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

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

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 LaTeX. 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 LaTeX. 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

Manjka kratek uvod v delo

1.1 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]:

- uporaba po sproščanju (angl. use-after-free), pri kateri program dostopa do bloka pomnilnika, ki je že bil sproščen in
- dvojno sproščanje (angl. double free), pri katerem se skuša isti blok pomnilnika sprostiti dvakrat.

V obeh primerih pride 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 upravljanja s pomnilnikom je avtomatično upravljanje pomnilnika, pri katerem zna sistem sam dodeljevati in sproščati pomnilnik. Pri avtomatičnem upravljanju s pomnilnikom nikoli ne pride do visečih kazalcev, saj je objekt na kopici odstranjen le v primeru, da nanj ne kaže kazalec iz nobenega drugega živega objekta. Ker pri je avtomatičnem upravljanju sistem za upravljanje s pomnilnikom edina komponenta, ki sprošča pomnilnik, je tudi zagotovljeno, da nikoli ne pride do dvojnega sproščanja. Glede na način delovanja ločimo posredne in neposredne metode. Pri neposrednih metodah zna sistem za upravljanje s pomnilnikom prepoznati živost objekta neposredno iz zapisa objekta na pomnilniku, pri posrednih metodah pa sistem s sledenjem kazalcem prepozna vse žive objekte, vse ostalo pa smatra za nedostopne oziroma neuporabljene objekte in jih ustrezno počisti.

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 števcev, kadar pa število kazalcev pade na nič, objekt izbrišemo iz pomnilnika. Ena izmed slabosti štetja referenc je, da mora sistem ob vsakem prirejanju vrednosti v spremenljivko posodobiti še števec v pomnilniku, kar privede do povečanja števila pomnilniških dostopov. Prav tako pa metoda ne deluje v primeru pomnilniških ciklov, kjer dva ali več objektov kažeta drug na drugega. V tem primeru števec referenc nikoli ne pade na nič, kar pomeni da pomnilnik nikoli ni ustrezno počiščen.

Posredne metode, npr. označi in pometi [3], ne posodabljajo 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. Za razliko od štetja referenc pa zna algoritem označi in pometi počistiti tudi pomnilniške cikle in se ga zato včasih uporablja v kombinaciji s štetjem referenc. Programski jezik Python za čiščenje pomnilnika primarno uporablja štetje referenc, periodično pa se izvede še korak metode označi in pometi, da odstrani pomnilniške cikle, ki jih prva metoda ne zmore [5].

1.2 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 [6], pri katerem zna med *prevajanjem* s posebnimi pravili zagotoviti, da se pomnilnik objektov na kopici avtomatično sprosti, 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 [7]. 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

V nadaljevanju si bomo na primerih ogledali principa premika in izposoje v jeziku Rust. V vseh primerih bomo uporabljali terko `Complex`, ki

predstavlja kompleksno število z dvema celoštevilskima komponentama in je definirana kot `struct Complex(i32, i32)`.

1.2.1 Premik

Eden izmed najpomembnejših konceptov v Rustu je lastništvo. Ta zagotavlja, da si vsako vrednost na pomnilniku lasti natanko ena spremenljivka. Ob prirejanju, tj. izrazu `let x = y`, pride do *premika* vrednosti na katero kaže spremenljivka `y` v spremenljivko `x`. Po prirejanju postane spremenljivka `y` neveljavna in se nanjo v nadaljevanju programa ni več moč sklicevati. Kadar gre spremenljivka, ki si lasti vrednost na pomnilniku izven dosega (angl. out-of-scope), lahko tako Rust ustrezno počisti njen pomnilnik. Naslednji primer prikazuje program, ki se v Rustu ne prevede zaradi težav z lastništvom.

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

Spremenljivka `a` prevzame lastništvo nad vrednostjo na katero kaže spremenljivka `number`, tj. strukturo `Complex(0, 1)`. Ob premiku postane spremenljivka `number` neveljavna, zato pri ponovnem premiku v spremenljivko `b`, prevajalnik javi napako.

Pravila lastništva [6] so v programskem jeziku Rust sledeča:

- Vsaka vrednost ima lastnika.
- V vsakem trenutku je lahko lastnik vrednosti le eden.
- Kadar gre lastnik izven dosega (angl. out-of-scope), je vrednost sproščena.

Rust v fazi analize premikov (angl. move check) preveri veljavnost pravil in na ustrezna mesta doda ukaze za sproščanje pomnilnika. Ker si vsako vrednost lasti natanko ena spremenljivka, lahko vrednost sprostimo takoj, ko gre ta izven dosega.

Funkcije

Kadar je spremenljivka uporabljena v argumentu pri klicu funkcije, je vrednost spremenljivke premaknjena v funkcijo. Če se vrednost spremenljivke uporabi za sestavljanje rezultata funkcije, potem je vrednost ponovno premaknjena iz funkcije in vrnjena klicatelju. Naslednji primer prikazuje identiteto implementirano v Rustu. Pri klicu funkcije, je vrednost spremenljivke `number` premaknjena v klicano funkcijo, ker pa je ta uporabljena pri rezultatu funkcije, je vrednost ponovno premaknjena v spremenljivko `a` v klicatelju.

```
1  fn identiteta(x: Complex) -> Complex { x }
2
3  let number = Complex(0, 1);
4  let a = identiteta(number);
```

Rust ✓

V naslednjem primeru funkcija `prepisi` prevzame lastništvo nad vrednostjo `Complex(0, 1)`, vrne pa novo vrednost `Complex(2, 1)`, pri čemer ne uporabi argumenta funkcije. Funkcija `prepisi` je tako odgovorna za čiščenje pomnilnika vrednosti argumenta.

```
1  fn prepisi(x: Complex) -> Complex { Complex(2, 1) }
2
3  let number = Complex(0, 1);
4  let a = prepisi(number);
```

Rust ✓

1.2.2 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. Preverjanje veljavnosti premikov se v Rustu izvaja v fazi analize izposoj (angl. borrow

checker), v kateri se zagotovi, da reference ne živijo dlje od vrednosti na katero se sklicujejo, prav tako pa poskrbi, da je lahko vrednost ali izposojena enkrat mutabilno ali da so vse izposoje imutabilne.

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.

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

Rust ✓

Zaradi principa izključitve je v Rustu prepovedano ustvariti več kot eno mutable referenco na objekt. V primeru, da bi bilo dovoljeno ustvariti več mutabilnih referenc, bi namreč lahko več niti hkrati spreminjalo in bralo vrednost spremenljivke, kar krši pravila varnosti pomnilnika, saj lahko privede do težav s sinhronizacijo? (angl. data races). V naslednjem primeru skušamo ustvariti dve mutabilni referenci, zaradi česar prevajalnik ustrezno javi napako.

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

Rust ✗

Prav tako v Rustu ni veljavno ustvariti mutabilne reference na spremenljivko, dokler nanjo obstaja kakršnakoli druga referenca. V nasprotnem primeru bi lahko bila vrednost v eni niti spremenjena, medtem ko bi jo druga nit brala.

