VILNIAUS UNIVERSITETAS

Audrius
ŠAIKŪNAS

Earley virtualiųjų mašinų panaudojimas plečiamai programavimo kalbų sintaksinei analizei

DAKTARO DISERTACIJOS SANTRAUKA

Technologijos mokslai,
Informatikos inžinerija T 007

VILNIUS 2019

Disertacija rengta 2015–2019 metais Vilniaus universitete.

Mokslinis vadovas:
prof. dr. Albertas Čaplinskas (Vilniaus universitetas, technologijos mokslai, informatikos inžinerija – T 007).

Gynimo taryba:

Pirmininkas – prof. dr. Julius Žilinskas (Vilniaus universitetas, technologijos mokslai, informatikos inžinerija – T 007).
Nariai:
prof. dr. Bostjan Brumen (Mariboro universitetas, Slovėnija, technologijos mokslai, informatikos inžinerija – T 007),
prof. habil. dr. Antanas Čenys (Vilniaus Gedimino technikos universitetas, technologijos mokslai, informatikos inžinerija – T 007),
prof. dr. Vacius Jusas (Kauno technologijos universitetas, technologijos mokslai, informatikos inžinerija – T 007),
prof. dr. Rimantas Vaicekauskas (Vilniaus universitetas, technologijos mokslai, informatikos inžinerija – T 007).

Disertacija ginama viešame Gynimo tarybos posėdyje 2019 m. gruodžio mėn. 19 d. 14 val. Vilniaus universiteto Duomenų mokslo ir skaitmeninių technologijų instituto 203 auditorijoje.
Adresas: Akademijos g. 4, LT-04812, Vilnius, Lietuva,
Disertacijos santrauka išsiuntinėta 2019 m. lapkričio mėn. 19 d.

Disertaciją galima peržiūrėti Vilniaus univeristeto bibliotekoje ir VU interneto svetainėje adresu:
`https://www.vu.lt/naujienos/ivykiu-kalendorius`

VILNIUS UNIVERSITY

Audrius
ŠAIKŪNAS

Extensible parsing with Earley virtual machines

SUMMARY OF DOCTORAL DISSERTATION

Technological Sciences,
Informatics Engineering T 007

VILNIUS 2019

The dissertation has been prepared at Vilnius University from 2015 to 2019.

Scientific Supervisor:
Prof. Dr. Albertas Čaplinskas (Vilnius University, Technological Sciences, Informatics Engineering – T 007).

This doctoral dissertation will be defended in a public meeting of the Dissertation Defence Panel:

Chairman – Prof. Dr. Julius Žilinskas (Vilnius University, Technological Sciences, Informatics Engineering – T 007).
Members:
Prof. Dr. Bostjan Brumen (University of Maribor, Slovenia, Technological Sciences, Informatics Engineering – T 007),
Prof. Habil. Dr. Antanas Čenys (Vilnius Gediminas Technical University, Technological Sciences, Informatics Engineering – T 007),
Prof. Dr. Vacius Jusas (Kaunas University of Technology, Technological Sciences, Informatics Engineering – T 007),
Prof. Dr. Rimantas Vaicekauskas (Vilnius University, Technological Sciences, Informatics Engineering – T 007).

The dissertation shall be defended at a public meeting of the Dissertation Defence Panel at 2 p.m. on 19th of December, 2019 in meeting room 203 of the Institute of Data Science and Digital Technologies of Vilnius University.
Address: Akademijos st. 4, LT-04812, Vilnius, Lithuania.
The summary of the doctoral dissertacion was distributed on the 19th of November, 2019.

The text of this dissertation can be accessed at the libraries of Vilnius university, as well as on the website of Vilnius University:
`https://www.vu.lt/naujienos/ivykiu-kalendorius`

TURINYS

1. ĮVADAS

1.1. Tyrimo aktualumas ir motyvacija

Programavimo kalbos ir kompiliatoriai – tai viena iš senesnių informatikos temų. Kaip ir daugumoje kitų informatikos temų, programavimo kalbose ir kompiliatoriuose yra nuolat atrandama naujų inovacijų. Verta nepamiršti, kad net ir struktūrinis programavimas ar funkcijos vienu metu buvo naujovė. Tačiau naujų savybių įgyvendinimas (ypač egzistuojančiose programavimo kalbose) nėra paprastas procesas.

Per paskutinius du dešimtmečius C++ – viena iš populiariausių programavimo kalbų – sulaukė net 4-ių naujų standartų: C++98, C++11, C++14 ir C++17 (neskaitant būsimo C++20). Daugybė žmonių iš viso pasaulio prisidėjo prie šių naujų C++ versijų kūrimo. Kiekvienas kalbos keitimo siūlymas buvo detaliai aptariamas įvairiuose forumuose ir konferencijose, griežtai dokumentuojamas iš anksto nustatytu formatu, o paskui galutiniai keitimai buvo priimami arba atmetami C++ standartizacijos komitete, sudarytame iš technologijų industrijos veteranų – tokių įmonių, kaip Apple, Google ir Microsoft. C++ keitimo ir tobulinimo procesas yra lėtas ir sudėtingas. Dėl šios priežasties daugelis potencialių C++ naudotojų, užuot prisidėję prie C++ tobulinimo, nusprendžia, kad kurti naują programavimo kalbą, kuri geriau patenkintų savo poreikius, yra greičiau ir paprasčiau.

Tai viena iš priežasčių, kodėl sukuriama daugybė naujų programavimo kalbų: atsiranda naujų dalykinės srities problemų, kurių aprašymas ir sprendimas su egzistuojančiomis programavimo kalbomis yra per sudėtingas. Dažnai tokioms problemoms spręsti atsiranda mažesnių, bet labiau specializuotų kompiuterių kalbų.

Net ir vidutinio dydžio projektuose, ypač susijusiuose su interneto technologijomis, dažnai pasitaiko situacijų, kad viename projekte naudojamos penkios ar daugiau kompiuterio kalbų, pvz.: HTML – interneto puslapio struktūrai, CSS – puslapio išvaizdai, JavaScript – puslapio elgesiui, SQL – duomenims iš duomenų bazės pasiekti ir Ruby/Python – puslapiams generuoti. Tokio pobūdžio projektuose dažnai tenka viena programavimo kalba generuoti kitos kompiuterių kalbos pradinį tekstą. Tačiau generuojant kitos programavimo kalbos pradinį tekstą kaip teksto eilutes yra ypač paprasta padaryti klaidų, kurias nėra lengva aptikti ir diagnozuoti. Dėl šios priežasties skirtingų (programavimo) kalbų integracijos klausimas tampa vis aktualesnis.

Tačiau kas būtų, jei trūkstamą funkcionalumą į egzistuojančią programavimo kalbą būtų galima *tiesiog įdėti* netaisant tos programavimo kalbos kompiliatoriaus kodo? Kas nutiktų, jei vieną programavimo kalbą be pastangų būtų galima integruoti į kitą? Dėl šių ir panašių klausimų būtent ir egzistuoja *plečiamosios programavimo kalbos*. Šios kalbos, priklausomai nuo suteikiamų plečiamumo galimybių, leidžia plėsti savo sintaksę ir semantiką nekeičiant savo kompiliatorių.

Deja, tokių kalbų nėra daug. Viena iš priežasčių – nėra tinkamų tokioms kalboms analizuoti sintaksinės analizės metodų. Dėl to šiame darbe yra pristatomas naujas sintaksinės analizės metodas (Earley virtualioji mašina su integruota leksine analize (SEVM), pritaikytas plečiamųjų programavimo kalbų analizei. Šis metodas tinka sintaksinei analizei be atskiro leksinio analizatoriaus atlikti, palaiko dinamiškai kintančias gramatikas, netaiko papildomų apribojimų bekontekstėms gramatikoms ir suteikia mechanizmų, kurie reikalingi programavimo kalboms ar jų plėtiniams apibrėžti taikant gramatikų kompoziciją.

1.2. Problemos formulavimas

Problema: reflektyviai plečiamų programavimo kalbų sintaksinė analizė, kurios pagrindas – virtualiosios mašinos, (1) apdorojančios analizės metu dinamiškai kintančias bekontekstes gramatikas su lokaliais plėtiniais, dekomponuojamas į smulkesnes gramatikas, ir (2) integruojančios leksinę analizę į bendrą sintaksinės analizės procesą (t. y. neatskiriančios leksinės analizės nuo sintaksinės analizės ir tuo pačiu pašalinančios būtinybę turėti savarankišką leksinį analizatorių).

Reflektyviai plečiamos programavimo (RPP) kalbos – tai tokios programavimo kalbos, kurių sintaksė ir semantika gali būti praplėsta dinamiškai nekeičiant transliatoriaus kodo (žr. 2.5.). Kadangi tokių kalbų gramatikos gali kisti dinamiškai analizės metu, jų analizei reikalingi specialūs sintaksinės analizės metodai, kurie turėtų šias savybes:

1. Dinamiškai kintančių gramatikų palaikymas.
2. Integruota leksinė analizė.
3. Apibendrinta (neapribota) bekontekstė analizė.
4. Lokalių gramatinių plėtinių palaikymas (žr. 3.1.).
5. Priimtinas našumas.

Įprastuose sintaksinės analizės metoduose gramatikos yra koduojamos naudojant perėjimų lenteles, kuriomis yra išreiškiama baigtinio automato su dėklu struktūra. Šios lentelės yra paskui naudojamos analizės procesui vykdyti. Tačiau tokios paprastos analizatoriaus struktūros nepakanka norint sukurti sintaksinės analizės metodą, kuris būtų tinkamas plečiamųjų kalbų analizei.

Kad sintaksinės analizės metodas palaikytų dinamiškai kintančias gramatikas vykdymo metu, turi būti numatyti gramatikų apibrėžimo kalbos elementai, kurie leistų manipuliuoti aktyvią (konkrečiam fragmentui analizuoti naudojamą) gramatiką. Taip pat, kad

būtų galima gramatikas apibrėžti taikant smulkesnių gramatikų kompoziciją, sintaksinės analizės metodas turėtų veikti be atskiro leksinio analizatoriaus. Tiksliau, leksinė analizė turėtų būti integruota į bendrą sintaksinės analizės procesą. Tai apsunkina analizės procesą, nes leksiniame lygmenyje kylančios dviprasmybės turi būti sprendžiamos sintaksiniame analizatoriuje. Kitaip tariant, tiek gramatikų apibrėžimo kalboje, tiek vidinėje analizatoriaus gramatikų formoje turi būti numatyta elementų, leidžiančių (pagal poreikį) spręsti iškilusias dviprasmybes.

Dėl šių priežasčių reikalinga raiškesnė analizatoriaus vidinė gramatikų forma: instrukcijų sekos, kurios gali būti vykdomos virtualiojoje mašinoje.

1.3. Tyrimo tikslas ir uždaviniai

Tyrimo objektas: plečiamųjų programavimo kalbų sintaksinė analizė.

Tyrimo tikslas: sukurti sintaksinės analizės metodą, tinkamą reflektyviai plečiamoms programavimo (RPP) kalboms analizuoti.

Tyrimo uždaviniai:

1. Gramatikų aprašymo kalbos sudarymas.
2. Virtualiosios mašinos, kuri būtų tinkama apibendrintai bekontekščių kalbų sintaksinei analizei su lokaliais gramatiniais plėtiniais vykdyti, konstravimas.
3. Bendros metodo struktūros konstravimas.
4. Leksinės analizės integravimas į sintaksinę analizę.
5. Principinio metodo įgyvendinamumo demonstravimas (sukuriant maketą).
6. Metodo našumo įvertinimas.

