

UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Niki Bizjak

**Lastništvo objektov namesto
avtomatskega čistilca pomnilnika med
lenim izračunom**

MAGISTRSKO DELO
MAGISTRSKI ŠTUDIJSKI PROGRAM DRUGE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA
SMER: RAČUNALNIŠTVO IN INFORMATIKA

MENTOR: doc. dr. Boštjan Slivnik

Ljubljana, 2023

To delo je ponujeno pod licenco *Creative Commons Priznanje avtorstva-Deljenje pod enakimi pogoji 2.5 Slovenija* (ali novejšo različico). To pomeni, da se tako besedilo, slike, grafi in druge sestavine dela kot tudi rezultati diplomskega dela lahko prosto distribuirajo, reproducirajo, uporabljajo, priobčujejo javnosti in predelujejo, pod pogojem, da se jasno in vidno navede avtorja in naslov tega dela in da se v primeru spremembe, preoblikovanja ali uporabe tega dela v svojem delu, lahko distribuira predelava le pod licenco, ki je enaka tej. Podrobnosti licence so dostopne na spletni strani creativecommons.si ali na Inštitutu za intelektualno lastnino, Streliška 1, 1000 Ljubljana.



Izvorna koda diplomskega dela, njeni rezultati in v ta namen razvita programska oprema je ponujena pod licenco GNU General Public License, različica 3 (ali novejša). To pomeni, da se lahko prosto distribuira in/ali predeluje pod njenimi pogoji. Podrobnosti licence so dostopne na spletni strani <http://www.gnu.org/licenses/>.

ZAHVALA

Na tem mestu zapišite, komu se zahvaljujete za izdelavo magistrske naloge. V zahvali se poleg mentorja spodobi omeniti vse, ki so s svojo pomočjo prispevali k nastanku vašega izdelka.

Niki Bizjak, 2023

Vsem rožicam tega sveta.

*"The only reason for time is so that
everything doesn't happen at once."*

— Albert Einstein

Kazalo

Povzetek

Abstract

1	Uvod	1
2	Pregled področja	3
2.1	Upravljanje s pomnilnikom	5
2.2	Rustov model lastništva	7
2.3	STG jezik	10
2.4	Linearen sistem tipov	16
3	Implementacija	19
3.1	Analiza izposoj	20
3.2	Analiza premikov	20
4	Rezultati	21

Seznam uporabljenih kratic

kratica	angleško	slovensko
GHC	Glasgow Haskell compiler	Haskellov prevajalnik Glasgow
STG	Spineless Tagless G-machine	Prevod
AST	Abstract syntax tree	Abstraktno sintaksno drevo

Povzetek

Naslov: Lastništvo objektov namesto avtomatskega čistilca pomnilnika med lenim izračunom

V vzorcu je predstavljen postopek priprave magistrskega dela z uporabo okolja L^AT_EX. Vaš povzetek mora sicer vsebovati približno 100 besed, ta tukaj je odločno prekratek. Dober povzetek vključuje: (1) kratek opis obravnavanega problema, (2) kratek opis vašega pristopa za reševanje tega problema in (3) (najbolj uspešen) rezultat ali prispevek magistrske naloge.

Ključne besede

prevajalnik, nestrog izračun, upravljanje s pomnilnikom, avtomatični čistilec pomnilnika, lastništvo objektov

Abstract

Title: Ownership model instead of garbage collection during lazy evaluation

This sample document presents an approach to typesetting your BSc thesis using L^AT_EX. A proper abstract should contain around 100 words which makes this one way too short. A good abstract contains: (1) a short description of the tackled problem, (2) a short description of your approach to solving the problem, and (3) (the most successful) result or contribution in your thesis.

Keywords

compiler, lazy evaluation, memory management, garbage collector, ownership model

Poglavje 1

Uvod

Pomnilnik je dandanes kljub uvedbi pomnilniške hierarhije še vedno eden izmed najpočasnejših delov računalniške arhitekture. Učinkovito upravljanje s pomnilnikom je torej ključnega pomena za učinkovito izvajanje programov. Upravljanje s pomnilnikom v grobem ločimo na ročno in avtomatično [1]. Pri ročnem upravljanju s pomnilnikom programski jezik vsebuje konstrukte za dodeljevanje in sproščanje pomnilnika. Odgovornost upravljanja s pomnilnikom leži na programerju, zato je ta metoda podvržena človeški napaki. Pogosti napaki sta puščanje pomnilnika (angl. memory leaking), pri kateri dodeljen pomnilnik ni sproščen, in viseči kazalci (angl. dangling pointers), ki kažejo na že sproščene in zato neveljavne dele pomnilnika [1].

Pri avtomatičnem upravljanju s pomnilnikom zna sistem sam dodeljevati in sproščati pomnilnik. Tukaj ločimo posredne in neposredne metode. Ena izmed neposrednih metod je npr. štetje referenc [2], pri kateri za vsak objekt na kopici hranimo metapodatek o številu kazalcev, ki se sklicujejo nanj. V tem primeru moramo ob vsakem spreminjanju referenc zagotavljati še ustrezno posodabljanje števcov, kadar pa število kazalcev pade na nič, objekt izbrišemo iz pomnilnika. Posredne metode, npr. označi in pometi [3], ne posodablajo metapodatkov na pomnilniku ob vsaki spremembi, temveč se izvedejo le, kadar se prekorači velikost kopice. Algoritem pregleda kopico in ugotovi, na katere objekte ne kaže več noben kazalec ter jih odstrani. Ne-

kateri algoritmi podatke na kopici tudi defragmentirajo in s tem zagotovijo boljšo lokalnost ter s tem boljše predpomnjenje [4].