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

Rust ✗

Življenjske dobe

Kot smo že omenili, analiza izposoj v Rustu zagotovi, da v danem trenutku na eno vrednost kaže le ena mutabilna referenca ali da so vse reference nanjo imutabilne. Prav tako pa mora prevajalnik v tej fazi zagotoviti, da nobena referenca ne živi dlje od vrednosti, ki si jo izposoja. To doseže z uvedbo *življenjskih dob* (angl. lifetimes), ki predstavljajo časovne okvirje, v katerih so reference veljavne. Prevajalnik navadno življenjske dobe določi implicitno s pomočjo dosegov (angl. scopes). Vsak ugnezden doseg uvede novo življenjsko dobo, za katero velja, da živi največ tako dolgo kot starševski doseg. Kadar se namreč izvedejo vsi stavki v dosegu, bo pomnilnik vseh spremenljivk definiranih v dosegu sproščen.

Naslednji primer prikazuje program, ki se v Rustu ne prevede zaradi težav z življenjskimi dobami. Program je sestavljen iz dveh dosegov:

- Spremenljivka `outer` živi v zunanjem dosegu, prevajalnik ji dodeli življenjsko dobo `'a`.
- Nov blok povzroči uvedbo novega dosega z življenjsko dobo `'b`, zato prevajalnik spremenljivki `inner` določi življenjsko dobo `'b`.

```rust
let outer;
    {
        let inner = 3;
        outer = &inner;  // Napaka: `notranja` does not live
        // long enough
    }
```

Rust ✗

Ker je notranji doseg uveden znotraj zunanjega dosega, si prevajalnik tudi

označi, da je življenjska doba `'a` daljša od dobe `'b`. To pomeni, da bodo vse spremenljivke, ki so definirane znotraj notranjega dosega živele manj časa od spremenljivk definiranih v zunanjem dosegu. V našem primeru bi tako spremenljivka `outer` kazala na spremenljivko, ki je že bila uničena, s čemer pa bi v program uvedli viseč kazalec, kar pa krši varnostni model jezika, zato Rust vrne napako.

Najbrž bi bilo tukaj smiselno razložiti še, da Rust ne zna sam izračunati *vseh* življenjskih dob in da jih moramo v primeru funkcij z več argumenti, definirati ročno.

Od leta 2022 Rust podpira neleksikalne življenjske dobe (angl. non-lexical lifetimes) [8, 9], pri katerih se preverjanje izposoj in premikov izvaja nad grafom poteka programa (angl. control flow graph) namesto nad abstraktnim sintaktičnim drevesom programa [10, 11]. 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.

Potrebno je poudariti, da bi se vsi primeri v tem poglavju v trenutni različici Rusta zaradi neleksikalnih življenjskih dob vseeno prevedli. Primere smo namreč zaradi boljšega razumevanja in jedrnatosti prikaza nekoliko poenostavili. Zato za vse primere v tem poglavju predpostavljamo, da so tako premaknjene, kot tudi izposojene spremenljivke v nadaljevanju programa še nekje uporabljene.

Kljub temu, da Rust v svoji dokumentaciji [6] 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 [12] formalizira delovanje analize izposoj v Rustu in dokaže, da velja izrek o varnosti (angl. soundness) za varno (angl. safe) podmnožico jezika Rust. Model RustBelt [13] poda prvi forma-

len in strojno preverjen dokaz o varnosti za jezik, ki predstavlja realistično podmnožico jezika Rust. Model Stacked borrows [11] definira operacijsko semantiko za dostopanje do pomnilnika v Rustu in model za prekrivanje (angl. aliasing model) in ga strojno dokaže. Model Oxide [10] je formalizacija Rustovega sistema tipov in je tudi prva semantika, ki podpira tudi neleksikalne življenjske dobe.

Poglavje 2

STG

Glede na paradigmo delimo programske jezike na imperativne in deklarativne. Imperativni jeziki, kot so npr. C, Rust, Python, ..., iz vhodnih podatkov izračunajo rezultat s pomočjo stavkov, ki spreminjajo notranje stanje programa. Ukazi oziroma stavki podrobno opisujejo kako naj se program izvaja, in jih je zato nekoliko lažje prevesti v strojne ukaze, ki jih zna izvajati procesor. Pri deklarativnem programiranju pa je program podan kot višjenivojski opis želenega rezultata, ki ne opisuje dejanskega poteka programa. Med deklarativne jezike štejemo npr. SQL, katerega program je opis podatkov, ki jih želimo pridobiti in kako jih želimo manipulirati, ne da bi podrobno opisovali kako naj sistem izvede te operacije.

Med deklarativne programske jezike pa spadajo tudi funkcijski. Ti za osnovno operacijo nad podatki smatrajo čisto (angl. pure) funkcijo, tj. funkcijo, ki za izračun svoje vrednosti uporablja le vrednosti svojih argumentov, ne pa tudi drugega globalnega stanja. Prednost čistih funkcij je, da bodo enaki vhodi vedno vračali enake izhode, kar omogoča optimizacijo programa s pomočjo memoizacije. Osnovni primitiv je v takih jezikih funkcija, kar pomeni, da jih je mogoče prirediti v spremenljivke, jih uporabiti kot argumente in rezultate drugih funkcij. Čist funkcijski program lahko tako opišemo kot kompozicijo čistih funkcij, ki vhod preslikajo v izhod. Taki jeziki pogosto omogočajo in celo spodbujajo tvorjenje novih funkcij z uporabo delne apli-

kacije [14], pri kateri je funkciji podanih le del njenih argumentov.

Poleg čistih funkcij je ena izmed ključnih značilnosti nekaterih funkcijskih jezikov tudi leni izračun (angl. lazy evaluation). Leni izračun je strategija računanja vrednosti izrazov, pri kateri se vrednost izraza ne izračuna takoj, ko je definiran, ampak šele, ko je njegova vrednost dejansko potrebna. Ta pristop je tesno povezan z deklarativno naravo funkcijskega programiranja in omogoča pisanje bolj modularne in učinkovite kode.

V poglavju se bomo podrobneje posvetili *lenim* funkcijskim jezikom, podrobneje bomo definirali leni izračun, ter predstavili Haskell, jezik, ki temelji na čistih funkcijah in lenosti. Podrobneje bomo opisali potek prevajanja jezika v vmesni jezik imenovan STG in opisali delovanje abstraktnega STG stroja, ki ga zna izvajati.

2.1 Leni izračun

Semantika programskega jezika opisuje pravila in principe, ki določajo kako se programi izvajajo. Semantiko za izračun izrazov v grobem delimo na strogo (angl. strict) in nestrogo (angl. non-strict). Večina programskih jezikov pri računanju izrazov uporablja strogo semantiko, ki določa, da se izrazi izračunajo *takoj, ko so definirani*. Nasprotno, nestroga semantika določa, da se izrazi izračunajo šele, ko so dejansko potrebni za nadaljnjo obdelavo, oziroma se sploh ne izračunajo, če niso potrebni.

Večina današnjih programskih jezikov temelji na strogi semantiki. Mednje spadajo tako imperativni jeziki, kot so npr. C, Rust, Python, kot tudi funkcijski programski jeziki kot sta npr. Ocaml in Lisp. Jezikov, ki temeljijo na nestrogi semantiki je bistveno manj, med njimi npr. jeziki Haskell, Miranda in Clean.

```haskell
konjunkcija :: Bool -> Bool -> Bool
konjunkcija p q =
    case p of
        False -> False
        True -> q
```

Haskell

Kot primer si oglejmo razliko pri evalvaciji izraza izraza `konjunkcija False ((1 / 0) == 0)`. Jeziki, ki temeljijo na strogi semantiki ob klicu funkcije najprej izračunajo vrednosti argumentov. Ker pride pri izračunu drugega argumenta do napake, se izvajanje programa tukaj ustavi. Pri jezikih z nestrogo semantiko pa se ob klicu funkcije ne izračunajo vrednosti argumentov, temveč se računanje izvede šele takrat, ko je vrednost dejansko potrebovana. Ker je prvi argument pri klicu funkcije `False`, funkcija drugega argumenta ne izračuna in tako je rezultat klica vrednost `False`.