Norint įgyvendinti aprašytą sintaksinės analizės metodą, pirmiausia reikia apibrėžti gramatikų apibrėžimo kalbą ir instrukcijų kalbą, į kurią tokios gramatikos būtų transliuojamos. Be to, instrukcijų kalba turi būti pakankamai raiški, kad joje išeitų pavaizduoti ne tik visas bekontekstes gramatikas, bet ir spręsti dėl integruotos leksinės analizės kylančias dviprasmybes. Galiausiai, turi būti apibrėžta ir įgyvendinta virtualioji mašina, kuri gebėtų tokias instrukcijas vykdyti priimtinu našumu, kad šis analizės metodas būtų tinkamas naudoti praktikoje.

1.4. Ginamieji teiginiai

1. Nė vienas iš tirtų sintaksės analizės metodų nėra tinkamas reflektyviai plečiamoms programavimo (RPP) kalboms analizuoti.
2. Earley sintaksinis analizatorius gali būti praplėstas taip, kad tenkintų visus funkcinius RPP kalbų analizatoriaus reikalavimus.
3. Earley virtualiosios mašinos su integruota sintaksine analize yra pakankamai našios naudoti praktikoje.

1.5. Tyrimo metodai

Pirmasis ginamasis teiginys yra įrodomas atliekant kritinę literatūros apžvalgą ir taikant koncepcinės analizės metodus. Prieš imantis kritinės apžvalgos, nustatomi jos kriterijai. Tiksliau tariant, suformuluojami RPP kalbų analizatorių reikalavimai. Paskui apžvelgiami visi analizės metodai, kurie tenkina bent dalį apibrėžtų kriterijų. Kadangi literatūros apžvalgos metu nėra randamas tinkamas sintaksinės analizės metodas RPP kalbų analizei, parenkama viena iš apžvelgtų analizatorių šeimų (Earley), kuri bus naudojama kaip pagrindas kitam tyrimo etapui vykdyti.

Antrajame tyrimo etape yra taikomas tyrimas konstravimu: remiantis pasirinkta analizatorių šeima sukonstruojami du sintaksinės analizės metodai. Earley virtualiosios mašinos (EVM) – tai naujas sintaksinės analizės algoritmas, kuris tenkina pirmus keturis RPP kalbų analizatorių reikalavimus. Sukonstravus EVM nustatomi šio analizės metodo trūkumai, kurie pašalinami konstruojant su integruota leksine analize papildytą EVM versiją – SEVM.

Trečiajame etape yra vykdomas kontroliuojamas eksperimentas. Įgyvendinto SEVM maketo našumas lyginamas su kitais sintaksinės analizės metodais, siekiant pademonstruoti, kad nepaisant siūlomo lankstumo, SEVM gali veikti ir priimtinu našumu. Šiam lyginimui atlikti sukuriami du tyrimo įrankiai: `bench_parsers` ir `north_cli`.

1.6. Rezultatai

- Sukurtas sintaksinės analizės metodas (SEVM), kuris yra tinkamas RPP kalboms analizuoti;
- sukurta gramatikų apibrėžimo kalba, kurios raiškos pakanka ne tik praktikoje naudojamoms programavimo kalboms, bet ir RPP kalboms apibrėžti;
- sukurtas SEVM algoritmo įgyvendinimas (pavadintas `north`);
- sukurtas tyrimo įrankis `bench_parsers`, kuris yra tinkamas įvairių sintaksinių analizatorių našumui lyginti;
- sukurtas tyrimo įrankis `north_cli`, kuris leidžia analizuoti vidinę SEVM analizatoriaus būseną.

1.7. Mokslinis naujumas

- SEVM yra analizės metodas, kurio pagrindas – virtualiosios mašinos. Nors ir seniau sukurta sintaksinės analizės metodų, kurie naudojo virtualiąsias mašinas, tačiau tai pirmasis atvejis,

kai virtualiosios mašinos instrukcijos yra naudojamos išreikšti neapribotoms bekontekstėms gramatikoms. SEVM taip pat rodo, kaip stipriai virtualiųjų mašinų naudojimas gali praplėsti gramatikų išraiškingumą ir bendrą analizatoriaus funkcionalumą, ypač lyginant SEVM su analizės metodais, naudojančiais perėjimų lenteles gramatikoms koduoti.
- SEVM determinuotų poaibių iškėlimo metodas yra tinkamas ir kitiems analizės metodams (pvz., originaliam Earley ir GLR), norint padidinti jų analizės našumą ir suteikti paprastą ir efektyvų leksinių dviprasmybių šalinimo mechanizmą.
- SEVM gramatikų aprašymo kalba suteikia galimybę lanksčiai apibrėžti kompiuterių kalbas ir jų plėtinius taikant gramatikų kompoziciją.

1.8. Praktinė gautų rezultatų nauda

Tyrimo, kuris yra šios disertacijos pagrindas, metu sukurtas analizės metodas turi nemažai naudoti praktikoje tinkamų savybių:

- Bekontekstė analizė supaprastina naujų programavimo kalbų gramatikų rašymą, nes nėra specialių apribojimų, kurie būtų taikomi rašomoms gramatikoms (pvz., rekursyviai nusileidžiančiuose analizatoriuose negalima taikyti kairiosios rekursijos, dėl to visos gramatikos turi būti perrašytos net ir tuo atveju, kai kairioji rekursija yra naudinga iš dešinės asociatyviems operatoriams apibrėžti).
- Kadangi SEVM veikia be atskiro leksinio analizatoriaus, nebereikia naudoti dviejų skirtingų gramatikų ir gramatikų kalbų, norint apibrėžti vieną programavimo kalbą.
- SEVM gramatikos gali būti apibrėžiamos ir sudarytos iš skirtingų modulių. Taip bazinė kalba gali būti apibrėžiama

vienoje gramatikoje, o papildomi šios kalbos elementai – gramatiniuose plėtiniuose. Tai labai palengvina naujų gramatikų realizavimą ir leidžia egzistuojančias gramatikas pakartotinai naudoti, nes jose iš anksto numatyti gramatiniai plėtimo taškai.
- Dinamiškai kintančių gramatikų palaikymas leidžia realizuoti sudėtingesnes programavimo kalbas su lankstesniais plėtimo mechanizmais (pvz., su makrokomandomis, kurių sintaksė nėra apribota).
- Virtualiųjų mašinų naudojimas realizuojant SEVM ne tik leidžia įvairesniais būdais apibrėžti kalbų gramatikas, bet ir suteikia ypač plačių SEVM analizės metodo plėtimo galimybių: SEVM naudotojai gali papildyti SEVM naujomis instrukcijomis, kurios gali leisti analizuoti dar sudėtingesnes gramatikas ar specializuotais atvejais analizę atlikti efektyviau.

1.9. Aprobavimas

Pagrindiniai disertacijos rezultatai pristatyti šiose tarptautinėse konferencijose:

- FedCSIS 2017, 6th Workshop on Advances in Programming Languages (WAPL'17), Praha, Čekijos Respublika, 2017.09.03–2017.09.07.
- FCSIT 2019, European Conference on Frontiers of Computer Science and Information Technology, Amsterdamas, Nyderlandų Karalystė, 2019.09.22–2019.09.24.

1.10. Publikacijos

Disertacijos rezultatai paskelbti šiose mokslinėse publikacijose:

- Šaikūnas A. (2017). Critical Analysis of Extensible Parsing Tools and Techniques. Baltic J. Modern Computing, Vol. 5 (2017), No. 1, 136–145.
- Šaikūnas A. (2019). Parsing with Scannerless Earley Virtual Machines. Baltic J. Modern Computing, Vol. 7 (2019), No. 2, 171–189.

Kiti straipsniai:

- Šaikūnas A. (2017). Parsing with Earley Virtual Machines. Communication Papers of the 2017 Federated Conference on Computer Science and Information Systems, Vol. 13 (2017), 165–173.
- Šaikūnas A. (2019). Just-in-time Parsing with Scannerless Earley Virtual Machines (*priimta publikuoti į konferencijos darbų sąrašą*).

1.11. Darbo apžvalga

Ši disertacija sudaryta iš 7 skyrių:

1. Įvadas.

2. Įvadinės žinios. Pateikiamos esminės žinios apie kompiliatorius ir sintaksinę analizę.

3. Literatūros analizė. Aprašyta atlikta literatūros analizė, apžvelgiami analizės metodai, kurie būtų tinkami RPP kalboms analizuoti.

4. Earley virtualiųjų mašinų sintaksinės analizės metodo sudarymas. Pateikiamas šio darbo metu sukurtas analizės metodas – Earley virtualioji mašina, kuris yra tinkama RPP kalboms analizuoti.

5. Earley virtualiosios mašinos su integruota leksine analize. Pateikiama EVM metodo patobulinta versija (SEVM), našesnė ir

su papildomomis galimybėmis, kurios yra reikalingos šiam metodui naudoti praktikoje.

6. Eksperimentinis Earley virtualiųjų mašinų su integruota sintaksine analize vertinimas. SEVM metodo įgyvendinimas yra eksperimentiškai lyginamas su kitais praktikoje naudojamais sintaksiniais analizatoriais.

7. Bendrosios išvados. Pateikiamos bendrosios disertacijos išvados.

2. ĮVADINĖS ŽINIOS

Šiame skyriuje trumpai pristatomi esminiai kompiliatorių ir programavimo kalbų principai. Taip pat apibrėžiamas *reflektyviai plečiamos programavimo kalbos* terminas.

2.1. Kompiliatoriai ir programavimo kalbos

Programavimo kalbos skirstomos į tris istorines kartas:

1. Pirmosios kartos programavimo kalbų kodai buvo saugomi fizinėse duomenų laikmenose (pvz., perfokortose, magnetinėse juostose, mechaniniuose jungikliuose).
2. Kad programuotojams nereikėtų programų rašyti dvejetainėje formoje rankiniu būdu, sukurtos antrosios kartos programavimo kalbos. Šios programos buvo rašomos tekstinėje formoje ir buvo tiesiogiai transliuojamos į binarinės formos mašininį kodą transliatoriais (dažnai vadinamais asembleriais). Transliatoriai tiesiogiai vertė pateiktą programos tekstą į mašinines instrukcijas netaikydami jokių papildomų analizės ar optimizavimo žingsnių.
3. Šiuo metu dauguma iš naudojamų kalbų yra trečios kartos. Šios kalbos įprastai gana stipriai atitolsta nuo aparatūrinės įrangos, kuriose šios programos yra vykdomos. Taip pat jose yra įvedamos aukšto lygio abstrakcijos, kurios neturi tiesioginių aparatūrinės įrangos atitikmenų (pvz.: kintamieji, duomenų struktūros ir pan.). Ne visos trečiosios kartos kalbos yra klasifikuojamos kaip *programavimo kalbos*. Šios kalbos dažnai yra stipriai specializuotos ir skirtos spręsti konkrečioms dalykinės srities problemoms. Todėl šios kalbos yra vadinamos *nuo domeno (dalykinės srities) priklausomomis kalbomis*.

Daugumą programavimo kalbų galima suskirstyti pagal naudojamus transliatorius į dvi pagrindines kategorijas:

1. *Kompiliuojamose programavimo kalbose* pradiniai tekstai įprastai yra transliuojami (kompiliuojami) į mašininį kodą. Tačiau naujesnėse trečios kartos programavimo kalbose dažnai kodas yra kompiliuojamas į *dvejetainį kodą*, kuris programos veikimo metu yra interpretuojamas arba transliuojamas į mašininį kodą naudojant *virtualiąsias mašinas*.
2. *Interpretuojamose programavimo kalbose* kodas yra vykdomas tiesiogiai, neverčiamas į mašininį kodą. Programa, kuri atlieka kodo vykdymą, yra vadinama *interpretatoriumi*, o pats vykdymo procesas – *programos interpretavimu*.