Avtomatično čiščenje pomnilnika pa ima tudi svoje probleme. Štetje referenc v primeru pomnilniških ciklov privede do puščanja pomnilnika, metoda označi in pometi pa nedeterministično zaustavi izvajanje glavnega programa in tako ni primerna za časovno-kritične (angl. real-time) aplikacije. Kot alternativa obem načinom upravljanja s pomnilnikom sistemski programski jezik Rust implementira model lastništva [5]. Med *prevajanjem* zna s posebnimi pravili zagotoviti, da se pomnilnik objektov na kopici avtomatično sprosti, kadar jih program več ne potrebuje. To pa zna storiti brez čistilca pomnilnika in brez eksplicitnega dodeljevanja in sproščanja pomnilnika, zato zagotavlja predvidljivo sproščanje pomnilnika.

V magistrskem delu se bomo primarno ukvarjali s STG jezikom. Operacijska semantika tega veleva, da so vsi izrazi v izvorni kodi v pomnilniku predstavljeni kot zaprtja. Jezik vsebuje izraz `let`, ki na kopici ustvari novo zaprtje, izbirni izraz `case` pa šele dejansko izračuna njegovo vrednost. Jezik STG za čiščenje zaprtij iz kopice uporablja generacijski čistilec pomnilnika [6, 7].

Cilj magistrske naloge je pripraviti simulator STG stroja, nato pa spremeniti STG jezik tako, da bo namesto avtomatičnega čistilca pomnilnika uporabljal model lastništva po zgledu programskega jezika Rust. Zanimalo nas bo, kakšne posledice to v STG stroj prinese, kakšne omejitve se pri tem pojavijo ter do kakšnih problemov lahko pri tem pride. Zavedati se moramo, da obstaja možnost, da koncepta lastništva ni mogoče vpeljati v STG stroj brez korenitih sprememb zasnove stroja samega - v tem primeru bomo podali analizo, zakaj lastništva v STG stroj ni mogoče vpeljati.

Poglavje 2

Pregled področja

Leni funkcijski programski jeziki funkcije obravnavajo kot prvorazredne objekte (angl. first-class objects), kar pomeni, da so lahko funkcije argumenti drugim funkcijam in da lahko funkcije kot rezultate vračajo druge funkcije. Taki jeziki pogosto omogočajo in spodbujajo tvorjenje novih funkcij z uporabo delne aplikacije [8], pri kateri je funkciji podanih le del njenih argumentov. Leni funkcijski programski jeziki uporabljajo nestrogo semantiko, ki deluje na principu prenosa po potrebi (angl. call-by-need) [9], pri kateri se pri klicu funkcij ne izračuna najprej vrednosti argumentov, temveč se računanje izvede šele takrat, ko telo funkcije vrednost dejansko potrebuje. Nestroga semantika je običajno implementirana z ovijanjem izrazov v zakasnitve (angl. thunks) [10], tj. funkcije brez argumentov, ki se evalvirajo šele, kadar je njihova vrednost dejansko zahtevana.

Ker je v funkcijskih jezikih lahko funkcija vrednost argumenta ali rezultata, je lahko izvajanje take funkcije zamaknjeno v čas po koncu izvajanja funkcije, ki je ustvarila vrednost argumenta ali rezultata. Zato klicnih zapisov takih funkcij ni mogoče hraniti na skladu, temveč na kopici [1]. Na kopici so zakasnitve in funkcije shranjene kot *zaprtja* (angl. closures), tj. podatkovne strukture, v katerih se poleg kode hranijo še kazalci na podatke, ki so zahtevani za izračun telesa. Pri izvajanju se tako na kopici nenehno ustvarjajo in brišejo nova zaprtja, ki imajo navadno zelo kratko življenjsko

dobro, zato je nujna učinkovita implementacija dodeljevanja in sproščanja pomnilnika. Haskell za to uporablja *generacijski* avtomatični čistilec pomnilnika [11, 12]. Danes vsi večji funkcijski programski jeziki, ki omogočajo leni izračun, uporabljajo avtomatični čistilec pomnilnika [13, 14, 15, 16, 17].

Lene funkcijske programske jezike najpogosteje implementiramo s pomočjo redukcije grafa [10]. Eden izmed načinov za izvajanje redukcije je abstraktni STG stroj (angl. Spineless Tagless G-machine) [6], ki definira in zna izvajati majhen funkcijski programski jezik STG. STG stroj in jezik se uporabljata kot vmesni korak pri prevajanju najpopularnejšega lenega jezika Haskell v prevajalniku GHC (Glasgow Haskell Compiler) [12].

Ena izmed alternativ STG stroja za izvajanje jezikov z nestrogo semantiko je prevajalnik GRIN [18] (angl. graph reduction intermediate notation), ki podobno kot STG stroj definira majhen funkcijski programski jezik, ki ga zna izvajati s pomočjo redukcije grafa. Napisane ima prednje dele za Haskell, Idris in Agdo, ponaša pa se tudi z zmožnostjo optimizacije celotnih programov (angl. whole program optimization) [19]. Za upravljanje s pomnilnikom se v trenutni različici uporablja čistilec pomnilnika [20].

Programski jezik Rust za upravljanje s pomnilnikom uvede princip lastništva (angl. ownership model) [5], pri katerem ima vsak objekt na kopici natančno enega *lastnika* [21, 22, 23]. Kadar gre spremenljivka, ki si lasti objekt, izven dosega (angl. out of scope), se pomnilnik za objekt sprosti. Rust definira pojem *premiika* (angl. move), pri katerem druga spremenljivka prevzame lastništvo (in s tem odgovornost za čiščenje pomnilnika) in *izposoje* (angl. borrow), pri katerem se ustvari (angl. read-only) referenca na objekt, spremenljivka pa *ne* prevzame lastništva. Preverjanje pravilnosti sproščanja pomnilnika se izvaja *med prevajanjem* v posebnem koraku analize izposoj in premikov (angl. borrow checker). Prevajalnik zna v strojno kodo dodati ustrezne ukaze, ki ustrezno sproščajo pomnilnik in tako na predvidljiv, varen in učinkovit način zagotovi upravljanje s pomnilnikom.