Če semantika jezika dopušča izračun vrednosti izraza, kljub temu, da njegovi podizrazi nimajo vrednosti, jo imenujemo za strogo semantiko. Bolj formalno lahko uvedemo vrednost $\perp$, ki jo imenujemo tudi dno (angl. bottom) in predstavlja vrednosti izrazov, katerim ni mogoče izračunati normalne oblike. Če izraza *expr* ne moremo evalvirati, lahko tako zapišemo kar $\textbf{eval}(expr) = \perp$. Za funkcijo enega argumenta pravimo, da je *stroga*, če velja naslednji pogoj.

$$f \perp = \perp$$

Za stroge funkcije torej velja, da če se argument ne bo izračunal, se tudi rezultat klica funkcije z argumentom ne bo evalviral. Če funkcija ni stroga, pravimo da je nestroga.

Strogi izračun (angl. eager evaluation) je način implementacija stroge semantike, pri katerem so vrednosti argumentov izračunane pred klicem funkcije. Strogemu izračunu zato pravimo tudi klic po vrednosti (angl. call by value). Obratno, je leni izračun (angl. lazy evaluation) način implementacije nestroge semantike. Pri lenem izračunu izrazi niso izračunani kadar jih *prire-*

dimo spremenljivki, temveč je njihov izračun odložen na poznejši čas, kadar je vrednost izraza dejansko potrebovana. Ker se izračun izvaja po potrebi po vrednosti, to strategijo imenujemo izračun po potrebi (angl. call by need).

Ena izmed slabosti strogega izračuna je v tem, da klicana funkcija morebiti argumenta sploh ne potrebuje, kar pomeni, da se je ob klicu funkcije vrednost argumenta izračunala brez potrebe. Prednost strogega izračuna pa je v tem, da je izvajanje bolj predvidljivo, saj za razliko od lenega izračuna natančno vemo kdaj se bodo argumenti evalvirali. Prednost lenega izračuna pa leži v tem, da so argumenti evalvirani *največ enkrat*. Če namreč argument v telesu funkcije ni nikoli uporabljen, potem tudi ne bo nikoli izračunan. Če pa je uporabljen, bo njegova vrednost izračunana *enkrat* in memozirana v vseh preostalih uporabah. Slabost lenega izračuna pa je v tem, da ga je težje implementirati in da se navadno izvaja počasneje od jezikov s strogim izračunom.

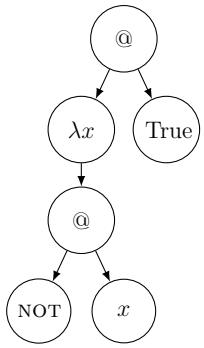
Redukcija grafa

Zgodnejši programski jeziki so bili namenjeni neposrednemu upravljanju računalnika in so kot taki precej odražali delovanje računalniške arhitekture na kateri so se izvajali [14]. Funkcijski programski jeziki omogočajo večji nivo abstrakcije, so bolj podobni matematični notaciji in posledično tudi bolj primerni za dokazovanje pravilnosti programov. Višji nivo abstrakcije pa pomeni, da jih je težje prevajati oziroma izvajati na današnji računalniški arhitekturi. Ena izmed ovir pri implementaciji lenega izračuna je učinkovito upravljanje z odloženimi izrazi (angl. thunks) in zagotavljanje, da se ti izrazi izračunajo le, ko so resnično potrebni.

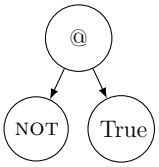
Leni izračun najpogosteje implementiramo s pomočjo redukcije grafa [15, 14]. Pri tej metodi prevajalnik na pomnilniku najprej sestavi abstraktno sintaksno drevo programa, kjer vozlišča predstavljajo aplikacije funkcij oziroma operacije nad podatki, njihova podvozlišča pa odvisnosti med njimi. V procesu *redukcije*, se sintaksno drevo obdeluje z lokalnimi transformacijami, ki ga predelujejo, dokler ni dosežena končna oblika, tj. dokler ni izračunan

rezultat programa. Pri tem se sintaksno drevo zaradi deljenja izrazov (angl. expression sharing) navadno spremeni v usmerjen graf.

Slika 2.1a prikazuje abstraktno sintaktično drevo izraza $(\lambda x\,.\,\text{NOT}\ x)$ True. Aplikacija funkcij je predstavljena z vozliščem @, ki vsebuje dva podizraza: funkcijo, ki se bo izvedla in njen argument. V tem primeru je funkcija anonimni lambda izraz $(\lambda x\,.\,\text{NOT}\ x)$, ki sprejme en argument. Pri redukciji, se vse uporabe parametra x zamenjajo z vrednostjo argumenta `True`. Slika 2.1b prikazuje drevo po redukciji. Ker se je v funkciji argument x pojavil le enkrat, je rezultat redukcije še vedno drevo.



(a) Abstraktno sintaksno drevo *pred* redukcijo

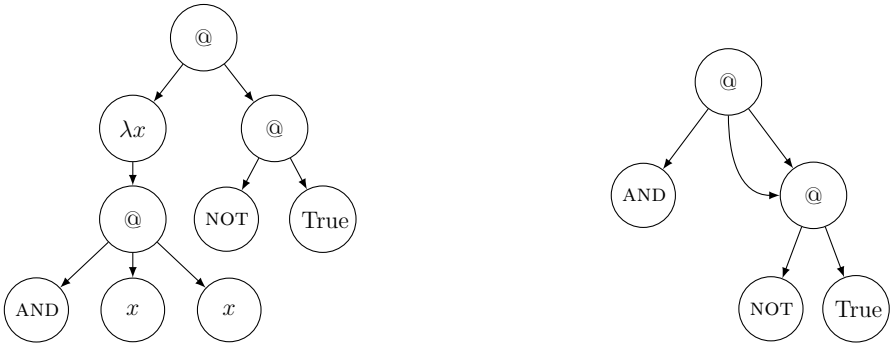


(b) Abstraktno sintaksno drevo *po* redukciji

Slika 2.1: Redukcija grafa izraza $(\lambda x\,.\,\text{NOT}\ x)$ `True`

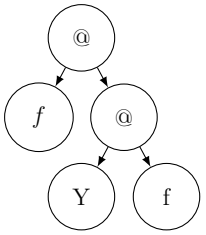
Slika 2.2 prikazuje en korak redukcije abstraktnega sintaksnega drevesa izraza $(\lambda x\,.\,\text{AND}\ x\ x)$ (NOT `True`). Po enem koraku redukcije sintaksnega telesa, se *vse uporabe* parametra x zamenjajo z njegovo vrednostjo. Ker je takih pojavitev več, pa rezultat ni več drevo, temveč acikličen usmerjen graf. Na pomnilniku tak izraz predstavimo z dvema kazalcema na isti objekt. Ko se objekt prvič izračuna, se vrednost objekta na pomnilniku posodobi z izračunano vrednostjo. Ob vseh nadaljnjih uporabah argumenta, tako ne bo potrebno še enkrat računati njegove vrednosti, s čemer dosežemo, da bo vsak argument izračunan največ enkrat.