2.2. Tipinė kompiliatorių architektūra

Kompiliatorius dažniausiai yra sudarytas iš šių pagrindinių komponentų:

- *Leksinis analizatorius*, kuris transformuoja pradinį tekstą į leksemų seką. *Leksema* – tai vienas programavimo kalbos žodis (pvz.: raktažodis, operatorius ar pan.). Šalia kiekvienos leksemos yra saugomas jos tipas, su leksema susieta reikšmė ir įvesties pozicija, kur ši leksema buvo rasta.
- *Sintaksinis analizatorius*, skirtas transformuoti leksemų sąrašą į abstraktų sintaksinį medį (ASM) – duomenų struktūrą, kuri pavaizduoja bendrą programos struktūrą.
- *Semantinis analizatorius*, kuris papildo abstraktų sintaksinį medį nauja gauta semantine informacija, kuri bus reikalinga tolimesniems kompiliavimo etapams. Šis komponentas taip pat patikrina, ar įvestyje nėra kalbos semantinių klaidų (pvz., ar nėra naudojami neapibrėžti kintamieji).

- *Kodo generatorius*, skirtas transformuoti ASM į žemesnio lygmens kodą, kuris per kelis etapus transliuojamas į dvejetainį ar mašininį kodą.

2.3. Leksinė ir sintaksinė analizė

Formaliosioms kalboms apibrėžti naudojamos formaliosios gramatikos. Kadangi jos tinkamesnės teorinei analizei, praktinėms kompiuterių (ir programavimo) kalboms apibrėžti naudojamos *gramatikų aprašymo kalbos* (pvz.: LEX, YACC).

Dauguma leksinių ir sintaksinių analizatorių naudoja duomenis (pvz., perėjimų lenteles), kurie yra gaunami iš gramatikų. Kad programuotojams nereikėtų rankiniu būdu šių duomenų skaičiuoti, naudojami leksinių ir sintaksinių analizatorių generatoriai, kurie iš gramatikų generuoja atitinkamo analizatoriaus pradinį tekstą.

Leksinės ir sintaksinės analizės žingsniai istoriškai yra atskirti tik dėl našumo: leksinei analizei atlikti galima taikyti gerokai paprastesnius algoritmus (naudojant determinuotus baigtinius automatus su tiesiniu asimptotiniu sudėtingumu). Nors dėl šio atskyrimo bendras našumas didėja, kyla kelios papildomos problemos: leksinei ir sintaksinei analizei naudojamos atskiros gramatikų kalbos; taip pat šių dviejų komponentų integravimas ne visuomet yra paprastas.

Padidėjus kompiuterių našumui pradėti naudoti *sintaksiniai analizatoriai su integruota leksine analize* – juose nėra naudojamas atskiras leksinis analizatorius: čia leksinė ir sintaksinė analizė sulietos į vieną procesą. Tokie analizatoriai paprastai yra mažiau našūs, tačiau turi naudingų savybių (pvz., galima rasti dviejų gramatikų kompoziciją), kurių neturi įprasti leksinės ir sintaksinės analizės metodai.

2.4. Kodo generavimas

Kad nereikėtų kiekvieno kompiliatoriaus perrašyti skirtingoms kompiuterių architektūroms, naudojamos *tarpinės formos* (TF). Kodo generavimo metu ASM yra transliuojamas pirmiausia į tarpinę formą, kurios kodas tada gali būti optimizuojamas. Paskui galima kompiliuoti optimizuotos tarpinės formos kodą į mašininį kodą.

Tarpinės formos ir jų optimizacijos algoritmai dažnai yra pakartotinai naudojami keliuose skirtinguose kompiliatoriuose. Dėl šios priežasties kiekvienam kompiliatoriaus kūrėjui nereikia iš naujo generuoti kodo kiekvienai skirtingai palaikomai architektūrai.

2.5. Plečiamumas

Dauguma egzistuojančių programavimo kalbų turi fiksuotą sintaksę ir semantiką. Todėl ne visas idėjas yra vienodai paprasta išreikšti kiekviena programavimo kalba. Bandant išreikšti idėjas kalba, kuri joms nėra pritaikyta, dažnai tenka rašyti daug *pasikartojančio kodo* (angl. *boilerplate code*). Siekiant palengvinti tokio kodo rašymą, naudojami įvairūs metodai:

- **Specializuotos kalbos**. Problemos yra išreiškiamos specializuotose (nuo domeno priklausančiose) kalbose, kurių kodas yra transliuojamas į žemesnio lygmens programavimo kalbą (pvz., gramatikos yra transliuojamos į C/C++ kodą).
- **Makrokomandos** leidžia įterpti jose apibrėžtą kodą į pasirinktas kodo pozicijas.
- **Šablonai** leidžia parametrizuoti ir pakartotinai naudoti pasirinktus kodo fragmentus.
- **Metaprogramavimas** suteikia galimybę programiškai generuoti kodą *metaprogramose*, kuris vėliau įterpiamas į galutinę programą.

- **Kompiliatoriaus įskiepiai** leidžia atlikti dar didesnių įvesties pakeitimų, transformacijų ir papildymų.
- **Aspektai**.

Istoriškai kalbos, kurios realizuodavo bent kai kuriuos iš šių mechanizmų, būdavo vadinamos *plečiamosiomis programavimo kalbomis* [27]. Siekiant geriau atskirti kalbas su sudėtingesniais plėtros mechanizmais įvestas terminas *programavimo kalbos su plečiamąja sintakse* – tai tokios kalbos, kurių sintaksė gali būti plečiama nekeičiant paties kompiliatoriaus kodo. Tačiau net ir šis terminas visiškai neatskleidžia metodo, kuriuo programavimo kalbos sintaksė yra plečiama.

Tad šiame darbe įvedamas terminas *reflektyviai plečiamos programavimo kalbos* (RPP kalbos): tai tokios programavimo kalbos, kurių sintaksę ir semantiką galima plėsti kodo kompiliavimo metu ir kurių plėtiniai bei normalus programos kodas gali būti derinami tame pačiame pradiniame tekste.

3. LITERATŪROS ANALIZĖ

Šiame skyriuje atliekama sintaksinės analizės metodų, kurie būtų tinkami RPP kalbų sintaksinei analizei, paieška ir apžvalga. Apžvelgiami metodai vertinami pagal 3.1. skyrelyje nustatytus kriterijus.

3.1. RPP kalbos sintaksinio analizatoriaus reikalavimai

Kad sintaksinės analizės algoritmas būtų tinkamas RPP kalboms analizuoti, jis turi turėti šias savybes:

1. **Dinamiškai kintančių gramatikų palaikymas**. Kad programavimo kalbos sintaksė galėtų kisti kompiliavimo metu, kalbos gramatika, apibrėžianti tą sintaksę, turi taip pat kisti. Vadinasi, analizės algoritmas turi palaikyti dinamiškai kintančias gramatikas.
2. **Integruota leksinė analizė**. Kad vieną kalbą būtų galima papildyti kita, turi būti galimybė rasti tų dviejų kalbų kompoziciją. Vienareikšmiškai ir be esminių konfliktų dviejų gramatikų kompoziciją galima rasti tik tuo atveju, jei tose gramatikose nėra naudojamos leksemos, kurios apibrėžtos už liejamų gramatikų ribų.
3. **Apibendrinta bekontekstė analizė**. Kad kalbos plėtiniams nebūtų taikomi atsitiktiniai analizatoriaus apribojimai, analizės algoritmas turi sugebėti atpažinti visas bekontekstes gramatikas.
4. **Lokalūs gramatikos plėtiniai**. Siekiant išvengti dviprasmybių tarp kelių aktyvių kalbos plėtinių, turi būti galimybė gramatikos plėtinį aktyvuoti tik ribotam įvesties fragmentui. Plėtiniai, kurie yra aktyvūs tik trumpą laiką, yra vadinami *lokaliaisiais gramatikos plėtiniais*.

5. **Priimtinas našumas**. Analizės algoritmas turi turėti visas šias savybes ir gebėti analizuoti įvestį priimtinu našumu. Našumas yra laikomas priimtinu, jei panašiausias analizės algoritmas, naudojamas praktikoje (lyginant pagal funkcionalumą ir atpažįstamų gramatikų klasę), trunka panašiai laiko vienodoms įvestims išanalizuoti.

3.2. Sintaksinės analizės metodai

LR(k) analizatorių šeima [18]. Tai vienas iš ankstyvesnių analizės algoritmų, kurio variacijos vis dar yra plačiai naudojamos. LR(k) gramatika yra transliuojama į perėjimų lentelę, kuri naudojama analizės metu. Kiekviena lentelės eilutė vaizduoja vieną analizatoriaus būseną. Kiekvienas stulpelis vaizduoja vieną įmanomą terminalinį simbolį. Analizatorius darbą pradeda pradinėje būsenoje. Pagal dabartinę būseną ir dabartinį įvesties simbolį analizės algoritmas iš perėjimų lentelės nustato kitą veiksmą. LR(k) analizatoriuose yra galimi 3 veiksmai: perėjimas į kitą būseną, redukcija ir darbo baigimas. Redukcijoms atlikti yra naudojamas dėklas. Kadangi ne visas gramatikas galima sutransliuoti į tokias perėjimų lenteles, gali atsitikti taip, kad vienoje perėjimų lentelėje bus daugiau kaip vienas veiksmas. Jei taip atsitinka, sakoma, kad tokiose gramatikose yra postūmio / redukcijos konfliktas ir tokios gramatikos nėra tinkamos analizei su LR(k). Nors ir egzistuoja algoritmas, leidžiantis palaipsniui pildant konstruoti perėjimų lenteles [7], gramatikų klasės apribojimai lemia, kad šis algoritmas nėra tinkamas RPP kalboms analizuoti.

GLR analizatorių šeima [29]. Tai LR(k) apibendrinimas, leidžiantis atpažinti beveik visas bekontekstes kalbas. Toks rezultatas pasiekiamas perėjimų lentelėse leidžiant daugiau kaip vieną veiksmą. Analizės algoritmas taip pat praplečiamas, nes analizatorius gali veikti

daugiau kaip vienoje būsenoje vienu metu. Be to, egzistuoja RNGLR [24], kuris yra GLR plėtinys, pašalinantis kai kuriuos GLR trūkumus. RIGLR [23] papildomai statiškai išsprendžia daugumą redukcijų, tačiau tai atliekama su gerokai didesne perėjimų lentele. Galiausiai, egzistuoja SGLR [9], paremtas RNGLR ir pritaikytas analizei be atskiro leksinio analizatoriaus.

Rekursyviai nusileidžiantis analizatorius [6]. Tai vienas iš populiariausių analizės metodų, kurį galima realizuoti net nenaudojant analizatorių generatorius. Analizatorius sudarytas iš abipusiai rekursyvių funkcijų, kiekviena iš jų realizuoja vieną gramatinę taisyklę (ir vieno neterminalinio simbolio analizavimą). Dabartinė analizės pozicija gali būti saugoma globaliame kintamajame, kuris nuolat didinamas atpažįstant naujus terminalinius simbolius. Dėl tokios metodo realizacijos šie analizatoriai nepalaiko kairiosios rekursijos (nes tai sukeltų amžinąją rekursiją) ir dviprasmybių analizės (nes kiekviena funkcija turi darbą baigti ties fiksuota atitinkamo neterminalinio simbolio pabaiga).