Na podlagi principa lastništva in izposoje iz Rusta je nastal len funkcijski programski jezik Blang [24]. Interpreter jezika zna pomnilnik za zaprtja

izrazov in spremenljivk med izvajanjem samodejno sproščati brez uporabe čistilcev, zatakne pa se pri sproščanju funkcij in delnih aplikacij.

Programski jezik *micro-mitten* [25] je programski jezik, podoben Rustu, ki za upravljanje s pomnilnikom uporablja princip ASAP (angl. *As Static As Possible*) [26]. Prevajalnik namesto principa lastništva izvede zaporedje analiz pretoka podatkov (angl. *data-flow*), namen katerih je aproksimirati statično živost spremenljivk na kopici. Pri tem prevajalnik ne postavi dodatnih omejitev za pisanje kode, kot jih poznamo npr. v Rustu, kjer mora programer za pisanje delujoče in učinkovite kode v vsakem trenutku vedeti, katera spremenljivka si objekt lasti in kakšna je njena življenjska doba. Metoda ASAP še ni dovolj raziskana in tako še ni primerna za produkcijske prevajalnike.

2.1 Upravljanje s pomnilnikom

Pomnilnik je dandanes kljub uvedbi pomnilniške hierarhije še vedno eden izmed najpočasnejših delov računalniške arhitekture. Učinkovito upravljanje s pomnilnikom je torej ključnega pomena za učinkovito izvajanje programov. Upravljanje s pomnilnikom v grobem ločimo na ročno in avtomatično [1]. Pri ročnem upravljanju s pomnilnikom programski jezik vsebuje konstrukte za dodeljevanje in sproščanje pomnilnika. Odgovornost upravljanja s pomnilnikom leži na programerju, zato je ta metoda podvržena človeški napaki. Pogosti napaki sta puščanje pomnilnika (angl. *memory leaking*), pri kateri dodeljen pomnilnik ni sproščen, in viseči kazalci (angl. *dangling pointers*), ki kažejo na že sproščene in zato neveljavne dele pomnilnika [1].

Pri ročnem upravljanju pomnilnika kot ga poznamo npr. pri programskem jeziku C, pride pogosto do dveh vrst napak [1]:

- viseči kazalci (angl. *dangling pointers*) so kazalci na pomnilnik, ki je že bil sproščen.
- puščanje pomnilnika (angl. *memory leak*),

- uporaba po sproščanju (angl. use-after-free), pri kateri skušamo dostopati do pomnilnika, ki je bil že prej sproščen in
- dvojno sproščanje (angl. double free), pri katerem skušamo dvakrat sprostiti isti pomnilniški naslov.

V obeh primerih bo prišlo do nedefiniranega obnašanja sistema za upravljanje s pomnilnikom (angl. memory management system). Ob uporabi po sproščanju lahko pride npr. do dostopanja do pomnilniškega naslova, ki ni več v lasti trenutnega procesa in posledično do sesutja programa. V primeru dvojnega sproščanja pa lahko pride do okvare delovanja sistema za upravljanje s pomnilnikom, kar lahko privede do dodeljevanja napačnih naslovov ali prepisovanja obstoječih podatkov na pomnilniku.

Druga možnost upravljanje s pomnilnikom je avtomatično upravljanje pomnilnika, pri katerem zna sistem sam dodeljevati in sproščati pomnilnik. Tukaj ločimo posredne in neposredne metode.

Ena izmed neposrednih metod je npr. štetje referenc [2], pri kateri za vsak objekt na kopici hranimo metapodatek o številu kazalcev, ki se sklicujejo nanj. V tem primeru moramo ob vsakem spreminjanju referenc zagotavljati še ustrezno posodabljanje števcov, kadar pa število kazalcev pade na nič, objekt izbrišemo iz pomnilnika. Ena izmed slabosti štetja referenc je, da mora sistem ob vsakem prirejanju v spremenljivke posodobiti še števce v pomnilniku, kar privede do povečanja števila pomnilniških dostopov. Prav tako pa metoda ne deluje v primeru pomnilniških ciklov, saj števec referenc nikoli ne pade na nič, kar pomeni da pomnilnik nikoli ni ustrezno počiščen.

Posredne metode, npr. označi in pometi [3], ne posodablajo metapodatkov na pomnilniku ob vsaki spremembi, temveč se izvedejo le, kadar se prekorači velikost kopice. Algoritem pregleda kopico in ugotovi, na katere objekte ne kaže več noben kazalec ter jih odstrani. Nekateri algoritmi podatke na kopici tudi defragmentirajo in s tem zagotovijo boljšo lokalnost ter s tem boljše predpomnjenje [4]. Problem metode označi in pometi pa je njeno nedeterministično izvajanje, saj programer oziroma sistem ne moreta predvideti kdaj se bo izvajanje glavnega programa zaustavilo in tako ni primerna

za časovno-kritične (angl. real-time) aplikacije.

2.2 Rustov model lastništva

Programski jezik Rust je namenjen nizkonivojskemu programiranju, tj. programiranju sistemske programske opreme. Kot tak mora omogočati hitro in predvidljivo sproščanje pomnilnika, zato avtomatični čistilnik pomnilnika ne pride v poštev. Rust namesto tega implementira model lastništva [5], pri katerem zna med *prevajanjem* s posebnimi pravili zagotoviti, da se pomnilnik objektov na kopici avtomatično sprost, kadar jih program več ne potrebuje. Po hitrosti delovanja se tako lahko kosa s programskim jezikom C, pri tem pa zagotavlja varnejše upravljanje s pomnilnikom kot C.