Slika 2.3 prikazuje dve možni implementaciji ciklične funkcije $Y\ f = f\ (Y\ f)$. Za razliko od primerov na slikah 2.1 in 2.2, pri katerih je bilo

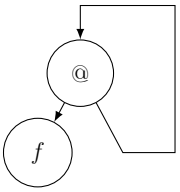


Slika 2.2: Redukcija grafa izraza $(\lambda x\,.\,\text{AND}\ x\ x)\ (\text{NOT}\ \texttt{True})$

reducirano sintaktično drevo še vedno usmerjen acikličen graf, pa temu pri funkciji Y ni več tako. Funkcija Y je namreč rekurzivna, kar pomeni, da se sama pojavi kot vrednost svojega argumenta. Na sliki 2.3a je funkcija Y implementirana s pomočjo acikličnega grafa, a v svojem telesu vseeno dostopa do proste spremenljivke Y, zaradi česar obstaja na pomnilniku cikel. Na sliki 2.3b je funkcija implementirana neposredno s pomnilniškem ciklu, kjer je vrednost argumenta kar vozlišče samo.



(a) Funkcija Y implementirana z uporabo proste spremenljivke



(b) Funkcija Y implementirana kot cikličen usmerjen graf

Slika 2.3: Graf funkcije $Y\ f = f\ (Y\ f)$

Zapis grafa na pomnilniku

Leni izračun najpogosteje implementiramo s pomočjo zakasnitev (angl. thunks). Te so na pomnilniku predstavljene kot struktura s kazalcem na kodo, ki izračuna njihovo vrednost in polja, ki vsebujejo vezane in proste spremen-

ljivke. 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. Pravimo, da se vrednost na pomnilniku posodobi. 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 telesu pojavi večkrat, se vrednost izračuna enkrat, za vsako nadaljnjo evalvacijo argumenta pa se preprosto vrne vrednost shranjeno na pomnilniku.

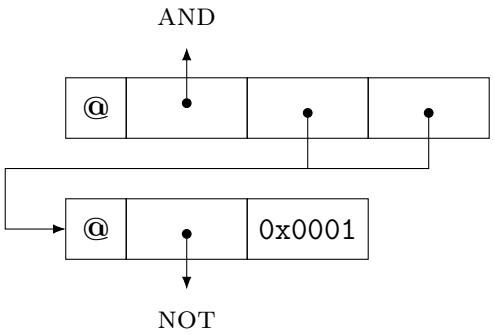
Slika 2.4 prikazuje eno izmed možnih predstavitev vozlišča grafa. Sestavljena je iz oznake vozlišča in polj z vsebino oziroma argumenti $a_1, \ldots, a_n$. V poglavju 2.3 bomo videli, da sta si struktura 2.4 in pomnilniški zapis objektov v jeziku STG precej podobna.

Oznaka	a_1	a_2	...	a_n

Slika 2.4: Pomnilniška predstavitev vozlišča grafa

Slika 2.5 prikazuje predstavitev izraza AND (NOT `True`) (NOT `True`) na pomnilniku. Celoten izraz je sestavljen iz dveh ovojnic, ki predstavljata dve aplikaciji. Spodnja ovojnica predstavlja izraz NOT `True` in je sestavljena kot aplikacija funkcije NOT na argument z vrednostjo `0x0001`, tj. vrednost `True`. Zgornja ovojnica predstavlja aplikacijo funkcije AND na dva argumenta, ki sta predstavljena kot kazalca na drugo ovojnico. Ko se izraz NOT `True` prvič izračuna, se ovojnica na pomnilniku posodobi z izračunano vrednostjo. Tako ob vseh nadaljnjih dostopih, vrednosti ni potrebno še enkrat računati, s čemer je dosežen ravno leni izračun.

Ena izmed slabosti jezikov z lenim izračunom je, da je, za razliko od strogih imperativnih jezikov, zelo težko predvideti, koliko prostora bo program porabil. Ker so v takih jezikih funkcije navadno obravnavane kot primitivi, kar pomeni, da lahko nastopajo kot vrednost argumenta ali rezultata, je lahko njihovo izvajanje 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]. Kot smo že lahko videli



Slika 2.5: Predstavitev izraza AND (NOT `True`) (NOT `True`) na pomnilniku

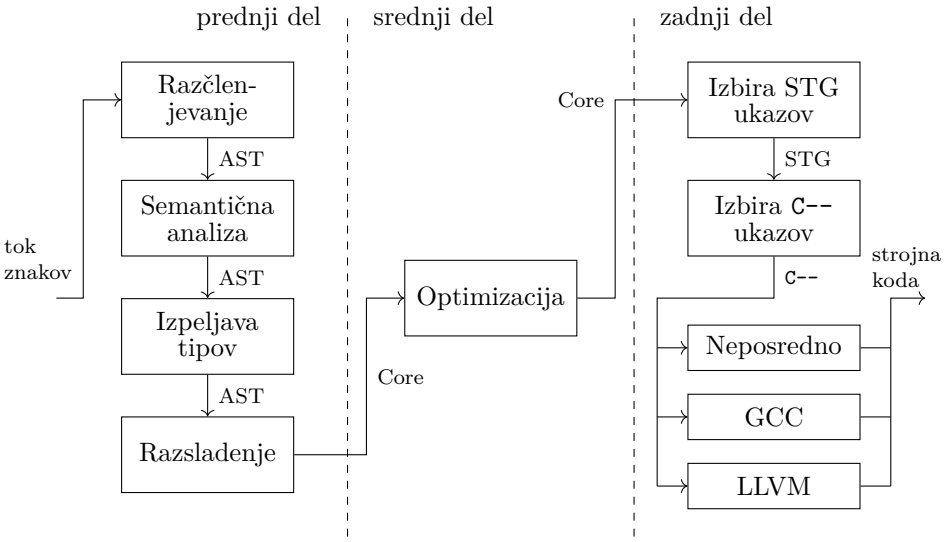
v poglavju 2.1, so zakasnitve in funkcije na kopici shranjene kot ovojnice, tj. podatkovne strukture, v katerih se poleg kode hranijo še kazalci na podatke, ki so zahtevani za izračun telesa. Pri izvajanju se tako na kopici nenehno ustvarjajo in brišejo nove ovojnice, ki imajo navadno zelo kratko življenjsko dobo, zato je nujna učinkovita implementacija dodeljevanja in sproščanja pomnilnika. Haskell za to uporablja *generacijski* avtomatični čistilec pomnilnika [22, 18]. Danes vsi večji funkcijski programski jeziki, ki omogočajo leni izračun, uporabljajo avtomatični čistilec pomnilnika [23, 24, 25, 26, 27].

V nadaljevanju se bomo osredotočili na delovanje prevajalnika GHC (angl. Glasgow Haskell compiler), ki izvorno 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 Prevajalnik GHC

Prevajalnik GHC prevajanje iz izvorne 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 izvorno 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.