Packrat analizatorius [11] [13]. Šis analizatorius yra paremtas nauju gramatikų formalizmu – PEG [12]. Tiek Packrat, tiek PEG paremti rekursyviai nusileidžiančiais analizatoriais. Esminis Packrat skirtumas – tarpiniai analizės rezultatai yra įsimenami ir pakartotinai naudojami. Tokiu būdu užtikrinama, kad tas pats įvesties fragmentas nebus analizuojamas kelis kartus iš naujo. Dėl to šio analizės metodo sudėtingumas blogiausiu atveju nėra eksponentinis. Ignoruojant padidėjusį našumą, Packrat analizatoriams galioja tie patys apribojimai kaip ir rekursyviai nusileidžiantiems analizatoriams (pvz., nepalaikoma kairioji rekursija). Taip pat net ir apribotai gramatikų klasei (PEG) neišeina rasti gramatikų kompozicijos [17].

APEG analizatorius [22]. Tai Packrat adaptacija, skirta dinamiškai kintančioms gramatikoms analizuoti. Kadangi analizės

metodas stipriai paremtas Packrat analizatoriumi, jam galioja tie patys apribojimai.

Specificity analizatorius [3]. Tai unikalus iš viršaus-žemyn analizės metodas, skirtas plečiamosioms gramatikoms analizuoti. Ties bet kuria įvesties pozicija šis analizatorius seka dar neišanalizuotą įvesties dalį ir kandidatinius produkcijų taisyklių likučius. Kad nustatytų, kurią kandidatinę produkciją pasirinkti, analizatorius skaičiuoja produkcijų *konkretumą* ir išrenka leksiškai konkrečiausią produkciją. Todėl šis analizės metodas nepalaiko dviprasmiškų įvesčių, nes vos tik užtiktos dviprasmybės išsprendžiamos taikant konkretumo taisykles. Taip pat šis analizės metodas nepalaiko kairiosios rekursijos ir turi kitų apribojimų, kuriuos reikia spręsti papildomais mechanizmais.

Earley analizatorius [8]. Tai dinaminiu programavimu pagrįstas, apibendrintas analizės algoritmas, skirtas visoms bekontekstėms gramatikoms analizuoti. Kiekvienai įvesties pozicijai Earley analizatorius turi vieną Earley būseną. Ši būsena susideda iš *Earley elementų*. Kiekvienas elementas saugo vieną gramatikos taisyklę, poziciją toje gramatinėje taisyklėje ir pradžios būsenos numerį. Šios informacijos visiškai pakanka žinoti apie kiekvieną įvesties (terminalinį) simbolį ir sintaksinės analizės eigą toje pozicijoje. Pats sintaksinės analizės procesas susideda iš trijų diskrečių žingsnių: prognozavimo, skenavimo ir užbaigimo. Kiekvienas iš šių žingsnių yra taikomas kiekvienam įvesties simboliui. Kadangi Earley analizatorius naudoja neapdorotas gramatikas kaip įvestį, jis yra pakankamai lėtas ir praktikoje be papildomų optimizacijų nėra naudojamas.

Reflective Earley analizatorius [28]. Tai Earley analizatoriaus plėtinys, kuris leidžia analizuoti dinamiškai kintančias gramatikas su lokaliais plėtiniais. Gramatikų plečiamumas yra pasiekiamas praplečiant kiekvieną Earley elementą nauju lauku – aktyvia gramatika.

Įvedus naujų primityvų gramatikose, galima parašyti gramatines taisykles, kurios atnaujina šį aktyvios gramatikos lauką. Dėl šios priežasties Earley analizatorius vienu metu gali analizuoti tą pačią įvestį net su keliomis gramatikomis, jei naujos kalbos / gramatikos pradžios ir / ar pabaigos ribos yra dviprasmiškos. Kaip ir originalus Earley, šis algoritmas nėra pritaikytas analizei be atskiro leksinio analizatoriaus ir nėra efektyvus.

Našus Earley analizatorius [15]. Tai Earley analizatoriaus plėtinys su svarbiomis optimizacijomis, kurios stipriai padidina bendrą analizatoriaus našumą. Didesnis našumas pasiekiamas pakeičiant vidinę gramatikų formą, kuri naudojama analizatoriaus viduje. Vietoje neapdorotų taisyklių, našus Earley analizatorius naudoja determinuotus baigtinius automatus (DBA) gramatikoms vaizduoti. Kiekviename Earley elemente vietoje gramatikos taisyklės ir pozicijos jame dabar yra saugomas vienas laukas: gramatikos baigtinio automato būsenos numeris. Todėl analizės metu sukuriama daug mažiau Earley elementų. Be minėtos Earley modifikacijos, egzistuoja ir **praktinis Earley analizatorius** [2] ir **greitesnis Earley analizatorius** [20], kuriuose pristatoma analogiškai veikianti optimizacija, tačiau ji pasiekiama kitu principu.

Yakker analizatorius [14]. Tai Earley analizatoriaus plėtinys, stipriai įkvėptas našaus Earley analizatoriaus [15]. Yakker analizatorius papildomai praplečia gramatikas su kintamaisiais ir apribojimais, kuriuos galima naudoti dar sudėtingesnėms gramatikoms realizuoti (pvz., fiksuoto pločio laukų analizei). Straipsnio autoriai taip pat siūlo naudoti šį analizatorių be atskiro leksinio analizatoriaus, tačiau nepateikia jokių dviprasmybių šalinimo mechanizmų.

1 lentelėje yra pateikta analizuotų sintaksinės analizės metodų apžvalga.

1 lentelė: Sintaksinės analizės metodų apžvalga

Analizės metodas	Analizatorių šeima	Dinaminės gramatikos	Integruota leksinė analizė	Apibendrintas bekontekstis	Lokalūs plėtiniai	Priimtinas našumas
LR(1)	(G)LR					✔
LALR(1)	(G)LR					✔
GLR	(G)LR			✔		
RNGLR	(G)LR			✔		✔
SGLR	(G)LR		✔	✔		✔
Recursive descent						✔
Packrat	PEG		✔			✔
APEG	PEG	✔				✔
Specificity		✔				✔
Earley	Earley			✔		
Reflective	Earley	✔		✔	✔	
Efficient Earley	Earley			✔		✔
Yakker	Earley			✔		✔

3.3. Susijusios kalbos ir įrankiai

Katahdin [26]. Tai viena iš nedaugelio egzistuojančių RPP kalbų. Gramatikos šioje kalboje yra apibrėžiamos naudojant PEG, o analizė atliekama naudojant pakeistą Packrat analizatorių. Tad galimi kalbos plėtiniai yra stipriai apriboti: nepalaikomos dviprasmybės, visi gramatikos plėtiniai yra globalūs, nėra kairiosios rekursijos ir pan.

SugarJ [10]. Tai programavimo kalba, kurioje sintaksiniai (ir semantiniai) plėtiniai yra apibrėžiami išorinėse bibliotekose. Sintaksinei analizei atlikti naudojamas SGLR analizatorius iš Stratego/SDF [5]. Dėl šios priežasties, kai kaskart užkraunama nauja biblioteka su gramatiniu plėtiniu, visa SGLR perėjimų lentelė turi būti sugeneruota iš naujo su naujai įterptu gramatiniu plėtiniu. Tai lemia, kad visi gramatiniai plėtiniai yra globalūs ir jų užkrovimas nėra efektyvus, nes kiekvienas (net ir nedidelio dydžio) plėtinys verčia perskaičiuoti visą gramatiką ir pergeneruoti perėjimų lenteles.

Neverlang [30]. Tai karkasas, skirtas kompiliatoriams konstruoti, kuriame kuriamos kalbos yra dalijamos į skiltis, o šios paskui yra apibrėžiamos nepriklausomai viena nuo kitos. Kalbos yra gaunamos suliejant pasirinktą aibę skilčių į vieną visumą. Sintaksinė analizė šiame karkase yra atliekama naudojant palaipsniui pildant generuotą LALR(1) analizatorių (kuris yra glaudžiai susijęs su LR(1). Dėl šios priežasties individualių skilčių gramatikos yra stipriai apribotos ir nėra tinkamos visoms programavimo kalboms išreikšti.

3.4. Išvados

Atlikus literatūros analizę paaiškėjo, kad nė vienas iš šiuo metu prieinamų analizės algoritmų nėra visiškai tinkamas RPP kalboms realizuoti. Analogiškai, esamos RPP kalbos ir susiję įrankiai nenaudoja labai specializuotų analizės algoritmų, kurie būtų pritaikyti RPP kalbų

analizei. Todėl tokių kalbų našumas ir plečiamumas yra apriboti dėl pasirinktų sintaksinių analizatorių trūkumų.

Artimiausia analizatorių šeima, įgyvendinanti daugelį RPP kalbų analizatorių reikalavimų, yra Earley. Galima daryti išvadą ir teigti, kad Earley analizatorius (ir jo plėtiniai) yra geras pagrindas naujam sintaksinės analizės algoritmui, tenkinančiam visus RPP kalbų analizatorių reikalavimus, realizuoti.

4. EARLEY VIRTUALIŲJŲ MAŠINŲ SINTAKSINĖS ANALIZĖS METODO SUDARYMAS

Šiame skyriuje yra aprašomas naujas sintaksinės analizės metodas – Earley virtualiosios mašinos (EVM). Analizatorius palaiko visas bekontekstes gramatikas, dinamiškai kintančias gramatikas su lokaliais plėtiniais ir veikia be atskiro leksinio analizatoriaus.

4.1. Earley virtualiosios mašinos

Nors Earley virtualiosios mašinos yra paremtos originaliu Earley analizatoriumi, skirtumai tarp EVM ir Earley analizatoriaus yra tokie dideli, kad EVM galima laikyti visiškai nauju analizės metodu.

EVM gramatikos yra rašomos naudojant EVM gramatikų kalbą. Šios gramatikos yra vadinamos *įvesties gramatikomis*. Jos yra transliuojamos į instrukcijų sekas, vadinamas *sukompiliuotomis gramatikomis*. Norint atlikti analizę, šios instrukcijos yra vykdomos *gijose*.

4.1.1. Gijos ir būsenos

Kiekviena gija (panašiai, kaip ir Earley elementai) turi šiuos laukus:

- **Pradžios poziciją** `origin`, nurodančią įvesties poziciją, kur ši gija paleista.
- **Dabartinę poziciją** `offset`, nurodančią šios gijos įvesties poziciją.
- **Instrukcijos poziciją** `ip`, nurodančią dabar vykdomos instrukcijos numerį (poziciją).

- **Aktyvią gramatiką** `grammar_id`. Šis laukas nurodo šioje gijoje aktyvios gramatikos unikalų numerį ir yra reikalingas reflektyvių gramatikų analizei.

Gijos yra saugomos *būsenose*. Kiekviena būsena yra susieta su viena įvesties pozicija (panašiai, kaip Earley analizatoriuje) ir atitinkamai su vienu terminaliniu simboliu, kuris rastas toje pozicijoje.

Gijos, kurios yra paruoštos vykdyti, yra saugomos *gijų eilėje*. Analizės metu virtualioji mašina išima po vieną giją iš eilės ir ją vykdo, kol ši sustoja. Gija gali sustoti dėl kelių priežasčių: aptiktas neatpažintas terminalinis simbolis, gija sėkmingai baigė darbą arba gija užmigdyta. Gijų eilė gali būti tuščia tik aptikus sintaksinę klaidą arba visos analizės pabaigoje.

Vykdant instrukcijas, gija gali būti perkeliama iš vienos būsenos į kitą (sėkmingai atpažinus terminalinį simbolį), užmigdyta (kreipiantis į kitą neterminalinį simbolį), kopijuojama (atsiradus lokaliai dviprasmybei), sustabdyta (sėkmingai baigus darbą arba radus neatpažintą terminalinį simbolį).