Rust doseže varnost pri upravljanju pomnilnika s pomočjo principa izključitve [21]. V poljubnem trenutku za neko vrednost na pomnilniku velja natanko ena izmed dveh možnosti:

- Vrednost lahko mutiramo preko *natanko enega* unikatnega kazalca
- Vrednost lahko beremo preko poljubno mnogo kazalcev

Premik

Eden izmed najpomembnejših konceptov v Rustu je lastništvo. Ta zagotavlja, da si vsako vrednost na pomnilniku lasti natanko ena spremenljivka. Kadar gre ta izven dosega (angl. out-of-scope), lahko tako Rust ustrezno počisti pomnilnik. Ob klicu funkcije se lastništvo nad argumenti prenese v funkcijo, ta pa postane odgovorna za čiščenje pomnilnika. Naslednji primer prikazuje program, ki se v Rustu ne prevede zaradi težav z lastništvom.

```
1 let number = Complex(0, 1);  
2     let a = number;  
3     let b = number; // Napaka: use of moved value: `number`
```

Rust ✗

Spremenljivka `a` prevzame lastništvo nad vrednostjo `number`. Pravimo tudi, da je bila vrednost `number` *premaknjena* v spremenljivko `a`. Ob premiku postane spremenljivka `number` neveljavna, zato pri ponovnem premiku v spremenljivko `b`, prevajalnik javi napako.

```
1 let number = Complex(1, 0);  
2     f(number);  
3     let x = number; // Napaka: use of moved value: `number`
```

Rust ✗

V zgornjem primeru se spremenljivka `number` pojavi kot argument funkciji `f`. Lastništvo spremenljivke `number` se ob klicu funkcije prenese v funkcijo in ta postane tudi odgovorna za čiščenje njenega pomnilnika. Ob ponovnem premiku v spremenljivko `x`, prevajalnik javi napako.

Izposoja

Drugi koncept, ki ga definira Rust je *izposoja*. Ta omogoča *deljenje* (angl. aliasing) vrednosti na pomnilniku. Izposoje so lahko spremenljive (angl. mutable) `&mut x` ali nespremenljive (angl. immutable) `&x`. V danem trenutku je lahko ena spremenljivka izposojena nespremenljivo oziroma samo za branje (angl. read-only) večkrat, spremenljivo pa natanko enkrat.

Naslednji primer se v Rustu uspešno prevede, ker sta obe izposoji nespremenljivi. Vrednosti spremenljivk `a` in `b` lahko le beremo, ne moremo pa jih spreminjati.

```
1 let number = Complex(2, 1);  
2     let a = &number;  
3     let b = &number;
```

Rust ✓

V naslednjem primeru skušamo ustvariti dve mutable referenci na spremenljivko `number`. Zaradi principa izključevanja to ni mogoče, zato prevajalnik javi napako.


```
1 let mut number = Complex(1, 2);
2     let a = &mut number;
3     let b = &mut number; // Napaka: cannot borrow `number` as
4                          ↪ mutable
                          // more than once at a time
```

Rust ✗

Prav tako ni veljavno ustvariti mutabilne reference na spremenljivko, dokler nanjo obstaja kakršnakoli druga referenca.

```
1 let mut number = Complex(0, 0);
2     let a = &number;
3     let b = &mut number; // Napaka: cannot borrow `number` as
4                          ↪ mutable
                          // because it is also borrowed as immutable
```

Rust ✗

V trenutni različici Rusta 2018 bi se primer [TODO] še vedno uspešno prevedel. Projekt Polonius [27, 28] je v programski jezik namreč uvedel neleksikalne življenjske dobe (angl. non-lexical lifetimes), ki preverjanje izposoj in premikov izvaja nad CFG namesto nad AST programa [22, 23]. Prevajalnik zna s pomočjo neleksikalnih življenjskih dob dokazati, da sta dve zaporedni mutable izposoji varni, če ena izmed njih ni nikjer uporabljena. Na tak način Rust zagotovi bolj drobnozrnat (angl. fine grained) pogled na program, saj prevajalnik vrača napake ob manj veljavnih programih.

Kljub temu, da Rust v svoji dokumentaciji [?] zagotavlja, da je njegov model upravljanja s pomnilnika varen, pa njegovi razvijalci niso nikoli uradno formalizirali niti njegove operacijske semantike, niti sistema tipov, niti modela za prekrivanje (angl. aliasing model). V literaturi se je tako neodvisno uveljavilo več modelov, ki skušajo čimbolj natančno formalizirati semantiko premikov in izposoj. Model Patina [29] formalizira delovanje analize izposoj v Rustu in dokaže, da velja izrek o varnosti (angl. soundness) za safe podmnožico jezika Rust. Model RustBelt [30] poda prvi formalen in strojno preverjen dokaz o varnosti za jezik, ki predstavlja realistično podmnožico jezika Rust. Model Stacked borrows [23] definira operacijsko semantiko za

dostopanje do pomnilnika v Rustu in model za prekrivanje (angl. aliasing model) in ga strojno dokaže. Model Oxide [22] je formalizacija Rustovega sistema tipov in je tudi prva semantika, ki podpira tudi neleksikalne življenjske dobe.