Slika 2.6: Pomembnejše faze prevajalnika Glasgow Haskell compiler

Bogata sintaksa programskega jezika Haskell predstavlja velik izziv za izdelavo prevajalnikov, saj zahteva natančno prevajanje raznolikih sintaktičnih struktur in konstruktov v strojno kodo. Težavo rešuje zadnji korak prednjega dela prevajalnika, imenovan razsladenje (angl. desugarification). V njem se sintaktično drevo jezika Haskell pretvori v drevo jezika Core, ki je minimalističen funkcijski jezik, osnovan na lambda računu. Kljub omejenemu naboru konstruktov omogoča Core zapis katerega koli Haskell programa. Vse nadaljnje faze prevajanja se tako izvajajo na tem precej manjšem jeziku, kar močno poenostavi celoten proces.

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 STG jezik

Kot smo si podrobneje pogledali v poglavju 2.1, lene funkcijske programske jezike najpogosteje implementiramo s pomočjo redukcije grafa. Eden izmed načinov za izvajanje redukcije je abstraktni STG stroj (angl. Spineless Tagless G-machine) [16, 17], 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) [18].

2.3.1 Definicija jezika

Novejši članek [17] pravi: "To make our discussion concrete we use a small, non-strict intermediate language similar to that used inside the Glasgow Haskell Compiler. / ... / In essence it is the STG language, but we have adjusted some of the details for this paper."Kako ustrezno napišemo, da se bomo v našem delu posvetili temu jeziku (in ne dejanskemu jeziku STG [16]), ker je nekoliko bolj preprost za razumevanje in ker nas ne zanimajo podrobnosti implementacije na dejanski računalniški arhitekturi?

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.

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

Kakšen je prevod za boxed? Unboxed? STG jezik podpira dva primitivna (angl. unboxed) podatkovna tipa: celoštevilske vrednosti in števila s plavajočo vejico. Unboxed primitivne tipe označujemo s simbolom lojtre: `n#`. Poleg tega jezik omogoča uvajanje novih algebraičnih podatkovnih tipov, ki jih lahko tvorimo oziroma inicializiramo s pomočjo konstruktorjev C. Pri tem je vredno omeniti, da so algebraični podatkovni tipi definirani v jeziku, ki ga prevajamo v STG, v našem primeru torej v Haskellu. Prav tako se v Haskellu izvaja tudi izpeljava in preverjanje tipov, abstraktni STG stroj pa med samim prevajanjem nima informacij o tipih.

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

Vsi argumenti pri aplikaciji funkcij in primitivnih operacij so v A-normalni obliki (angl. A-normal form) [19], kar pomeni, da so atomarni (angl. atomic). Atomarni ali atomični argumenti? 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.

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

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. V teh primerih funkcije označimo z neznano mestnostjo in jim med izvajanjem posvetimo posebno pozornost. 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	$\coloneqq$	a	atom
	$\mid$	$f^k\ a_1 \ldots a_n$	aplikacija funkcije ($n \geq 1$)
	$\mid$	$\oplus\ a_1 \ldots a_n$	primitivna operacija ($n \geq 1$)
	$\mid$	`let` $x = obj$ `in` e	
	$\mid$	`case` e `of` $\{alt_1; \ldots; alt_n\}$	

Primitivne operacije so funkcije implementirane v izvajalnem okolju (angl. runtime) in so namenjene izvajanju računskih operacij nad unboxed primitivnimi podatki. Jezik podpira celoštevilske operacije `+#`, `-#`, `*#`, `/#`, in operacije nad števili s plavajočo vejico. 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 nasičeno 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 zasičenosti.

Izraz `let` na kopici ustvari nov objekt in je kot tak tudi edini izraz, ki omogoča alokacijo podatkov na pomnilniku. Objekti na kopici bodo po-

drobneje opisani v nadaljevanju. V naši poenostavljeni različici STG jezika, izraz `let` ne omogoča rekurzivnih definicij. Edini vir rekurzije je v jeziku omogočen s pomočjo statičnih definicij (angl. top-level definitions), ki se lahko rekurzivno sklicujejo med sabo oziroma same nase.

alt	$:=$	$C\ x_1 \ldots x_n \to expr$	algebraična alternativa
	$\mid$	$x \to expr$	privzeta alternativa

Izraz `case` je sestavljen iz podizraza e, ki se evalvira in seznama alternativ $alts$, od katerih se vedno izvede *natanko ena*. S preverjanjem tipov v fazi pred prevajanjem v STG je zagotovljeno, da vsi konstruktorji pri alternativah spadajo pod isti algebraični vsotni podatkovni tip (angl. sum type) in da so alternative izčrpne, tj. da obstaja privzeta alternativa ali da algebraične alternative pokrijejo ravno vse možne konstruktorje podatkovnega tipa. Izraz `case` najprej shrani vse žive spremenljivke, ki so uporabljene v izrazih v alternativah, na sklad doda continuation (angl. continuation), tj. naslov kjer se bo izvajanje nadaljevalo in nato začne računati vrednost izraza e. Zaradi bolj učinkovitega izvajanja, se pri prevajanju za konstruktorje vsotnih podatkovnih tipov generirajo oznake, navadno kar nenegativne celoštevilske vrednosti, ki se hranijo v objektu CON. Pri razstavljanju sestavljenega podatkovnega tipa v `case` izrazu lahko tako primerjamo cela števila in ne celih nizov.

obj	$:=$	$\text{FUN}(x_1 \ldots x_n \to e)$	aplikacija
	$\mid$	$\text{PAP}(f\ a_1 \ldots a_n)$	delna aplikacija
	$\mid$	$\text{CON}(C\ a_1 \ldots a_n)$	konstruktor
	$\mid$	$\text{THUNK}\ e$	zakasnitev
	$\mid$	BLACKHOLE	črna luknja

Kot smo že omenili, se novi objekti na kopici tvorijo le s pomočjo `let` izraza. Jezik STG podpira pet različnih vrst objektov, ki se razlikujejo glede na oznako v ovojnici na pomnilniku.

Objekt FUN predstavlja funkcijsko ovojnico (angl. closure) z argumenti $x_1, \ldots, x_n$ 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 $x_1, \ldots, x_n$. Pri tem je zagotovljeno, da bo f objekt tipa FUN, katerega mestnost bo *vsaj* n.

Objekt CON predstavlja nasičeno aplikacijo konstruktorja C na argumente $a1, \ldots a_n$. 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 [20] in neskončnih rekurzivnih struktur. Objekt BLACKHOLE se lahko pojavi le kot rezultat evalvacije zakasnitve, nikoli pa v vezavi v `let` izrazu.

$$program \quad := \quad f_1 = obj_1 \; ; \; \ldots \; ; \; f_n = obj_n$$

Program v jeziku STG je zaporedje vezav, ki priredijo STG objekte v spremenljivke.