Norint užtikrinti, kad ta pati įvestis nėra analizuojama kelis kartus su ta pačia gramatine taisykle, kiekvienoje būsenoje yra saugoma *sekimo aibė*. Joje yra saugomos poros `<ip, stack>`, kur `ip` yra įvykdytos instrukcijos numeris toje būsenoje, o `stack` – atitinkamos gijos dėklo turinys (kuris dažniausiai tuščias). Šis mechanizmas taip pat apsaugo nuo amžinosios rekursijos, nes prieš paleidžiant naujas gijas, patikrinama, ar sekimo aibėje gija nebuvo vykdyta anksčiau.

4.1.2. Terminalinių simbolių analizė

EVM terminaliniai simboliai yra atpažįstami su `i_match_char` ir `i_match_chars` instrukcijomis. `i_match_char S -> IP` instrukcija turi du operandus: atpažįstamą terminalinį simbolį `S` ir naują

poziciją `IP`. Jei dabartinis įvesties simbolis sutampa su `S`, gija perkeliama į kitą būseną su atnaujinta instrukcijos pozicija `IP` ir gija iš naujo užregistruojama gijų eilėje vykdyti. Jei įvesties simbolis nesutampa su pateiktu operandu, dabartinė gija yra tiesiog nutraukiama. `i_match_chars` veikia analogiškai, tik šiai instrukcijai pateikiamas `S` ir `IP` porų sąrašas: tokiu būdu ši instrukcija gali atpažinti daugiau kaip vieną terminalinį simbolį vienu metu. Abi instrukcijos veikia deterministiškai.

4.1.3. Neterminalinių simbolių analizė

EVM neterminalinių simbolių analizė yra sudėtingesnė ir susideda iš kelių žingsnių:

1. Pirmiausia paleidžiamos atitinkamo neterminalinio simbolio analizės gijos (jei neterminalinis simbolis turi daugiau kaip vieną apibrėžimą, gali būti paleista daugiau kaip viena gija). Tai atliekama su `i_call_dyn S, P` instrukcija, kur `S` yra neterminalinis simbolis, į kurį kreipiamasi, o `P` yra minimalus kreipinio prioritetas (šis operandas reikalingas analizuojant operatorių su skirtingais prioritetais hierarchijas). `i_call_dyn` tik įtraukia naują giją(-as) į gijų eilę (jei prieš tai tokios gijos nebuvo atsižvelgiant į sekimo aibę).
2. Dabartinė gija yra užmigdoma dabartinėje būsenoje su `i_match_syms S -> IP`. Užmigdyta gija yra įtraukiama į dabartinės būsenos užmigdytų gijų sąrašą.
3. Naujai sukurta gija, sėkmingai atpažinusi kviestąjį neterminalinį simbolį atlieka *redukciją* su `i_reduce S, P`, taip informuodama EVM, kad neterminalinis simbolis `S` su prioritetu `P` sėkmingai baigtas atpažinti šioje pozicijoje. Gijos, kurios užmigdytos

pradinėje būsenoje su `S` neterminaliniu simboliu, yra prikeliamos padarant jų kopiją ir ją įkeliant į gijų eilę.

4. Naujai sukurta gija baigia darbą su `i_stop` instrukcija.

Jei atsitinka taip, kad sukurtoji gija nesėkmingai atpažįsta neterminalinį simbolį (t. y. yra nutraukiama prieš `i_reduce` instrukcijos įvykdymą), tada pagrindinė gija lieka amžinai užmigdyta ir yra neprikeliama. Todėl įvykus sintaksės klaidai analizės metu galima nustatyti klaidos priežastį analizuojant įvairių būsenų užmigdytų gijų sąrašus.

4.2. Gramatikų kompiliavimas

Visos gramatikų taisyklės yra sukompiliuojamos į vieną instrukcijų seką. Kompiliavimo metu ryšiai tarp gramatinių taisyklių nėra analizuojami ir skirtingi gramatinių taisyklių fragmentai yra transliuojami į iš anksto numatytus instrukcijų šablonus.

4.3. Bendros paskirties skaičiavimai su EVM

EVM analizatorių galima praplėsti, kad palaikytų bendros paskirties skaičiavimus. Tai leistų apibrėžti gramatines taisykles imperatyviniu stiliumi ir praplėstų bendrą atpažįstamų gramatikų klasę (pvz., leistų atpažinti įvestis su fiksuoto ilgio laukais). Kad toks funkcionalumas būtų pasiektas, reikia įvesti papildomų instrukcijų:

- Sąlyginio (`i_bz`) ir nesąlyginio valdymo perdavimo (`i_br`) instrukcijos leistų apibrėžti ciklus ir sąlyginius sakinius.
- Steko manipuliavimo operacijos (`i_pop`, `i_push`, `i_peek`) leistų gramatikų kalboje apibrėžti ir naudoti kintamuosius.
- Aritmetinės instrukcijos leistų atlikti įvairius skaičiavimus (pvz., `i_int_add`, `i_int_sub` ir t.t.).

4.4. Gramatikų lankstumo plėtimas

Kad būtų lengviau apibrėžti ir komponuoti gramatikas, galima praplėsti EVM gramatikų kalbą naujais elementais:

- **Reguliarių išraiškų operatoriai dešiniosiose produkcijų pusėse** leistų lengviau apibrėžti gramatines taisykles su pasikartojančiais fragmentais nenaudojant rekursijos. Tai papildomai didintų bendrą našumą, nes pasikartojantiems elementams analizuoti užtektų mažiau redukcijų (`i_reduce` kreipinių). Tokius operatorius (`e?`, `e*`, `e+`, `a | b`) galima realizuoti pridedant vieną naują instrukciją `i_fork IP`, kuri padaro dabartinės gijos kopiją ties instrukcijos pozicija `IP` ir įtraukią naują giją į gijų eilę.
- **Gramatinių taisyklių prioritetai** leistų lengviau apibrėžti naujus operatorius. Juos realizuoti galima priskiriant kiekvienai gramatinei taisyklei skaitinę prioriteto reikšmę ir praplečiant `i_match_syms` ir `i_reduce` funkcionalumą.
- **Operatorių asociatyvumą** realizuoti galima naudojant gramatinių taisyklių prioritetus: jei gramatinei taisyklei yra leidžiama rekursyviai kreiptis į save, tada kreipinio ir pačios taisyklės prioritetai turi sutapti; priešingu atveju – kreipinio prioritetas turėtų būti vienetu didesnis.

4.5. Būsimų simbolių ribojimai

Būsimų simbolių ribojimai (angl. *lookahead*) leidžia gramatikose kontroliuoti, kokie simboliai privalo eiti po gramatinės taisyklės kūno jų įvesties nesuvartojant (pvz., galima apibrėžti įdentifikatoriaus gramatinę taisyklę, kuriame nurodyta, kad po tokio žodžio privalo eiti tarpo simbolis). Būsimų simbolių ribojimai skirstomi į dvi kategorijas: fiksuoto ir kintamojo ilgio.

Fiksuoto ilgio būsimų simbolių ribojimus galima realizuoti įdedant vieną papildomą instrukciją `i_advance`, kuri leistų gijai sugrįžti į ankstesnę analizės poziciją per fiksuotą kiekį terminalinių simbolių.

Kintamojo ilgio būsimų simbolių ribojimus galima realizuoti įvedant dvi naujas instrukcijas: `i_push_offset` ir `i_pop_offset`: pirmoji išsaugo dabartinę gijos poziciją dėkle, o antroji tą poziciją atstato iš dėklo.

Abiem atvejais būsimi simboliai yra analizuojami tomis pačiomis instrukcijomis kaip ir normalūs simboliai (`i_match_chars`, `i_match_syms`), tačiau baigus būsimų simbolių atpažinimą dar yra sugrįžtama į ankstesnę analizės poziciją su `i_advance` arba `i_pop_offset`.

4.6. Nuo duomenų priklausantys ribojimai

Nuo duomenų priklausantys ribojimai gramatikose leidžia pagal atpažįstamą turinį kontroliuoti sintaksinės analizės eigą. Pvz., pagal anksčiau atpažintą XML žymės turinį vykdyti terminalinių simbolių atpažinimą ir įsitikinti, kad atsidarančios ir užsidarančios XML žymės sutampa. Šis funkcionalumas EVM viduje yra realizuojamas įterpiant papildomas instrukcijas ir gramatinius elementus, kuriais įvesties fragmentus galima išsaugoti lokaliuose kintamuosiuose (gijų dėkle), o paskui vykdyti įvesties atpažinimą pagal tų kintamųjų turinį.

4.7. Abstraktaus sintaksės medžio konstravimas

EVM analizatoriuje abstrakčius sintaksės medžius galima konstruoti dviem būdais:

Automatinis analizės medžio konstravimas automatiškai sukonstruoja analizės medį (arba supakuotą analizės mišką, jei įvestyje yra dviprasmybių). Tai realizuojama papildant EVM gijas dar

vienu elementu – iki šiol sukonstruotu analizės medžio fragmentu. Šis gijos laukas yra automatiškai atnaujinamas vykdant `i_match_syms` ir `i_reduce` instrukcijas, kurios papildo sukonstruoto medžio fragmentą naujomis šakomis.

Rankinis analizės medžio konstravimas leidžia tiksliau sukonstruoti norimą abstraktų sintaksės medį (pvz., nevykdant informacijos nenešančių šakų konstravimą), tačiau vartotojas turi pats gramatikas papildyti kreipiniais, apibrėžiančiais, kaip ASM bus konstruojamas. Šis medžio konstravimo mechanizmas yra realizuojamas įtraukiant naujų EVM instrukcijų ir gramatinių elementų, kurie leidžia detaliai kontroliuoti konstruojamų medžių formą (šakų viršūnių pavadinimus, vaikinius elementus ir pan.).

EVM taip pat palaiko ASM konstravimą naudodamas atidėtus semantinius veiksmus. Šis medžio konstravimo mechanizmas pirmąkart pritaikytas Yakker analizatoriuje [16].

4.8. Reflektyvių gramatikų sintaksinė analizė

Kad EVM gebėtų analizuoti reflektyvias gramatikas, egzistuoja mechanizmų, kuriuos naudojant kartu galima atpažinti tokias gramatikas:

- Kiekviena gija turi dabar aktyvios gramatikos numerį, kuris naudojamas `i_call_dyn` ir `i_match_syms` instrukcijoms vykdyti. Tai leidžia naudoti kelias gramatikas vienu metu.
- Kiekvienos gramatikos taisyklės yra suskirstytos į *domenus*. Viena gramatinė taisyklė gali priklausyti keliems domenams vienu metu. Gramatikų autoriai išreikštinai gali apibrėžti naujus domenus ir priskirti jiems skirtingų gramatinių taisyklių.
- Naujos gramatikos yra sudaromos automatiškai sujungiant visus aktyvius domenus į vieną visumą.

- Analizės metu galima dinamiškai keisti aktyvių domenų aibę su tam skirtomis instrukcijomis, kurios pasiekiamos per atitinkamus gramatinius elementus (domenų aktyvavimo sakinius). Tai leidžia keisti kiekvienoje gijoje naudojamą gramatiką.
- Sukompiliuotos gramatikos yra laikomos *gramatikų moduliuose*. Gramatikų moduliai gali būti dinamiškai sukurti ir užkrauti analizės metu. Dinamiškai aktyvuojant tuose moduliuose apibrėžtus domenus yra praplečiama dabar naudojama gramatika ir taip pasiekiama reflektyvių gramatikų analizė.

