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

Povzetek

Abstract

1 Uvod **1**

1.1 Upravljanje s pomnilnikom 1

1.2 Rustov model lastništva 3

2 STG **7**

2.1 Leni izračun 8

2.2 Prevajalnik GHC 13

2.3 STG jezik 15

3 Sorodno delo **19**

3.1 Linearen sistem tipov 21

4 Zaključek **23**

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

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 upravlja-

nje 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. 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.

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 v spremenljivke posodobiti še števce 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 z 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.

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 [5], 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 [6]. 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

1.2.1 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.

```
let number = Complex(0, 1);
let a = number;
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.

```
let number = Complex(1, 0);
f(number);
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.

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.

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 ✓

V naslednjem primeru skušamo ustvariti dve mutable referenci na spremenljivko `number`. Zaradi principa izključevanja to ni mogoče, zato prevajalnik 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 ni veljavno ustvariti mutabilne reference na spremenljivko, dokler nanjo obstaja kakršnakoli druga referenca.

```
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 ✗

1.2.3 Delna izposoja

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

Rust ✓

1.2.4 Ponovna izposoja

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

Rust ✓

V trenutni različici Rusta 2018 bi se vsi zgornji primeri vseeno uspešno prevedli. Projekt Polonius [7, 8] je v programski jezik namreč uvedel neleksikalne življenjske dobe (angl. non-lexical lifetimes), ki preverjanje izposoj in premikov izvaja nad CFG namesto nad abstraktnim sintaktičnim drevesom programa [9, 10]. Prevajalnik zna s pomočjo neleksikalnih življenskih 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. Za vse zgornje primere torej predpostavljamo, da tako premaknjene, kot tudi izposojene spremenljivke spremenljivke v nadaljevanju še nekje uporabljene.

Kljub temu, da Rust v svoji dokumentaciji [5] 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 [11] formalizira delovanje analize izposoj v Rustu in dokaže, da velja izrek o varnosti (angl. soundness) za safe podmnožico jezika Rust. Model RustBelt [12] poda prvi formalen in strojno preverjen dokaz o varnosti za jezik, ki predstavlja realistično podmnožico jezika Rust. Model Stacked borrows [10] definira operacijsko semantiko za dostopanje do pomnilnika v Rustu in model za prekrivanje (angl. aliasing model) in ga strojno dokaže. Model Oxide [9] 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 producirali 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 drugi funkcij. Čist funkcijski program lahko tako opišemo kot kompozicijo čistih funkcij, ki vhod preslikajo v izhod.