2.3.2 Operacijska semantika

Operacijska semantika malih korakov bo opisana s pravili podanimi v obliki, ki jo prikazuje enačba 2.1. Pri tem z e označujemo izraze (iz poglavja 2.3.1), oznaka s predstavlja sklad nadaljevanj (angl. continuation), H pa kopico.

$$e_1;\, s_1;\, H_1 \;\Rightarrow\; e_2;\, s_2;\, H_2 \tag{2.1}$$

Na skladu s hranimo nadaljevanja, ki stroju povedo, kaj storiti, ko bo trenutni izraz e izračunan. Zapis $s = k : s'$ označuje sklad s, kjer je k trenutni element na vrhu sklada, s' pa predstavlja preostanek sklada pod njim. Nadaljevanja so lahko ene izmed naslednjih oblik:

$$\begin{aligned} k \;\; := \;\; & \texttt{case} \bullet \texttt{of}\, \{alt_1; \ldots; alt_n\} \\ \mid \;\; & \mathit{Upd}\, t \bullet \\ \mid \;\; & (\bullet\, a_1 \ldots a_n) \end{aligned}$$

Nadaljevanje $\texttt{case} \bullet \texttt{of}\, \{alt_1; \ldots; alt_n\}$ izvede glede na trenutno vrednost e natanko eno izmed alternativ $alt_1, \ldots, alt_n$. Pri nadaljevanju $\mathit{Upd}\, t \bullet$ se zakasnitev na naslovu t posodobi z vrnjeno vrednostjo, tj. trenutnim izrazom e. Zadnje nadaljevanje $(\bullet\, a_1 \ldots a_n)$ pa uporabi vrnjeno funkcijo nad argumenti $a_1, \ldots, a_n$. Sestavimo ga kadar je pri klicu funkcije argumentov preveč. Če vemo, da funkcija sprejme n argumentov, pri aplikaciji pa je podanih $n + k$ argumentov, bo na sklad dodano nadaljevanje $(\bullet\, a_{n+1} \ldots a_{n+k})$ s k argumenti.

Kopica H je v operacijski semantiki predstavljena kot slovar (angl. map), ki spremenljivke slika v objekte na kopici. Oznaka $H[x]$ predstavlja dostop naslova x na kopici H. Imena spremenljivk se torej v operacijski semantiki kar enačijo s pomnilniškimi naslovi. V dejanski implementaciji je kopica kar zaporeden kos pomnilnika, do katerega dostopamo preko pomnilniških naslovov, naloga prevajalnika pa je, da spremenljivkam dodeli pomnilniške naslove in s tem zagotovi pravilno preslikavo spremenljivk na objekte v pomnilniku. Prav tako je vsak objekt FUN, THUNK, ... na kopici predstavljen kot funkcijska ovojnica, ki hrani naslove oziroma vrednosti prostih spremenljivk, ki se v njem pojavijo.

Omenimo še, da je pri pravilih operacijske semantike vrstni red pomemben, saj se pri izvajanju izvede prvo pravilo, ki ustreza trenutnemu stanju

abstraktnega STG stroja.

Dodeljevanje novih objektov na pomnilniku

Pravilo LET poda operacijsko semantiko `let` izraza. Ta na kopici ustvari nov objekt *obj* in ga priredi novi unikatni spremenljivki x', ki ni uporabljena nikjer v programu, kar ustreza alokaciji še neuporabljenega naslova na kopici. Zapis $e[x'/x]$ predstavlja izraz e, v katerem so vse proste spremenljivke x zamenjane z vrednostjo x'. Pri `let` izrazu se torej najprej alocira nov objekt na pomnilniku, nato pa se izvede telo, v katerem so vse pojavitve spremenljivke x zamenjane z novim pomnilniškim naslovom.

$$\frac{x' \text{ je sveža spremenljivka}}{\texttt{let } x = obj \texttt{ in } e;\ s;\ H \ \Rightarrow\ e[x'/x];\ s;\ H} \quad \text{(LET)}$$

V pravi implementaciji je substitucija $e[x'/x]$ implementirana kot prepisovanje spremenljivke x z kazalcem na objekt x' v klicnem zapisu funkcije.

Pogojna izbira s stavkom `case`

V primeru, da je izraz, ki ga evalviramo v `case` izrazu (angl. scrutinee) pomnilniški naslov, ki kaže na konstruktor C, potem se izvede telo veje s konstruktorjem C, pri katerem se vrednosti parametrov $x_1, \ldots x_n$ zamenjajo z argumenti $a_1, \ldots, a_n$ (pravilo CASECON).

$$\frac{}{\texttt{case } v \texttt{ of } \{\ldots; C\, x_1, \ldots, x_n \to e; \ldots\};\ s;\ H[v \mapsto \text{CON}(C\, a_1 \ldots a_n)] \ \Rightarrow\ e[a_1/x_1 \ldots a_n/x_n];\ s;\ H} \quad \text{(CASECON)}$$

Če se noben izmed konstruktorjev v algebraičnih alternativah ne ujema s konstruktorjem na naslovu v, ali če je v literal, potem se izvede privzeta alternativa (pravilo CASEANY). Pri tem se v telesu alternative parameter x zamenja z v. V STG jeziku so vrednosti objekti FUN, PAP in CON.

$$\frac{(v \text{ je literal}) \vee (H[v] \text{ je vrednost, ki ne ustreza nobeni drugi alternativi})}{\texttt{case}\ v\ \texttt{of}\ \{\ldots; x \to e\};\ s;\ H \ \Rightarrow\ e[v/x];\ s;\ H} \quad (\textsc{caseany})$$

Pravilo CASE začne izračun `case` izraza. Najprej se začne računati izraz e, na sklad pa se potisne nadaljevanje case izraza, ki se bo izvedlo, ko se bo vrednost e do konca izračunala.

$$\frac{}{\texttt{case}\ e\ \texttt{of}\ \{\ldots\};\ s;\ H \ \Rightarrow\ e;\ (\texttt{case}\ \bullet\ \texttt{of}\ \{\ldots\}) : s;\ H} \quad (\textsc{case})$$

Zadnje pravilo RET se izvede, ko se izraz v `case` izrazu do konca izračuna v literal ali kazalec na objekt na kopici. Glede na tip objekta na katerega kaže spremenljivka v, se bo nato izvedlo eno izmed pravil CASECON ali CASEANY.

$$\frac{(v \text{ je literal}) \vee (H[v] \text{ je vrednost})}{v;\ (\texttt{case}\ \bullet\ \texttt{of}\ \{\ldots\}) : s;\ H \ \Rightarrow\ \texttt{case}\ v\ \texttt{of}\ \{\ldots\};\ s;\ H} \quad (\textsc{ret})$$

Pomembno je še omeniti, da se pri izvajanju nikoli ne brišejo vezave objektov na kopici. V primeru pravila CASECON, se tako iz kopice ne izbriše vezava $v \mapsto \textsc{con}(C\, a_1 \ldots a_n)$. Brisanje elementov iz kopice je implementirano s pomočjo avtomatičnega čistilca pomnilnika.

Zakasnitve in njih posodobitve

Pravili THUNK in UPDATE sta namenjeni izračunu zakasnitev. Če je trenutni izraz ravno kazalec na zakasnitev, potem se na sklad doda nov posodobitveni okvir $\mathit{Upd}\ t\ \bullet$, ki kaže na spremenljivko x, ki se bo po izračunu posodobila. Prav tako se na kopici objekt zamenja z BLACKHOLE. Če med izračunom vrednosti izraza e abstraktni stroj ponovno naleti na spremenljivko x, lahko predpostavi, da program vsebuje cikel. Ker nobeno izmed pravil operacijske semantike na levi strani ne vsebuje objekta BLACKHOLE, se v primeru ciklov tako izračun zaustavi.

$$\overline{x;\ s;\ H[x \mapsto \textsc{thunk}(e)] \ \Rightarrow\ e;\ (\mathit{Upd}\, t\ \bullet) : s;\ H[x \mapsto \textsc{blackhole}]} \quad (\textsc{thunk})$$

Pri pravilu UPDATE se izvede posodobitev zakasnitve na kopici, pri čemer se spremenljivko x preusmeri na objekt na katerega kaže spremenljivka y.

$$\frac{H[y] \text{ je vrednost}}{y;\ (\mathit{Upd}\, x\ \bullet) : s;\ H \ \Rightarrow\ y;\ s;\ H[x \mapsto H[y]} \quad (\textsc{update})$$

V dejanski implementaciji se pri posodobitvi na pomnilniku prejšnji objekt prepiše s preusmeritvijo (angl. indirection). To je objekt, ki je (podobno kot ostali objekti v STG jeziku glede na sliko 2.7) sestavljen iz kazalca na metapodatke objekta in še enega dodatnega polja - kazalca na drug objekt. To pa tudi pomeni, da morajo imeti vsi objekti na kopici prostora vsaj za 2 kazalca (2 polji), saj bi sicer pri posodobitvi prišlo do prekoračitve kopice oziroma prepisovanja naslednjega objekta.