4.9. EVM našumo tobulinimai

Siekiant padidinti EVM analizės našumą, taikomos trys optimizacijos[1]:

1. **Šiukšlių surinktuvas** periodiškai automatiškai šalina kai kurių būsenų sekimo aibes arba visas būsenas. Tai sumažina naudojamos atminties kiekį ir padidina naudojamos atminties lokalumą.
2. **Dinaminių neterminalinių simbolių kreipinių eliminavimas** leidžia supaprastinti naujų gijų kūrimą keičiant `i_call_dyn` instrukcijas su tiesioginiais `i_call` kreipiniais. Šis mechanizmas tinka tuomet, kai yra statiškai žinoma aktyvių domenų aibė.
3. **Instrukcijų poaibių konstrukcija** yra pati svarbiausia ir sudėtingiausia EVM optimizacija, kuri analizės metu leidžia apjungti kelių gramatinių taisyklių instrukcijų sekas į vieną optimizuotą instrukcijų seką. Ji paremta jau egzistuojančiu poaibių konstrukcijos algoritmu, skirtu nedeterminuotų į

[1] Šiame darbe terminas optimizacija yra interpretuojamas kaip pakeitimų ir algoritmų rinkinys, dėl kurio padidinamas sintaksinės analizės metodo našumas (tiek vykdymo laiko, tiek sunaudojamos atminties atžvilgiu)

determinuotų baigtinių automatų konvertavimui [21], tačiau pritaikytu dirbi su instrukcijų sekomis. Optimizavimo metu suliejami sutampančių gramatinių taisyklių priešdėliai ir, kaip to rezultatas, išvengiama pakartotinės įvesties fragmentų analizės. Ši optimizacija traktuoja instrukcijų sekas kaip nedeterminuotą baigtinį automatą (NBA) ir atlieką specializuotą poaibių konstrukciją, kuri konvertuoja NBA į ekvivalentų determinuotą baigtinį automatą. Optimizuotos instrukcijos atpažįsta vienodą įvestį, tačiau tam reikia daug mažiau gijų (pvz., vietoje 20–40 gijų, reikalingų atpažinti vienai programavimo kalbos išraiškai, pakanka tik vienos).

4.10. Išvados

- EVM analizės metodas tenkina visus funkcinius RPP kalbų analizės reikalavimus. Todėl šio metodo teikiamo funkcionalumo pakanka tokių kalbų analizei.
- EVM terminalinių simbolių apdorojimas nėra efektyvus (nes vienam simboliui atpažinti reikia interpretuoti mažiausiai vieną instrukciją ir užregistruoti atnaujintą giją sekimo aibėje).
- Norint dar labiau padidinti našumą, reikalingi papildomi leksinių dviprasmybių šalinimo mechanizmai.
- Sekimo aibėje ir kitose analizatoriaus struktūrose saugoma perteklinė informacija, atsisakius jos kaupimo būtų galima dar labiau padidinti analizės našumą.

5. EARLEY VIRTUALIOSIOS MAŠINOS SU INTEGRUOTA LEKSINE ANALIZE

Šiame skyriuje pristatomos Earley virtualiosios mašinos su integruota leksine analize (SEVM). Tai praplėsta EVM versija su patobulinta vidine analizatoriaus struktūra ir naudojamais algoritmais taip, kad būtų galima pritaikyti naujas optimizacijas neprarandant turimo funkcionalumo.

5.1. EVM su integruota leksine analize

Esminiai SEVM patobulinimai ir pakeitimai:

- SEVM būsenos yra kuriamos tik prie neterminalinių simbolių ribų, o ne kiekvienam įvesties terminaliniam simboliui.
- Įtraukti nauji dviprasmybių šalinimo mechanizmai, leidžiantys analizės metu šalinti dviprasmybes ir taip sumažinti žingsnių skaičių analizei atlikti.
- Sukurta nauja gramatikų kalba su padidinta raiška, kurios pakanka realių programavimo kalbų gramatikoms apibrėžti.
- Gramatikos dabar kompiliuojamos į naują tarpinę formą – MIR, kuri vis dar sudaryta iš instrukcijų, tačiau turi griežtesnę struktūrą. Ši nauja forma yra tinkamesnė dinaminiam MIR vertimui į mašininę instrukcijų kalbą.
- Papildomos optimizacijos stipriai padidina bendrą analizatoriaus našumą.
- EVM sekimo aibės pakeistos efektyvesniais (tiek naudojamos atminties, tiek laiko atžvilgiu) pakartotinės analizės vengimo mechanizmais.

5.2. Gramatikų raiškos gerinimas

SEVM gramatikų kalboje yra 3 nauji esminiai elementai, padidinantys gramatikų išraiškingumą.

Abstrakčios gramatinės taisyklės yra tokios gramatinės taisyklės, kurias apibrėžiant nereikia nurodyti jų realizacijos. Tada arba toje pačioje gramatikoje, arba atskirame gramatiniame plėtinyje galima apibrėžti vieną arba daugiau (konkrečių) gramatinių taisyklių, kurios realizuoja prieš tai apibrėžtą abstrakčią gramatinę taisyklę. Kreipiantis į neterminalinį simbolį, apibrėžtą su abstrakčia gramatine taisykle, iškviečiamos visos tos taisyklės realizacijos. Kitaip tariant, tai yra alternatyvus `Z -> A | B | C` išreiškimo būdas, kur apibrėžiant neterminalinį simbolį `Z` nereikia išvardyti visų jo alternatyvų (`A`, `B` ir `C` šiuo atveju).

Abstrakčios gramatinės taisyklės yra esminis gramatikų plėtimo mechanizmas. Apibrėžiant naują programavimo kalbą ją galima pildyti su gramatiniais plėtiniais, kurie prideda (arba šalina) egzistuojančių abstrakčių gramatinių taisyklių realizacijas.

Abstrakčios gramatinės taisyklės taip pat naudojamos išraiškų ir operatorių hierarchijoms apibrėžti, kur skirtingos išraiškos ir operatoriai gali turėti skirtingų prioritetų. Tai yra realizuojama kiekvienai abstrakčios gramatinės taisyklės realizacijai priskiriant skaitinę prioriteto reikšmę. Paskui kreipiantis į neterminalinį simbolį leidžiama specifikuoti minimalią prioriteto reikšmę, kuri reikalinga atitinkamoje gramatikos pozicijoje. Operatorių asociatyvumas yra realizuojamas taip pat naudojant abstrakčių gramatinių taisyklių prioritetus.

Vardinės prioritetų grupės leidžia suteikti vardus konkrečiai abstrakčios gramatinės taisyklės prioriteto reikšmei. Praktikoje šis mechanizmas palengvina gramatikų apibrėžimą, nes programuotojui

kreipiantis į gramatines abstrakčias taisykles nereikia atsiminti konkrečių skaitinių prioritetų reikšmių.

Dominuojantys terminaliniai simboliai leidžia apibrėžti kai kuriuos terminalinius simbolius kaip dominuojančius. Atliekant poaibių konstrukciją, dominuojantys terminaliniai simboliai virsta dominuojančiais perėjimais tarp determinuoto baigtinio automato būsenų. Jei vykdant poaibių konstrukciją yra bandoma lieti kelis skirtingus perėjimus tarp būsenų, tarp kurių bent vienas yra dominuojantis, tai tik dominuojantis perėjimas paliekamas. Šis mechanizmas palengvina leksemų apibrėžimą ir yra naudojamas dviprasmybėms šalinti.

5.3. Dviprasmybių šalinimas

Kadangi SEVM veikia be atskiro leksinio analizatoriaus, analizės metu kyla daug dviprasmybių, kurios paprastai yra pašalinamos leksiniame analizatoriuje. Pavyzdžiui, analizatorius turi gebėti atskirti raktažodžius (`if`, `while`) nuo funkcijų ar kintamųjų vardų. Taip pat turi būti galimybė vieną nuo kito atskirti skirtingo ilgio operatorius su vienodais priešdėliais (pvz.: `/`, `/*` ir `*`). Įprastai analizatoriuose su integruota leksine analize toks dviprasmybių šalinimas atliekamas po analizės, taikant dviprasmybių šalinimo filtrus sukonstruotame analizės miške [4], tačiau šis mechanizmas nėra efektyvus, nes kiekviena analizės metu aptikta dviprasmybė reikalauja papildomų resursų ir lėtina tolimesnę analizę.

Neigiamos redukcijos yra pirmasis ir bendriausias būdas, leidžiantis šalinti dviprasmybes. Pirmąkart jis pasiūlytas SGLR analizatoriuje [9], bet sėkmingai pritaikytas ir SEVM analizatoriui. Šis mechanizmas leidžia apibrėžti neigiamas gramatines taisykles, kurios

leidžia atmesti kai kurias įprastai priimamas įvestis (pvz., iš funkcijos vardo taisyklės atmesti `while` ir kitus raktažodžius).

Norint šį mechanizmą realizuoti SEVM viduje, skirtas didelis dėmesys optimizuotų instrukcijų sekų tvarkai užtikrinti: svarbu, kad pirma suveiktų gramatinė taisyklė, atmetanti atpažintą raktažodį kaip nekorektišką vardą, prieš suveikiant to vardo redukcijai.

Neigiami neterminalinių simbolių atpažinimai leidžia aptikti, kai neterminalinis simbolis neatpažintas. Šis mechanizmas yra naudojamas godiesiems sąrašams atpažinti.

Leksemų lygmens dviprasmybių šalinimas naudoja poaibių konstrukciją, kurios metu kelios gramatinės taisyklės gali būti sujungtos į vieną, ir dominuojančius terminalinius simbolius, kur terminalinis simbolis iš vienos gramatinės taisyklės gali dominuoti „virš“ terminalinio simbolio iš kitos gramatinės taisyklės. Šis mechanizmas gali būti taip pat panaudotas raktažodžiams atskirti nuo vardų, bet tai atliekama efektyviau (su daug mažesniu redukcijų skaičiumi).

5.4. Analizatoriaus optimizacijos

SEVM analizatoriuje yra taikomos 3 naujos esminės optimizacijos.

JIT gramatikų transliavimas leidžia transliuoti tarpinę gramatikų formą (MIR) į mašininį kodą, taip išvengiant instrukcijų interpretavimo. JIT gramatikų transliavimas yra realizuojamas taikant instrukcijų kodo šablonus: kiekviena instrukcija yra verčiama į atitinkamą mašininio kodo fragmentą, o pats mašininis kodas yra generuojamas naudojant LLVM IR biblioteką, kad į analizatorių nereikėtų integruoti kiekvienos skirtingos procesoriaus architektūros palaikymo. MIR tarpinė forma yra apibrėžta taip, kad kuo labiau atitiktų LLVM IR – LLVM bibliotekoje naudojamą tarpinę formą.

Determinuotų poaibių iškėlimas yra nauja optimizacija, reikalinga JIT gramatikų transliavimo trūkumams pašalinti: kadangi kiekviena instrukcija yra verčiama į mašininį kodą, o kai kurios instrukcijos (ypač `CtlMatchChar` ir `CtlMatchChars`, naudojamos terminaliniams simboliams atpažinti) dažnai kartojasi ir išsiskleidžia į didelį mašininio kodo instrukcijų skaičių, dėl to sukompiliuotos mašininių instrukcijų sekos taip pat yra didelės apimties. Tokių instrukcijų sekų generavimas taip pat užima daug laiko (ypač taikant LLVM biblioteką). Dėl šios priežasties taikomas determinuotų poaibių iškėlimas: determinuotos terminalinių simbolių atpažinimo instrukcijų sekos iškeliamos į atskirą determinuotą baigtinį automatą. Atlikus iškėlimą, originalioje instrukcijų sekoje paliekami tik kreipiniai į šiuos išorinius automatus ir taip išvengiama instrukcijų transliavimo į mašininį kodą ir didelių generuoto kodo apimčių.

Determinuotų terminalinių simbolių atpažinimo iškėlimas į atskirus automatus taip pat leidžia papildyti tų automatų funkcionalumą: konkrečiau, galima realizuoti godžiuosius atpažinimus ir taip atskirti operatorių `/*` nuo `/` ir `*` sekos.