Poleg čistih funkcij je ena izmed ključnih značilnosti mnogih 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 uprablja 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 evalvira 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 [TODO] (angl. call by value). Obratno, je leni izračun (angl. lazy evaluation) način implementacije nestroge semantike. Pri lenem izračunu izrazi niso evalvirani 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.

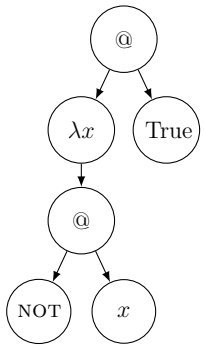
Redukcija grafa

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.

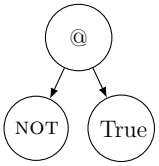
Leni izračun najpogosteje implementiramo s pomočjo redukcije grafa [13, 14]. Pri tej metodi prevajalnik na pomnilniku najprej sestavi abstraktno sintaksno drevo programa, kjer vozlišča predstavljajo aplikacije funkcij oziroma operacije nad podatki, njihova podvozlišča pa odvisnosti med njimi. V procesu *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

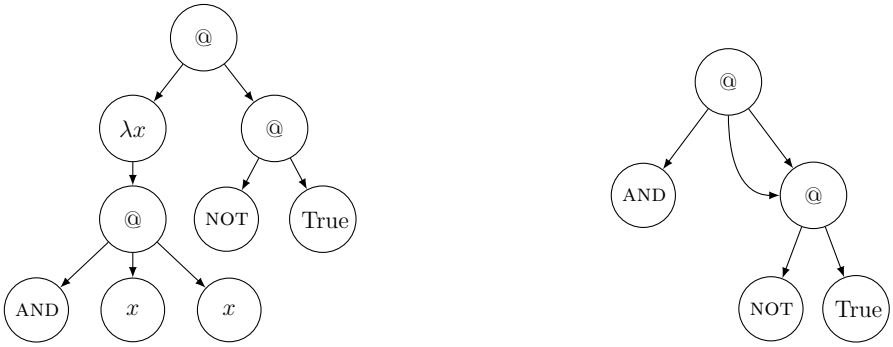


(b) Abstraktno sintaksno drevo *po* redukciji

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

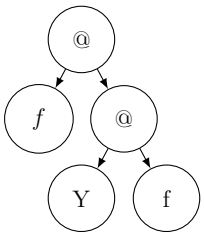
Slika 2.2 prikazuje en korak redukcije abstraktnega sintaksnega drevesa izraza $(\lambda x\,.\,\text{AND}\ x\ x)$ (NOT `True`). Po enem koraku redukcije sintaksnega telesa, se *vse uporabe* parametra x zamenjajo z njegovo vrednostjo. Ker je takih pojavitev več, pa rezultat ni več drevo, temveč acikličen 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 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,

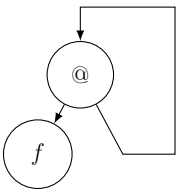


Slika 2.2: Redukcija grafa izraza $(\lambda x\,.\,\text{AND}\ x\ x)\ (\text{NOT}\ \texttt{True})$

kjer je vrednost argumenta kar vozlišče samo.



(a) Funkcija Y implementirana z uporabo proste spremenljivke



(b) Funkcija Y implementirana kot cikličen usmerjen graf

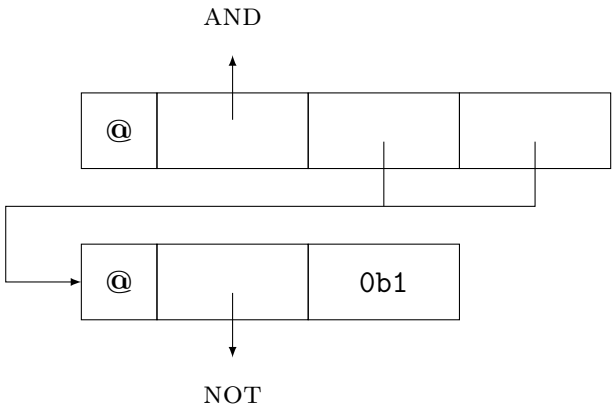
Slika 2.3: Graf funkcije $Y\ f = f\ (Y\ f)$

Zapis grafa na pomnilniku

Oznaka	a_1	a_2	...	a_n

Slika 2.4: Pomnilniška predstavitev vozlišča grafa

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 vrendost, izračunano vrednost pa se shrani v struktro na pomnilniku, da je ob naslednji evalvaciji ni potrebno ponovno računati. Na tak način se z nestrogo



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

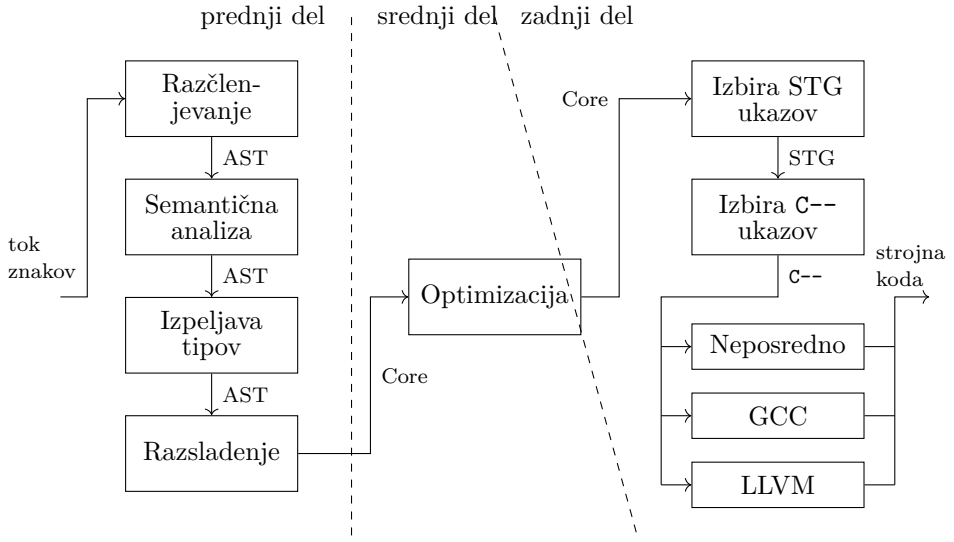
semantiko doseže, da se vsak izraz izračuna *največ enkrat*. Če se argument ne pojavi nikjer v telesu funkcije, se zakasnitve nikoli ne računa, če pa se v telesu pojavi večkrat, se vrednost izračuna enkrat, za vsako nadaljno evalvacijo argumenta pa se preprosto vrne vrednost shranjeno na pomnilniku.

V nadeljevanju 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 (angl. Glasgow Haskell compiler) 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 redukcje grafa. Eden izmed načinov za izvajanje redukcije je abstraktni STG stroj (angl. Spineless Tagless G-machine) [15, 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) [16].