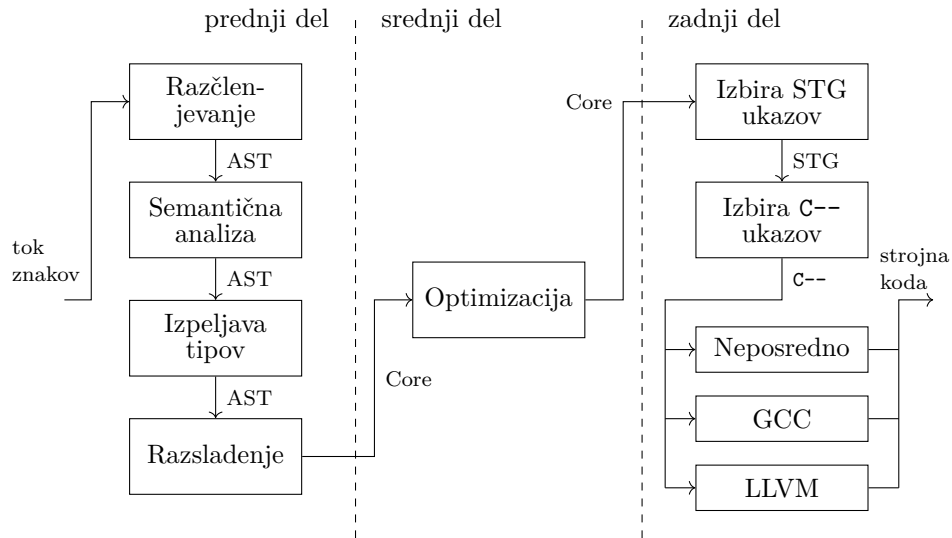
2.3 STG jezik

Na tem mestu bomo na kratko opisali delovanje prevajalnika GHC (angl. Glasgow Haskell compiler), ki izvirno kodo napisano v programskem jeziku Haskell prevede v strojno kodo. Pri prevajanju se program transformira v več različnih vmesnih predstavitev (angl. intermediate representation), mi pa se bomo v magistrskem delu osredotočili predvsem na vmesno kodo imenovano STG jezik (angl. Spineless tagless G-Machine language), katerega delovanje bomo podrobneje opisali v razdelku 2.3.2.

2.3.1 Prevajalnik GHC

Prevajalnik GHC (angl. Glasgow Haskell compiler) prevajanje iz izvirne kode v programskem jeziku Haskell v strojno kodo izvaja v več zaporednih fazah oziroma modulih. Vsaka faza kot vhod prejme izhod prejšnje, nad njim izvede določeno transformacijo in rezultat posreduje naslednji fazi. Faze glede na njihovo funkcijo v grobem delimo na tri dele. V prednjem delu (angl. front-end) se nad izvirno kodo najprej izvede leksikalna analiza, pri kateri se iz toka znakov, ki predstavljajo vhodni program pridobi abstraktno sintaktično drevo (angl. abstract syntax tree). Nad drevesom se izvede še zaporedje semantičnih analiz pri katerih se preveri ali je program pomensko pravilen. Sem sodi razreševanje imen, pri kateri se razreši vsa imena spremenljivk iz uvoženih modulov v programu in preveri ali so vse spremenljivke deklarirane pred njihovo uporabo. Izvede se še preverjanje tipov, kjer se za vsak izraz izpelje njegov najbolj splošen tip in preveri ali se vsi tipi v programu ujemajo.

Ker je Haskell programski jezik, ki je namenjen ljudem, je precej velik jezik, z veliko različnimi sintaktičnimi oblikami in konstruktorji. Tako lahko



Slika 2.1: Pomembnejše faze prevajalnika Glasgow Haskell compiler

npr. programer napiše isto kodo na več različnih načinov. S perspektive piscev prevajalnikov pa to pomeni veliko več dela, saj je potrebno zagotoviti, da se vsi konstrukti v strojno kodo prevedejo pravilno. Za odpravljanje te težave se v zadnjem koraku prednjega dela prevajalnika imenovanem razsladenje (angl. desugarification), sintaktično drevo jezika Haskell pretvori v sintaktično drevo jezika Core. Ta je zelo majhen funkcijski programski jezik, ki temelji na lambda računu, kljub svojem majhnemu naboru konstruktorov pa še vedno omogoča zapis poljubnega Haskell programa. Naslednje faze tako operirajo nad dosti manjšim jezikom, kar precej poenostavi prevajanje.

Srednji del (angl. middle-end) prevajalnika sestavlja zaporedje optimizacij, ki kot vhod sprejmejo program v Core jeziku in vrnejo izboljšan program v Core jeziku. Rezultat niza optimizacij se posreduje zadnjemu delu (angl. back-end) prevajalnika, ki poskrbi za prevajanje Core jezika v strojno kodo, ki se lahko neposredno izvaja na procesorju. Na tem mestu se Core jezik prevede v STG jezik, ta pa se nato prevede v programski jezik C++. Slednji je podmnožica programskega jezika C in ga je mogoče v strojno kodo prevesti na tri načine: neposredno ali z enim izmed prevajalnikov LLVM ali GCC. Prednost take vrste prevajanja je v večji prenosljivosti programov, saj znata

LLVM in GCC generirati kodo za večino obstoječih procesorskih arhitektur, poleg tega pa imata vgrajene še optimizacije, ki pohitrijo delovanje izhodnega programa.

2.3.2 STG jezik

Leni funkcijske programske jezike najpogosteje implementiramo s pomočjo redukcije grafa [10]. Eden izmed načinov za izvajanje redukcije je abstraktni STG stroj (angl. Spineless Tagless G-machine) [6], ki definira in zna izvajati majhen funkcijski programski jezik STG. STG stroj in jezik se uporabljata kot vmesni korak pri prevajanju najpopularnejšega lenega jezika Haskell v prevajalniku GHC (Glasgow Haskell Compiler) [12].

Lena evalvacija

Ena izmed najpomembnejših lastnosti lenih funkcijskih programskih jezikov je njihova nestroga semantika (angl. non-strict semantic). Ta narekuje kako se evalvirajo oziroma računajo argumenti pri klicu funkcije. Pri jezikih z nestrogo semantiko se računanje vrednosti argumentov funkcije ne izvede ob klicu, temveč šele takrat ko se izvaja telo funkcije. Ker se vrednosti argumentov ne izračunajo nemudoma, temveč se računanje zamakne v čas dokler funkcija vrednosti argumenta dejansko ne potrebuje, je taka semantika imenovana tudi za leno.

$$f\perp = \perp$$

Leno evalvacijo najpogosteje implementiramo s pomočjo zakasnitev (angl. thunks). Te so na pomnilniku predstavljene kot kazalec na kodo, ki izračuna njihovo vrednost. Ob evalvaciji zakasnitve se najprej izračuna njihova vrednost, izračunano vrednost pa se shrani v strukturo na pomnilniku, da je ob naslednji evalvaciji ni potrebno ponovno računati. Na tak način se z nestrogo semantiko doseže, da se vsak izraz izračuna *največ enkrat*. Če se argument ne pojavi nikjer v telesu funkcije, se zakasnitve nikoli ne računa, če pa se v te-

lesu pojavi večkrat, se vrednost izračuna enkrat, za vsako nadaljno evalvacijo argumenta pa se preprosto vrne vrednost shranjeno na pomnilniku.