Iš dalies įtrauktos redukcijos leidžia statiškai išspręsti redukcijas, kurios įvyksta taikant kairiąją rekursiją: įvykus paprastai redukcijai, SEVM analizatorius turi patikrinti, ar yra užmigdytų gijų (užduočių), kurios būtų prikeltos dėl redukcijos. Aptiktas tokias gijas reikia prikelti: padaryti jų kopijas ir paruošti jas vykdyti. Jei kreipinys į neterminalinį simbolį yra rekursyvus iš kairės, galima statiškai nustatyti, kurioje vietoje gija bus prikelta, ir prikėlimui pakartotinai naudoti dabartinę giją, taip visiškai išvengiant dinaminio gijų prikėlimo mechanizmo.

5.5. Eksponentinio asimptotinio sudėtingumo vengimo būdai

Norint išvengti begalinės rekursijos ir eksponentinio analizės sudėtingumo dviprasmiškose gramatikose, EVM analizatoriuje naudojamos sekimo aibės, kuriose buvo registruojamos ankstesnės analizės pozicijos, kad jų nereikėtų kartoti. Tačiau atlikus EVM profiliavimą, nustatyta, kad šis mechanizmas ne tik reikalauja daug atminties, bet ir jo vykdymas užima daug laiko. SEVM kūrimo metu taip pat pastebėta, kad pakartotinės įvesties fragmentų analizės galima išvengti daugeliu atvejų remiantis kita (ne sekimo aibėse) kaupiama informacija.

Dėl šios priežasties SEVM kūrimo metu nustatytos keturios situacijos (vadinamos konfliktais), dėl kurių gali prireikti pakartotinai analizuoti tuos pačius įvesties fragmentus. Taip pat nustatyta, kokie SEVM pakeitimai yra reikalingi, norint kiekvienos iš jų išvengti.

5.6. Analizės medžio konstravimas

SEVM analizatoriuje atsisakyta rankinio abstraktaus sintaksės medžio konstravimo. Pagrindinis to argumentas: automatiškai konstruojant analizės medžius, galima kontroliuoti tam medžiui užkoduoti naudojamą struktūrą ir ją tiksliau pritaikyti dviprasmybėms vaizduoti. Taip pat tampa paprasčiau rasti dviejų gramatikų kompoziciją, nes abiejose gramatikose tikrai bus atliekamas toks pat analizės medžio kodavimas.

5.7. Išvados

- Tiesioginis instrukcijų interpretavimo keitimas JIT transliavimu gali turėti neigiamų pasekmių analizatoriaus našumui ir atminties

naudojimui, nes terminalinių simbolių atpažinimui užkoduoti reikia didelio skaičiaus instrukcijų.

- Determinuotų poaibių iškėlimas leidžia ne tik sumažinti generuojamų mašininio kodo instrukcijų skaičių, bet ir praplėsti analizatoriaus funkcionalumą, nes iškeltuose baigtiniuose automatuose galima įgyvendinti funkcionalumą, kurį būtų sudėtinga pritaikyti pagrindinėje virtualiojoje mašinoje.
- Taikant poaibių konstrukciją gramatikose su integruota leksine analize ženkliai išauga optimizuotų taisyklių dydis (jas sudarančių instrukcijų skaičius), nes į jas dažnai yra įtraukiamas didelis leksemas apibrėžiančių taisyklių kiekis. To galima išvengti taikant determinuotų poaibių iškėlimą su sutampančių būsenų pakartotiniu naudojimu.
- SEVM yra ne tik našesnis už EVM, bet ir išlaiko visą esminį funkcionalumą. Dėl to SEVM patenkina visus funkcinius RPP kalbų analizės reikalavimus ir su naujais patobulinimais turėtų būti visiškai tinkamas RPP kalbų analizei.

6. EKSPERIMENTINIS EARLEY VIRTUALIŲJŲ MAŠINŲ SU INTEGRUOTA SINTAKSINE ANALIZE VERTINIMAS

Šiame skyriuje SEVM analizatoriaus maketas `north` kontroliuojamame eksperimente yra lyginamas su kitomis analizatorių realizacijomis ir taip bandoma parodyti, kad SEVM yra pakankamai našus naudoti praktikoje.

6.1. Kalbų pasirinkimas

Našumui lyginti pasirinktos dviejų programavimo kalbų gramatikos: ANSI C ir Rust.

ANSI C kalbos gramatika yra nedidelė ir dažnai naudojama sintaksinių analizatorių našumui matuoti. Esminė ANSI C ypatybė – įprasti sintaksiniai analizatoriai šios kalbos tekstų analizavimo metu atlieka ir dalinę semantinę analizę, kad atskirtų kintamųjų / funkcijų vardus nuo tipų vardų. Taikant apibendrintus analizės metodus dalinė semantinė analizė tampa nebūtina, nes apibendrinti analizatoriai geba atpažinti ir išreikšti dviprasmybes, kylančias dėl kintamųjų ir tipų vardų atskyrimo. Dėl šios priežasties šią gramatiką galima traktuoti kaip nedidelės apimties su dideliu skaičiumi dviprasmybių.

Rust kalbos gramatika yra sudaryta iš didesnio skaičiaus gramatinių taisyklių ir neturi jokių papildomų dviprasmybių.

6.2. Analizatorių pasirinkimas

Lyginti pasirinktos šios analizatorių realizacijos:

- **north**. Tai šiame darbe sukurta SEVM realizacija.

- **bison** su **flex**. `bison` yra vienas iš populiariausių sintaksinių analizatorių. Tai LALR(1) analizatorius, gebantis generuoti ir GLR analizatorius. Su šiuo sintaksiniu analizatoriumi taip pat yra naudojamas leksinis analizatorius `flex`.
- **yaep** su **flex**. `yaep` yra Earley analizatoriaus realizacija su įvairiomis optimizacijomis, kurios stipriai padidina šio analizatoriaus našumą. Šis analizatorius neveikia be leksinio analizatoriaus, todėl šiam atvejui pasirinktas `flex`.
- **dparser**. Tai SGLR analizatoriaus realizacija.
- **syn**. Tai rankiniu būdu realizuotas rekursyviai nusileidžiantis analizatorius, skirtas Rust programavimo kalbos kodui analizuoti.

6.3. Lyginimo metodas

Lyginimui atlikti sukurtas įrankis `bench_parsers`. Visas eksperimentas yra sudarytas iš smulkesnių scenarijų, o kiekvienas scenarijus – iš testų. Kiekvienu scenarijumi testuojamas vienas pasirinktas analizės metodas su atitinkama įvestimi. Prieš vykdymą konkretus scenarijus paleidžiamas „apšilimui“, siekiant išvengti su aparatūrine įranga ir operacine sistema susijusių pašalinių efektų. Paskui pateikta įvestis yra analizuojama N kartų pakartotinai su dabartine analizatoriaus realizacija. Pasibaigus scenarijui, kiekvieno individualaus testo laikai išsaugomi į išorinį failą.

6.4. Eksperimento aplinka

Eksperimentas vykdytas kompiuteryje, kuriame įmontuotas Intel i7-3930k procesorius, su 16 GB DDR3 1333 Mhz darbinės atminties.

2 lentelė: Skirtingų analizatorių realizacijų našumo palyginimas

Analizatorius	Kalba	N	IQR	Išskirtys (%)	Laikas
bison	ANSI C	50	0.0008	20.0	0.4974
dparser	ANSI C	50	0.0104	20.0	16.1007
north	ANSI C	50	0.0162	0.0	4.6132
yaep	ANSI C	50	0.0737	0.0	1.7231
north	Rust	50	0.0197	0.0	6.3258
syn	Rust	50	0.0346	0.0	5.5434

6.5. Pradiniai eksperimento duomenys

Analizatoriams lyginti sukonstruotos dvi didelės apimties įvestys:

1. `input_gcc_470k.i` – tai GCC 4.0 kompiliatoriaus pradinis tekstas, sudėtas į vieną failą. Apimtis: 14,8 MB, ∼470000 eilučių, be komentarų.
2. `item_rust_650k.rs` – tai Rust kompiliatoriaus pradinis tekstas, sudėtas į vieną failą. Apimtis: 22,3 MB, ∼650000 eilučių, su komentarais.

6.6. Eksperimento rezultatai

2 lentelėje pateikiami pagrindiniai eksperimento rezultatai: vidutinės laiko trukmės, kurių prireikė įvestims `input_gcc_470k.i` ir `item_rust_650k.rs` išanalizuoti.

`dparser` yra panašiausias analizatorius lyginant su `north`: abu analizatoriai palaiko visas bekontekstes gramatikas ir veikia be atskiro leksinio analizatoriaus, tačiau `north` net tik papildomai palaiko reflektyvių gramatikų analizę, bet ir turi ∼3.5 karto didesnį našumą šiame lyginime. Kaip galima nuspėti, sparčiausias analizatorius yra `bison`, bet ir jo atpažįstamų gramatikų klasė yra labiausiai apribota.

Lyginant `north` su `syn`, `north` demonstruoja išties neblogą našumą, turint omenyje tai, kad `syn` yra rankiniu būdu realizuotas rekursyviai nusileidžiantis analizatorius.

6.7. Pagrįstumas

Vidinis eksperimento pagrįstumas. Buvo imtasi šių veiksmų, norint garantuoti vidinį eksperimento pagrįstumą:

- Visi scenarijai ir testai vykdomi toje pačioje aplinkoje.
- Prieš vykdymą kiekvienas scenarijus paleidžiamas ribotam laikui „apšilimui“, siekiant išvengti nuo operacinės sistemos ar aparatinės įrangos priklausančių pašalinių efektų.
- Gauti rezultatai apžvelgti, kad juose nebūtų didelio skaičiaus išskirčių. Išskirtys reikštų, kad eksperimento scenarijai neigiamai veikia su nekontroliuojamais išoriniais veiksniais.
- Kiekvienas scenarijaus testas vykdomas kelis kartus, siekiant užtikrinti rezultatų tikslumą.

Išorinis eksperimento pagrįstumas. Norint užtikrinti, kad stebėti rezultatai galioja ir už eksperimento konteksto ribų, didelis dėmesys skiriamas analizuojamoms gramatikoms ir jų įvestims pasirinkti: pagrindinis faktorius, kuris yra kintantis ir lemia SEVM lėtumą, yra dviprasmybių skaičius analizės metu. Todėl kalbos ir gramatikos su dideliu dviprasmybių skaičiumi turėtų būti analizuojamos panašiu greičiu kaip ANSI C kalba. Analogiškai, kalbos, kurios yra labiau vienaprasmiškos, turėtų veikti panašiu į Rust našumu. Taip pat abi pagrindinės scenarijuose naudotos įvestys yra paimtos iš tikrų projektų (o ne sintetinės) ir yra didelės apimties, todėl vidutinis laikas, kurio prireikia vienam įvesties simboliui išanalizuoti turėtų išlikti panašus ir kitose tipinėse ANSI C ir Rust įvestyse.

6.8. Išvados

Atlikus Earley virtualiųjų mašinų su integruota leksine analize (SEVM) lyginimą su kitais egzistuojančias analizatoriais, galima daryti išvadą, kad SEVM yra pakankamai našus naudoti praktikoje. Tad SEVM tenkina visus analizatoriaus reikalavimus, kurie reikalingi reflektyviai plečiamoms programavimo kalboms analizuoti. Galima teigti, kad pagrindinis šio darbo tikslas yra pasiektas.

7. BENDROSIOS IŠVADOS

Pagrindinė išvada: suformuluoti teiginiai buvo įrodyti.