Klici funkcij z znano mestnostjo

Pri pravilu KNOWNCALL gre za uporabo funkcije z mestnostjo n nad natanko n argumenti. Izvede se telo funkcije, v katerem so vsi parametri zamenjani z vrednostmi argumentov, sklad in kopica pa ostajata nespremenjena.

$$\frac{}{f^n\, a_1 \ldots a_n;\ s;\ H[f \mapsto \textsc{fun}(x_1 \ldots x_n \to e)] \ \Rightarrow\ e[a_1/x_1 \ldots a_n/x_n];\ s;\ H} \quad (\textsc{knowncall})$$

Pravilo PRIMOP je zelo podobno pravilu KNOWNCALL, le da se vrednost primitivne operacije izračuna neposredno v okolju za izvajanje. Ker so primitivne operacije po definiciji v STG jeziku vedno nasičene, zanje niso potrebna nobena dodatna pravila.

$$\frac{a = \oplus\, a_1 \ldots a_n}{\oplus\, a_1 \ldots a_n;\ s;\ H \ \Rightarrow\ a;\ s;\ H} \quad (\textsc{primop})$$

Pri pravilih KNOWNCALL in PRIMOP gre za aplikacijo funkcije z znano mestnostjo n na natanko n argumentov. Kaj pa če argumentov ni dovolj

ali pa jih je preveč? V primeru, da je argumentov premalo, gre za delno aplikacijo funkcije. Če pa je argumentov preveč, je potrebno funkcijo najprej aplicirati na ustrezno število argumentov. Ko se bo aplikacija do konca izvedla, bo rezultat funkcija, ki jo apliciramo na preostanek argumentov.

Klici funkcij z neznano mestnostjo

Pravilo EXACT je zelo podobno pravilu KNOWNCALL, le ga gre v tem primeru za funkcijo neznane mestnosti. V tem primeru je število argumentov shranjeno v tabeli metapodatkov objekta FUN na kopici, kot bomo videli v poglavju 2.3.3.

$$\frac{}{\begin{array}{c} f^{\bullet}\, a_1 \ldots a_n;\ s;\ H[f \mapsto \textsc{fun}(x_1 \ldots x_n \to e)] \\ \Rightarrow\ \ e[a_1/x_1 \ldots a_n/x_n];\ s;\ H \end{array}} \quad (\textsc{exact})$$

Pravili CALLK in PAP2 se ukvarjata z aplikacijami funkcij, pri katerih je število argumentov različno od njihove mestnosti. Če je število argumentov m več kot parametrov n (pravilo CALLK), potem se prvih n argumentov porabi pri aplikaciji funkcije, vrednosti preostalih argumentov $a_{n+1}, \ldots, a_m$ pa se doda na sklad. Ko se aplikacija do konca izračuna, bo rezultat še ena funkcija, ki je uporabljena na preostalih argumentih (pravilo RETFUN). Če argumentov pri aplikaciji ni dovolj, se na kopici ustvari nov objekt PAP, ki predstavlja delno aplikacijo.

$$\frac{m > n}{\begin{array}{c} f^{k}\, a_1 \ldots a_m;\ s;\ H[f \mapsto \textsc{fun}(x_1 \ldots x_n \to e)] \\ \Rightarrow\ \ e[a_1/x_1 \ldots a_n/x_n];\ (\bullet\ a_{n+1} \ldots a_m) : s;\ H \end{array}} \quad (\textsc{callk})$$

$$\frac{m < n, p \text{ je sveža spremenljivka}}{\begin{array}{c} f^{k}\, a_1 \ldots a_m;\ s;\ H[f \mapsto \textsc{fun}(x_1 \ldots x_n \to e)] \\ \Rightarrow\ \ p;\ s;\ H[p \mapsto \textsc{pap}(f\, a_1 \ldots a_m)] \end{array}} \quad (\textsc{pap2})$$

Pravilo TCALL pokrije možnost, da funkcija še ni izračunana, torej da je zakasnitev. V tem primeru se prične računanje zakasnitve f, obenem pa se

na sklad doda nadaljevanje $(\bullet\ a_1 \ldots a_m)$, ki bo sprožilo klic funkcije, ko bo vrednost zakasnitve do konca izračunana.

$$\frac{}{f^\bullet\ a_1 \ldots a_m;\ s;\ H[f \mapsto \textsc{thunk}(e)] \quad \Rightarrow \quad f;\ (\bullet\ a_1 \ldots a_m) : s;\ H} \qquad (\textsc{tcall})$$

Pri pravilu PCALL gre za uporabo delne aplikacije nad preostankom argumentov. V tem primeru se funkcija g uporabi nad vsemi m argumenti. Ker je g zaradi definicije v poglavju 2.3.1 zagotovo funkcija, se bo v naslednjem koraku zagotovo zgodilo eno izmed pravil EXACT, CALLK ali PAP2.

$$\frac{}{f^k\ a_{n+1} \ldots a_m;\ s;\ H[f \mapsto \textsc{pap}(g\ a_1 \ldots a_n)] \quad \Rightarrow \quad g^\bullet\ a_1 \ldots a_n\ a_{n+1} \ldots a_m;\ s;\ H} \qquad (\textsc{pcall})$$

Zadnje pravilo RETFUN, ki deluje na podoben način kot pravilo RET pri `case` izrazih. Kadar se izraz evalvira v kazalec, ki kaže na funkcijo FUN ali delno aplikacijo PAP in je na skladu nadaljevanje z argumenti, se ustrezno izvede aplikacija funkcije nad argumenti.

$$\frac{(H[f] \text{ je } \textsc{fun}(\ldots)) \vee (H[f] \text{ je } \textsc{pap}(\ldots))}{f;\ (\bullet\ a_1 \ldots a_n) : s;\ H \quad \Rightarrow \quad f^\bullet\ a_1 \ldots a_n;\ s;\ H} \qquad (\textsc{retfun})$$

V nadaljevanju bomo na kratko opisali delovanje abstraktnega STG stroja in predstavitev objektov na pomnilniku.