Nestrogo semantiko je moč implementirati na dva različna načina:

- model potisni in vstopi (angl. push / enter model)
- model izračunaj in apliciraj (angl. eval / apply model)

Abstraktni STG stroj je zgodovinsko najprej uporabljal model potisni / vstopi [6]. Trenutna različica STG stroja pa uporablja model evalviraj / apliciraj [7], saj se je empirično izkazal za bolj učinkovito implementacijo nestroge semantike.

Definicija jezika

V sledečem poglavju bomo podali formalno definicijo jezika STG definirano v [7]. Od originalne implementacije v [6] se razlikuje po tem, da implementacija ni več brez oznak (angl. tagless), temveč nosi vsak objekt na kopici še dodatno polje z informacijo o njegovi vrsti. Ker je število tipov objektov majhno, se za oznako objekta navadno uporablja kar celoštevilčna vrednost. V originalni implementaciji so bili vsi objekti na kopici predstavljeni enotno, kar je pomenilo bolj kompaktno predstavitev podatkov v pomnilniku, prav tako pa STG stroju ni bilo treba preverjati vrste objektov ob vsakem klicu funkcije. Prednost ponovne uvedbe oznak pa je v tem, da STG stroju ni potrebno vzdrževati dveh ločenih skladov za argumente in vrednosti, kar pa tudi poenostavi delovanje čistilca pomnilnika.

Sledi formalna definicija STG jezika. Pri tem bomo spremenljivke označevali s poševnimi malimi tiskanimi črkami x, y, f, g , konstruktorje pa s poševnimi velikimi tiskanimi črkami C .

$literal \quad := \quad \underline{int} \mid \underline{double} \qquad \text{primitivne vrednosti}$

STG jezik podpira dva primitivna (angl. unboxed) podatkovna tipa: celoštevilske vrednosti in števila s plavajočo vejico. Poleg tega omogoča

uvajanje novih algebraičnih podatkovnih tipov. Objekte algebraičnih tipov tvorimo s pomočjo konstruktorjev C .

$$a, v \quad := \quad literal \mid x \quad \text{argumenti so atomarni}$$

Vsi argumenti pri aplikaciji funkcij in primitivnih operacij so v A-normalni obliki (angl. A-normal form) [31], kar pomeni, da so atomarni (angl. atomic). Tako je vsak argument ali primitivni podatkovni tip ali pa spremenljivka. Pri prevajanju v STG jezik lahko prevajalnik sestavljene argumente funkcij priredi novim spremenljivkam z ovijanjem v `let` izraz in spremenljivke uporabi kot argumente pri klicu funkcije. Pri tem je potrebno zagotoviti, da so definirane spremenljivke unikatne oziroma, da se ne pojavijo v ovitem izrazu. Aplikacijo funkcije $f (\oplus x y)$ bi tako ovili v `let` izraz `let $a = \oplus x y$ in $f a$` , s čemer bi zagotovili, da so vsi argumenti atomarni.

$$\begin{array}{ll} k & := \bullet \quad \text{neznana mestnost funkcije} \\ & \mid n \quad \text{znana mestnost } n \geq 1 \end{array}$$

Prevajalnik lahko med prevajanjem za določene funkcije določi njihovo mestnost (angl. arity), tj. število argumentov, ki jih funkcija sprejme. Ker pa je STG funkcijski jezik, lahko funkcije nastopajo tudi kot argumenti drugih funkcij, zato včasih določevanje mestnosti ni mogoče. Povsem veljavno bi bilo vse funkcije v programu označiti z neznano mestnostjo \bullet , a je mogoče s podatkom o mestnosti klice funkcij implementirati bolj učinkovito, zato se med prevajanjem izvaja tudi analiza mestnosti.

$expr$	$:=$	a	atom
		$f^k a_1 \dots a_n$	klic funkcije ($n \geq 1$)
		$\oplus a_1 \dots a_n$	primitivna operacija ($n \geq 1$)
		let $x = obj$ in e	
		case e of $\{alt_1; \dots; alt_n\}$	

Pri tem velja, da so vse primitivne operacije *zasičene*, kar pomeni, da sprejmejo natanko toliko argumentov, kot je mestnost (angl. arity) funkcije. Če programski jezik omogoča delno aplikacijo primitivnih funkcij, potem je potrebno take delne aplikacije z η -dopolnjevanjem razširiti v saturirano obliko. Pri tem delno aplikacijo ovijemo v nove lambda izraze z uvedbo novih spremenljivk, ki se ne pojavijo nikjer v izrazu. Tako npr. izraz $(+ \ 3)$, ki predstavlja delno aplikacijo vgrajene funkcije za seštevanje prevedemo v funkcijo $\lambda x.(+3x)$ in s tem zadostimo pogoju saturiranosti.

alt	$:=$	$C \ x_1 \dots x_n \rightarrow expr$	algebraična alternativa
		$x \rightarrow expr$	privzeta alternativa

Podatkovne objekte na kopici je mogoče ustvarjati le z enim konstruktom, in sicer **let** izrazom. Ta nam omogoča prirejanje objekta spremenljivki, ki je vidna v telesu **let** izraza.

obj	$:=$	$FUN(x_1 \dots x_n \rightarrow e)$	aplikacija
		$PAP(f \ a_1 \dots a_n)$	delna aplikacija
		$CON(C \ a_1 \dots a_n)$	konstruktor
		THUNK e	zakasnitev
		BLACKHOLE	črna luknja

Objekt FUN predstavlja funkcijsko ovojnico (angl. closure) z argumenti x_1, \dots, x_n in telesom e , ki pa se lahko poleg argumentov x_i sklicuje še na druge proste spremenljivke. Pri tem velja, da je lahko funkcija aplicirana na več kot n ali manj kot n argumentov, tj. je curryrana.