Earley virtualiosios mašinos su integruota leksine analize (SEVM) yra tinkamos reflektyvių programavimo kalbų sintaksinei analizei, nes šis analizės metodas patenkina visus pagrindinius tokių kalbų analizatorių reikalavimus. Taip pat dėl šio analizės metodo gaunamas priimtinas našumas ir lankstumas, kad būtų taikomas praktikoje net ir neplečiamosioms kalboms analizuoti.

Papildomos išvados:

- SEVM analizatorius gali efektyviau šalinti leksines dviprasmybes už SGLR.
- Determinuotų poaibių iškėlimas į atskirus baigtinius automatus padidina bendrą analizės našumą. Šis metodas taip pat gali būti taikomas Earley analizatoriams, norint juos pritaikyti analizei be atskiro leksinio analizatoriaus.
- Analizės metu periodiškai suveikiantis šiukšlių surinktuvas ne tik sumažina naudojamos atminties kiekį, bet ir padidina bendrą analizatoriaus našumą.
- Kairioji rekursija gramatinėse taisyklėse veikia efektyviau negu kitos rekursijos rūšys.
- LLVM ORC JIT transliatorius nėra tinkamas dinaminiam gramatikų transliavimui į mašininį kodą.

Galimos tolimesnių tyrimų kryptys:

- Automatinis analizės pratęsimas po sintaksės klaidos aptikimo (remiantis [1] ir/arba [31]).
- Našesnis gramatikų kompiliavimas į mašininį kodą nenaudojant LLVM JIT.

- Rekursijos ir dėklo atminties naudojimas, siekiant sumažinti dinaminių atminties išskyrimų skaičių analizės metu.
- Kvadratinio dešiniosios rekursijos sudėtingumo pašalinimas (remiantis [19]).
- Nuo išdėstymo priklausančių (angl. *layout-sensitive*) kalbų sintaksinė analizė (pvz.: Python).

EXTENSIBLE PARSING WITH EARLEY VIRTUAL MACHINES

Importance and Motivation

Extensible programming languages enable simpler programming language integration, can speed-up programming language evolution and increase programming language longevity. Specialised parsing methods are needed to parse such languages. In this work a new parsing method is presented, which is suitable for parsing reflectively extensible programming (REP) languages. A REP language is a programming language which can be extended dynamically without any direct compiler modifications.

Problem Statement: Virtual-machine-based scannerless parsing of reflectively extensible programming languages, where dynamically changing grammars with local grammar extensions are supported and grammars can be decomposed into several smaller grammars and their extensions.

The research object is the extensible programming language syntax analysis.

The research goal is the creation of a syntax analysis method suitable for parsing reflectively extensible programming (REP) languages.

Objectives:

1. Definition of a grammar definition language,
2. Creation of a virtual machine that would be suitable for generalised context-free parsing with local grammar extensions,
3. Definition of the overall parsing method (SEVM),
4. Lexical analysis integration (scannerless parsing),
5. SEVM proof-of-concept implementation, and

6. SEVM performance evaluation.

Defended Claims

1. No existing parsing algorithm matches the criteria needed to implement a general REP language.
2. The Earley parser or its derivatives can be extended to support parsing REP languages.
3. The proposed SEVM parser offers acceptable parsing performance for practical use.

Research Methods

- The first claim is proved by performing a critical literature survey and using conceptual analysis methods.
- Research by design is applied to construct the main parsing algorithm.
- A controlled experiment is conducted to evaluate SEVM parsing performance.

Results

- The new parsing method SEVM is suitable for REP language syntax analysis.
- The SEVM grammar definition language is not only suitable for defining real programming languages but REP languages as well.
- The `bench_parsers` research tool can be used to compare the performance of different parser implementations.
- The `north_cli` research tool can be used to analyse and inspect the internal state of the SEVM parser.

Scientific Contribution of the Research

- The SEVM is a virtual-machine-based parsing method.
- The deterministic finite automata extraction method that is used to speed up the SEVM grammars can be adapted to other parsing methods to enable performance-wise cheaper (faster) token-level disambiguation.

- SEVM grammar definition language enables more flexible computer language and their extension definition using grammar composition.

Practical Significance of the Results

- Good parsing performance.
- Generalised context-free parsing.
- Only a single grammar definition language is needed to define programming languages.
- The SEVM grammar language allows defining grammars in a modular fashion.
- The SEVM parser supports dynamically changing grammars.
- The parser can be further extended by adding additional instructions.

Approbation

- FedCSIS 2017, 6th Workshop on Advances in Programming Languages (WAPL'17), Prague, Czech Republic, 2017.09.03–07.
- FCSIT 2019, European Conference on Frontiers of Computer Science and Information Technology, Amsterdam, Netherlands, 2019.09.22–24.

Publications

The main results of this dissertation were published in the following papers:

- Šaikūnas A. (2017). Critical Analysis of Extensible Parsing Tools and Techniques. Baltic J. Modern Computing, Vol. 5 (2017), No. 1, pp. 136–145.
- Šaikūnas A. (2019). Parsing with Scannerless Earley Virtual Machines. Baltic J. Modern Computing, Vol. 7 (2019), No. 2, pp. 171–189.

Other papers:

- Šaikūnas A. (2017). Parsing with Earley Virtual Machines. Communication Papers of the 2017 Federated Conference on Computer Science and Information Systems, Vol. 13 (2017), pp. 165–173.
- Šaikūnas A. (2019). Just-in-time Parsing with Scannerless Earley Virtual Machines (*accepted for publishing*).

General Conclusions

The primary conclusion is that the goal of this research has been achieved.

Additional conclusions include the following:

- The SEVM can more effectively eliminate lexical parsing ambiguities than the SGLR.
- The DFA extraction improves overall parsing performance. It also can be used to augment other parsing methods (such as Earley or SGLR) to enable more efficient scannerless parsing.
- The garbage collector not only reduces the overall memory usage but also improves parsing performance.
- Left-recursion (and left calls) is more efficient than right-recursion (due to subset construction and partial reduction incorporation).
- The LLVM ORC JIT is not suitable for dynamic grammar compilation into machine code because its code generator is too slow. It produces high-quality low-level code even when optimisations are turned off, which is unnecessary for SEVM.

LITERATŪRA

[1] Anderson, S., Backhouse, R. (1981). *Locally Least-Cost Error Recovery in Earley's Algorithm*. ACM Trans. Program. Lang. Syst., 318–347.

[2] Aycock, J., Horspool, R. (2002). *Practical Earley Parsing*. Compututer Journal, 620–630.

[3] Brabrand, C., Schwartzbach, M. (2007). *The Metafront System: Safe and Extensible Parsing and Transformation*. Science of Comp. Prog., 2–20.

[4] Brand, M., Scheerder, J., Vinju, J., Visser, E. (2002). *Disambiguation Filters for Scannerless Generalized LR Parsers*. Compiler Construction.

[5] Bravenboer, M., Kalleberg, K., Vermaas, R., Visser, E. (2008). *Stratego/XT 0.17. A Language and Toolset for Program Transformation*. Sci. Comput. Program., 52–70.

[6] Burge, W. (1975). *Recursive Programming Techniques*. Addison-Wesley.

[7] Cazzola, W., Vacchi, E. (2014). *On the incremental growth and shrinkage of LR goto-graphs*. Acta Inf., 419–447.

[8] Earley, J. (1970). *An Efficient Context-free Parsing Algorithm*. Commun. ACM, 94–102.

[9] Economopoulos, G., Klint, P., Vinju, J. (2009). *Faster Scannerless GLR Parsing*. Springer Berlin Heidelberg, Berlin, Heidelberg.

[10] Erdweg, S., Rendel, T., Kästner, C., Ostermann, K. (2011). *SugarJ: Library-based Syntactic Language Extensibility*. SIGPLAN Not., 391–406.

[11] Ford, B. (2002). *Packrat parsing: a practical linear-time*

algorithm with backtracking. M.S. Thesis, Massachusetts Institute of Technology, Cambridge, United States.

[12] Ford, B. (2004). *Parsing Expression Grammars: A Recognition-based Syntactic Foundation*. SIGPLAN Not., 111–122.

[13] Grimm, R. (2004). *Practical Packrat Parsing*. New York University, Computer Science, Tech. Report TR2004-854.

[14] Jim, T., Mandelbaum, Y., Walker, D. (2010). *Semantics and Algorithms for Data-dependent Grammars*. Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '10, ACM, New York, United States.

[15] Jim, T., Mandelbaum, Y. (2010). *Efficient Earley Parsing with Regular Right-hand Sides*. Electronic Notes in Theoretical Computer Science, 135–148.

[16] Jim, T., Mandelbaum, Y. (2011). *Delayed Semantic Actions in Yakker*. Proceedings of the Eleventh Workshop on Language Descriptions, Tools and Applications, LDTA '11, ACM, New York, NY, USA.

[17] Kats, L., Visser, E., Wachsmuth, G. (2010). *Pure and Declarative Syntax Definition: Paradise Lost and Regained*. SIGPLAN Not., 918–932.

[18] Knuth, D. (1965). *On the translation of languages from left to right*. Information and Control, 607–639.

[19] Leo, J. (1991). *A general context-free parsing algorithm running in linear time on every LR(k) grammar without using lookahead*. Theoretical Computer Science, 165–176.

[20] McLean, P., Horspool, R. (1996). *A Faster Earley Parser*. Proceedings of the 6th International Conference on Compiler Construction, CC '96, Springer-Verlag, London, UK.

[21] Rabin, M., Scott, D. (1959). *Finite Automata and Their Decision Problems*. IBM Journal of Research and Development, 114–125.

[22] Reis, L., Bigonha, R., Iorio, V., Amorim, L. (2014). *The Formalization and Implementation of Adaptable Parsing Expression Grammars*. Sci. Comput. Program., 191–210.

[23] Scott, E., Johnstone, A. (2005). *Generalized Bottom Up Parsers With Reduced Stack Activity*. Comput. J., 565–587.

[24] Scott, E., Johnstone, A. (2006). *Right Nulled GLR Parsers*. ACM Trans. Program. Lang. Syst., 577–618.

[25] Scott, E. (2008). *SPPF-Style Parsing From Earley Recognisers*. Electronic Notes in Theoretical Computer Science, 53–67.

[26] Seaton, C. (2007). *A Programming Language Where the Syntax and Semantics Are Mutable at Runtime*. M.S. Thesis, University of Bristol, Bristol, United Kingdom.

[27] Standish, T. (1975). *Extensibility in Programming Language Design*. SIGPLAN Not., 18–21.

[28] Stansifer, P., Wand, M. (2011). *Parsing Reflective Grammars*. Proceedings of the Eleventh Workshop on Language Descriptions, Tools and Applications, LDTA '11, ACM, New York, United States.

[29] Tomita, M. (1985). *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, Norwell, MA, USA.

[30] Vacchi, E., Cazzola, W. (2015). *Neverlang: A framework for feature-oriented language development*. Comp. Lang., Syst. & Struct., 1–40.

[31] Valkering, R. (2007). *Syntax Error Handling in Scannerless Generalized LR Parsers*. M.S. Thesis, University of Amsterdam, Amsterdam, Netherlands.

Audrius Šaikūnas

EARLEY VIRTUALIŲJŲ MAŠINŲ PANAUDOJIMAS PLEČIAMAI PROGRAMAVIMO KALBŲ SINTAKSINEI ANALIZEI

Daktaro disertacijos santrauka

Technologijos mokslai
Informatikos inžinerija (T 007)

Redaktorė: Jorūnė Rimeisytė-Nekrašienė

Vilniaus universiteto leidykla
Saulėtekio al. 9, LT-10222 Vilnius
El. p. info@leidykla.vu.lt,
www.leidykla.vu.lt
Tiražas: 30 egz.