical Report. 1978. [Online]. Available: http://www.bitsavers.org/pdf/intel/ISIS_II/9800642A_MCS-86_Assembly_Language_Converter_Operating_Instructions_for_ISIS-II_Users_Mar79.pdf (приступљено 18. јула 2024).

[3] Research, D. XLT86 8080 to 8086 Assembly Language Translator, User’s Guide. Technical Report. 1981. [Online]. Available: http://www.s100computers.com/Software%20Folder/Assembler%20Collection/Digital%20Research%20XLT86%20Manual.pdf (приступљено 18. јула 2024).

[4] M. Hirzel and H. Klaeren, “Code coverage for any kind of test in any kind of transcompiled cross - platform applications,” in Proceedings of the 2nd International Workshop on User Interface Test Automation, in INTUITEST 2016. New York, NY, USA: Association for Computing Machinery, 2016, pp. 1–10. doi: 10.1145/2945404.2945405.

[5] P. Chaber and M. Ławryńczuk, ”Effectiveness of PID and DMC control algorithms automatic code generation for microcontrollers: Application to a thermal process,” 2016 3rd Conference on Control and Fault - Tolerant Systems (SysTol), Barcelona, Spain, 2016, pp. 618 - 623, doi: 10.1109/SYSTOL.2016.7739817.

[6] M. Bysiek, A. Drozd, and S. Matsuoka, “Migrating Legacy Fortran to Python While Retaining Fortran - Level Performance through Transpilation and Type Hints,” 2016 6th Workshop on Python for High - Performance and Scientific Computing (PyHPC), pp. 9–18, 2016.

[7] A. Pilsch, “Translating the Future: Transpilers and the New Temporalities of Programming in JavaScript,” 2018.

[8] P. Misse - Chanabier, V. Aranega, G. Polito, and S. Ducasse, “Illicium A modular transpilation toolchain from Pharo to C,” in IWST19 - International Workshop on Smalltalk Technologies, Köln, Germany, Aug. 2019. [Online]. Available: https://hal.science/hal-02297860 (приступљено 19. јула 2024).

[9] E. Ilyushin and D. Namiot, “On source - to - source compilers,” International Journal of Open Information Technologies, vol. 4, Apr. 2016.

[10] S. Muchnick, Advanced compiler design implementation. Morgan kaufmann, 1997.

[11] V. A. Alfred, S. L. Monica, and D. U. Jeffrey, Compilers principles, techniques & tools. pearson Education, 2007.

[12] A. Martelli, A. M. Ravenscroft, S. Holden, and P. McGuire, Python in a Nutshell. O'Reilly Media, Inc., 2023.

[13] B. Dayley, Python phrasebook: essential code and commands. Sams Pub., 2007.

[14] E. D. Berger, S. Stern, and J. A. Pizzorno, "Triangulating python performance issues with SCALENE," in 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23), 2023, pp. 51–64.

[15] S. Klabnik, "The History of Rust," Association for Computing Machinery (ACM), on YouTube, June, vol. 22, 2016.

[16] S. Klabnik and C. Nichols, The Rust programming language. No Starch Press, 2023.

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

[18] M. J. Sottile, T. G. Mattson, and C. E. Rasmussen, Introduction to concurrency in programming languages. CRC Press, 2009.

[19] K. R. Fulton, A. Chan, D. Votipka, M. Hicks, and M. L. Mazurek, "Benefits and drawbacks of adopting a secure programming language: Rust as a case study," in Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021), 2021, pp. 597–616.