Objekt PAP predstavlja delno aplikacijo (angl. partial application) funkcije f na argumente x_1, \dots, x_n . Pri tem je zagotovljeno, da bo f objekt tipa FUN, katerega mestnost bo *vsaj* n .

Objekt CON predstavlja saturirano aplikacijo konstruktorja C na argumente a_1, \dots, a_n . Pri tem je število argumentov, ki jih prejme konstruktor natančno enako številu parametrov, ki jih zahteva.

Objekt THUNK predstavlja zakasnitev izraza e . Kadar se vrednost izraza uporabi, tj. kadar se izvede `case` izraz, se izračuna vrednost e , THUNK objekt na kopici pa se nato posodobi s preusmeritvijo (angl. indirection) na vrednost e . Pri evalvaciji zakasnitve se objekt THUNK na kopici zamenja z objektom BLACKHOLE, s čemer se preprečuje puščanje pomnilnika [32] in neskončnih rekurzivnih struktur. Objekt BLACKHOLE se lahko pojavi le kot rezultat evalvacije zakasnitve, nikoli pa v vezavi v `let` izrazu.

2.4 Linearen sistem tipov

Večina programskih jezikov uporablja neomejen (angl. unrestricted) sistem tipov, kar omogoča da je lahko vsaka spremenljivka v programu uporabljena poljubno mnogokrat.

Linearen sistem tipov zahteva, da je vsaka spremenljivka uporabljena *natanko enkrat*. Tak sistem tipov omogoča implementacijo bolj učinkovitega čiščenja pomnilnika, saj zagotavlja, da lahko objekt izbrišemo takoj po uporabi spremenljivke. Poleg linearnih tipov pa taki sistemi tipov običajno določajo tudi neomejene tipe (angl. unrestricted types), ki omogočajo večkratno uporabo spremenljivk, saj se izkaže, da so v večini primerov linearni tipi preveč omejujoči.

Programski jezik Granule [33] je *strong* funkcijski jezik, ki v svojem sis-

temu tipov združuje tako linearne, kot tudi (angl. graded modal) tipe. Za čiščenje pomnilnika se uporablja avtomatski čistilec.

V članku [34] programski jezik Granule razširijo s poenostavljenimi pravili za lastništvo in izposoje na podlagi tistih iz Rusta. Za čiščenje pomnilnika graded modal tipov, se še vedno uporablja avtomatski čistilec, medtem ko se za čiščenje unikatnih in linearnih tipov uporablja Rustov model upravljanja s pomnilnikom.

Poglavje 3

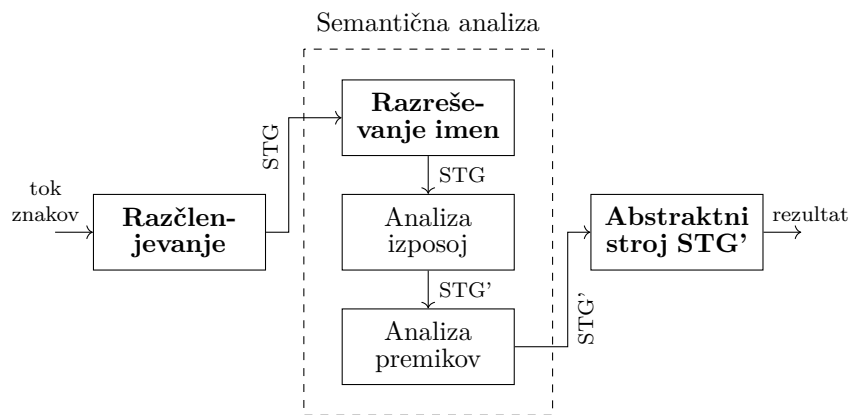
Implementacija

V fazi semantične analize prevajalnik izvede najprej razreševanje imen (angl. name resolution), ki mu sledita analiza izposoj (angl. borrow check) in premikov (angl. move check).

Pri razreševanju imen prevajalnik preveri, ali so vsa imena spremenljivk definirana pred njihovo uporabo. Prevajalnik se v tej fazi rekurzivno sprehodi čez abstraktno sintaksno drevo STG jezika. Pri tem vzdržuje kontekst trenutno živih spremenljivk. Pri uporabi spremenljivk prevajalnik preveri, ali se ime nahaja v kontekstu in v nasprotnem primeru vrne napako. Če se proces razreševanja imen zaključi brez napake, je zagotovljeno, da tekom izvajanja programa ne bo prišlo do napake zaradi uporabe nedefinirane spremenljivke.