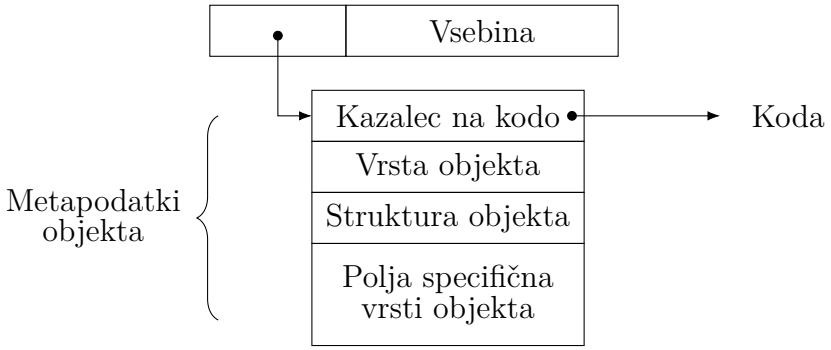
2.3.3 Abstraktni STG stroj

To poglavje ni dokončano, ker nisem bil prepričan, ali sodi v magistrsko delo.

Slika 2.7 prikazuje strukturo STG objektov na pomnilniku. Vsak objekt je sestavljen iz kazalca na tabelo metapodatkov in vsebine objekta, ki je sestavljena iz kazalcev na druge objekte ali literalov (angl. literals), tj. celih števil oziroma števil s plavajočo vejico. Različni objekti nosijo različno vsebino. Objekt $\textsc{con}(C\ a_1\ \ldots\ a_n)$ je npr. predstavljen s strukturo, ki ima na prvem mestu kazalec do metapodatkov objekta tipa CON, vsebina pa ima

n polj, z vrednostmi argumentov $a_1, \ldots, a_n$ (ki so, kot smo videli v poglavju 2.3.1 atomarne vrednosti, tj. kazalci ali literali).

Tabela metapodatkov vsebuje dodatne informacije, ki jih potrebujemo pri izvajanju programa. Sestavlja jo kazalec na kodo, vrsta objekta, struktura objekta in dodatna polja, ki so specifična vrsti objekta. Kazalec na kodo kaže na kodo, ki se izvede pri evalvaciji objekta. Vrsta objekta je oznaka (npr. kar celoštevilska vrednost), ki razlikuje pet različnih vrst objektov (THUNK, FUN, PAP, CON, BLACKHOLE) med seboj. Struktura objekta vsebuje podatke o tem katera polja predstavljajo kazalce in katera polja dejanske vrednosti oziroma literale. V STG stroju se uporablja za avtomatično čiščenje pomnilnika, saj je pri tem v prvem koraku potrebno ugotoviti kateri objekti so živi, to pa sistem stori tako, da preko kazalcev obišče vse objekte. Vsak objekt vsebuje še dodatna polja, ki pa so odvisna od vrste objekta. Tako funkcija FUN npr. vsebuje njeno mestnost, medtem ko hrani konstruktor FUN njegovo oznako.



Slika 2.7: Struktura STG objektov na pomnilniku

- Model **potisni / vstopi** (angl. push / enter) najprej potisne vse argumente na sklad in nato *vstopi* v funkcijo. Funkcija je odgovorna za preverjanje ali je na skladu dovolj argumentov. Če jih ni, potem mora sestaviti delno aplikacijo na kopici in končati izvajanje. Če pa je argumentov preveč, jih mora funkcija s sklada vzeti le ustrezno število, ostale argumente pa pustiti, saj jih bo uporabila naslednja funkcija, tj. funkcija v katero se bo evalviral trenutni izraz.

- Model **izračunaj / uporabi** (angl. eval / apply) klicatelj najprej evalvira funkcijo in jo nato aplicira na ustrezno število argumentov. Pri aplikaciji je potrebno zahtevano število argumentov funkcije pridobiti med izvajanjem programa iz funkcijske ovojnice.

Modela se razlikujeta glede na prelaganje odgovornosti za ... Pri modelu potisni / vstopi mora klicana funkcija preveriti ali je na skladu dovolj argumentov. Pri modelu izračunaj in apliciraj pa mora klicoča funkcija med izvajanjem pogledati v funkcijsko ovojnico in jo poklicati z ustreznim številom argumentov.

Izkaže se, da je hitrejši model izračunaj / apliciraj [17], zato je ta model tudi uporabljen v Haskellovem prevajalniku GHC.

2.3.4 Primer

Manjka primer izvajanja manjšega programa v STG jeziku.

Poglavje 3

Sorodno delo

Da je upravljanje s pomnilnikom v funkcijskih programskih jezikih še vedno aktualno področje raziskovanja, priča mnogo eksperimentalnih jezikov razvitih v zadnjih dvajsetih letih. V sledečem poglavju se bomo podrobneje posvetili sistemom tipov

3.1 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 [34] je *strong* funkcijski jeizk, ki v svojem sistemu tipov združuje tako linearne, kot tudi (angl. graded modal) tipe. Za čiščenje pomnilnika se uporablja avtomatski čistilec.

V članku [35] 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.

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] G. Van Rossum, et al., Python programming language., in: USENIX annual technical conference, Vol. 41, Santa Clara, CA, 2007, pp. 1–36.

[6] S. Klabnik, C. Nichols, The Rust Programming Language, 2nd Edition, No Starch Press, 2023.

[7] R. Jung, Understanding and evolving the Rust programming language, Ph.D. thesis, Saarland University (2020).

[8] N. Matsakis (2018). [link].
URL `https://smallcultfollowing.com/babysteps/blog/2018/04/27/an-alias-based-formulation-of-the-borrow-checker/`

[9] 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

[10] A. Weiss, O. Gierczak, D. Patterson, A. Ahmed, Oxide: The essence of Rust, arXiv preprint arXiv:1903.00982 (2019).

[11] 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.

[12] 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).

[13] 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

[14] P. Hudak, Conception, evolution, and application of functional programming languages, ACM Computing Surveys 21 (3) (1989) 359–411.

[15] S. L. Peyton Jones, The implementation of functional programming languages (prentice-hall international series in computer science), Prentice-Hall, Inc., 1987.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] R. Jones, Tail recursion without space leaks, Journal of Functional Programming 2 (1) (1992) 73–79.

[21] M. Scott, Programming language pragmatics, 4th Edition, Morgan Kaufmann, 2016.

[22] 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.

[23] 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.

[24] E. Czaplicki, Elm : Concurrent FRP for functional GUIs, Senior thesis, Harvard University 30 (2012).

[25] 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.

[26] D. Syme, The f# compiler technical overview (3 2017).

[27] 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.

[28] U. Boquist, T. Johnsson, The GRIN project: A highly optimising back end for lazy functional languages, in: Implementation of Functional Languages: 8th International Workshop, IFL'96 Bad Godesberg, Germany, September 16–18, 1996 Selected Papers 8, Springer, 1997, pp. 58–84.

[29] P. Podlovics, C. Hruska, A. Pénzes, A modern look at GRIN, an optimizing functional language back end, Acta Cybernetica 25 (4) (2022) 847–876.

[30] U. Boquist, Code optimization techniques for lazy functional languages, Doktorsavhandlingar vid Chalmers Tekniska Hogskola (1495) (1999) 1–331.

[31] T. Kocjan Turk, Len funkcijski programski jezik brez čistilca pomnilnika, Master's thesis, Univerza v Ljubljani (2022).

[32] N. Corbyn, Practical static memory management, Tech. rep., University of Cambridge, BA Dissertation (2020).

[33] R. L. Proust, ASAP: As static as possible memory management, Tech. rep., University of Cambridge, Computer Laboratory (2017).

[34] 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.

[35] 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