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.

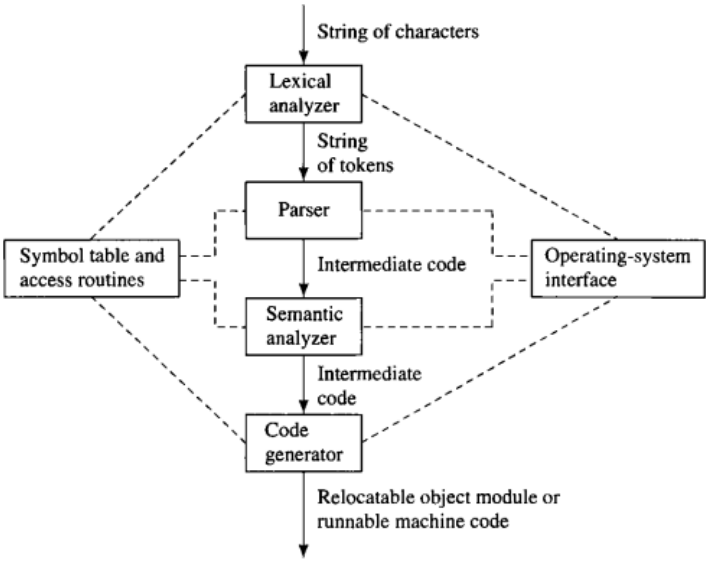
- **Семантичка провера (енгл. *semantic checking*)**: Врши проверу програма за статичку-семантичку исправност, утврђујући да ли програм задовољава статичке-семантичке особине захтеване од изворног језика.

- **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.

3 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.