Pri analizi izposoj prevajalnik zagotovi, da izposoja ne živi dlje od spremenljivke, ki jo referencira.

```
-- Glavna funkcija
main = THUNK(
  let a = THUNK(12) in
    &a
)
```



Slika 3.1: Faze implementirane prevajalnika

3.1 Analiza izposoj

3.2 Analiza premikov

Poglavje 4

Rezultati

Literatura

- [1] R. Jones, A. Hosking, E. Moss, The garbage collection handbook: the art of automatic memory management, CRC Press, 2023.
- [2] G. E. Collins, A method for overlapping and erasure of lists, Communications of the ACM 3 (12) (1960) 655–657.
- [3] J. McCarthy, Recursive functions of symbolic expressions and their computation by machine, part I, Communications of the ACM 3 (4) (1960) 184–195.
- [4] R. R. Fenichel, J. C. Yochelson, A LISP garbage-collector for virtual-memory computer systems, Communications of the ACM 12 (11) (1969) 611–612.
- [5] S. Klabnik, C. Nichols, The Rust Programming Language, 2nd Edition, No Starch Press, 2023.
- [6] S. L. P. Jones, Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine, Journal of functional programming 2 (2) (1992) 127–202.
- [7] S. Marlow, S. P. Jones, Making a fast curry: push/enter vs. eval/apply for higher-order languages, ACM SIGPLAN Notices 39 (9) (2004) 4–15.
- [8] P. Hudak, Conception, evolution, and application of functional programming languages, ACM Computing Surveys 21 (3) (1989) 359–411.

-
- [9] M. Scott, Programming language pragmatics, 4th Edition, Morgan Kaufmann, 2016.
 - [10] S. L. Peyton Jones, The implementation of functional programming languages (prentice-hall international series in computer science), Prentice-Hall, Inc., 1987.
 - [11] P. M. Sansom, S. L. Peyton Jones, Generational garbage collection for Haskell, in: Proceedings of the conference on Functional programming languages and computer architecture, 1993, pp. 106–116.
 - [12] S. P. Jones, K. Hammond, W. Partain, P. Wadler, C. Hall, S. Peyton Jones, The Glasgow Haskell Compiler: a technical overview, in: Proceedings of Joint Framework for Information Technology Technical Conference, Keele, DTI/SERC, 1993, pp. 249–257.
 - [13] D. A. Turner, Miranda: A non-strict functional language with polymorphic types, in: Conference on Functional Programming Languages and Computer Architecture, Springer, 1985, pp. 1–16.
 - [14] E. Czaplicki, Elm : Concurrent FRP for functional GUIs, Senior thesis, Harvard University 30 (2012).
 - [15] T. H. Brus, M. C. van Eekelen, M. Van Leer, M. J. Plasmeijer, Clean — a language for functional graph rewriting, in: Functional Programming Languages and Computer Architecture: Portland, Oregon, USA, September 14–16, 1987 Proceedings, Springer, 1987, pp. 364–384.
 - [16] D. Syme, The f# compiler technical overview (3 2017).
 - [17] M. Sperber, R. K. Dybvig, M. Flatt, A. Van Straaten, R. Findler, J. Matthews, Revised6 report on the algorithmic language Scheme, Journal of Functional Programming 19 (S1) (2009) 1–301.
 - [18] U. Boquist, T. Johnsson, The GRIN project: A highly optimising back end for lazy functional languages, in: Implementation of Functional Lan-

- guages: 8th International Workshop, IFL'96 Bad Godesberg, Germany, September 16–18, 1996 Selected Papers 8, Springer, 1997, pp. 58–84.
- [19] P. Podlovics, C. Hruska, A. Péntzes, A modern look at GRIN, an optimizing functional language back end, *Acta Cybernetica* 25 (4) (2022) 847–876.
- [20] U. Boquist, Code optimization techniques for lazy functional languages, *Doktorsavhandlingar vid Chalmers Tekniska Hogskola* (1495) (1999) 1–331.
- [21] R. Jung, Understanding and evolving the Rust programming language, Ph.D. thesis, Saarland University (2020).
- [22] A. Weiss, O. Gierczak, D. Patterson, A. Ahmed, Oxide: The essence of Rust, *arXiv preprint arXiv:1903.00982* (2019).
- [23] R. Jung, H.-H. Dang, J. Kang, D. Dreyer, Stacked borrows: an aliasing model for rust, *Proceedings of the ACM on Programming Languages* 4 (POPL) (2019) 1–32.
- [24] T. Kocjan Turk, Len funkcijski programski jezik brez čistilca pomnilnika, Master's thesis, Univerza v Ljubljani (2022).
- [25] N. Corbyn, Practical static memory management, Tech. rep., University of Cambridge, BA Dissertation (2020).
- [26] R. L. Proust, ASAP: As static as possible memory management, Tech. rep., University of Cambridge, Computer Laboratory (2017).
- [27] N. Matsakis (2018). [link].
URL <https://smallcultfollowing.com/babysteps/blog/2018/04/27/an-alias-based-formulation-of-the-borrow-checker/>
- [28] N. Matsakis, N. Nethercote, P. D. Faria, R. Rakic, D. Wood, M. Jasper, S. Pastorino, F. Klock, Non-lexical lifetimes (nll) fully stable: Rust blog

(Aug 2022).

URL <https://blog.rust-lang.org/2022/08/05/nll-by-default.html>

- [29] E. Reed, Patina: A formalization of the rust programming language, University of Washington, Department of Computer Science and Engineering, Tech. Rep. UW-CSE-15-03-02 264 (2015).
- [30] R. Jung, J.-H. Jourdan, R. Krebbers, D. Dreyer, Rustbelt: securing the foundations of the rust programming language, *Proc. ACM Program. Lang.* 2 (POPL) (dec 2017). doi:10.1145/3158154.
URL <https://doi.org/10.1145/3158154>
- [31] C. Flanagan, A. Sabry, B. F. Duba, M. Felleisen, The essence of compiling with continuations, in: *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation, 1993*, pp. 237–247.
- [32] R. Jones, Tail recursion without space leaks, *Journal of Functional Programming* 2 (1) (1992) 73–79.
- [33] D. Orchard, V.-B. Liepelt, H. Eades III, Quantitative program reasoning with graded modal types, *Proceedings of the ACM on Programming Languages* 3 (ICFP) (2019) 1–30.
- [34] D. Marshall, D. Orchard, Functional ownership through fractional uniqueness, *Proc. ACM Program. Lang.* 8 (OOPSLA1) (apr 2024). doi:10.1145/3649848.
URL <https://doi.org/10.1145/3649848>