Definicija jezika

V sledečem poglavju bomo podali formalno definicijo jezika STG definirano v [17]. Od originalne implementacije v [15] 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.

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

Vsi argumenti pri aplikaciji funkcij in primitivnih operacij so v A-normalni obliki (angl. A-normal form) [18], 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.

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

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. 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
	$\mid$	$f^k a_1 \ldots a_n$	klic 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` $\{alt_1; \ldots; alt_n\}$	

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	$\coloneqq$	$C\ x_1 \ldots x_n \rightarrow expr$	algebraična alternativa
	$\mid$	$x \rightarrow expr$	privzeta alternativa

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

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

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 saturirano 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 [19] in neskončnih rekurzivnih struktur. Objekt BLACKHOLE se lahko pojavi le kot rezultat evalvacije zakasnitve, nikoli pa v vezavi v `let` izrazu.

Abstraktni STG stroj

Poglavje 3

Sorodno delo

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 [14], 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) [20], 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) [13], 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

dobo, zato je nujna učinkovita implementacija dodeljevanja in sproščanja pomnilnika. Haskell za to uporablja *generacijski* avtomatični čistilec pomnilnika [21, 16]. Danes vsi večji funkcijski programski jeziki, ki omogočajo leni izračun, uporabljajo avtomatični čistilec pomnilnika [22, 23, 24, 25, 26].

Lene funkcijske programske jezike najpogosteje implementiramo s pomočjo redukcije grafa [13]. Eden izmed načinov za izvajanje redukcije je abstraktni STG stroj (angl. Spineless Tagless G-machine) [15], 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) [16].

Ena izmed alternativ STG stroja za izvajanje jezikov z nestrogo semantiko je prevajalnik GRIN [27] (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) [28]. Za upravljanje s pomnilnikom se v trenutni različici uporablja čistilec pomnilnika [29].

Programski jezik Rust za upravljanje s pomnilnikom uvede princip lastništva (angl. ownership model) [5], pri katerem ima vsak objekt na kopici natančno enega *lastnika* [6, 9, 10]. Kadar gre spremenljivka, ki si lasti objekt, izven dosega (angl. out of scope), se pomnilnik za objekt sprosti. Rust definira pojem *premika* (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 [30]. 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 [31] je programski jezik, podoben Rustu, ki za upravljanje s pomnilnikom uporablja princip ASAP (angl. As Static As Possible) [32]. 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.

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 [33] 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 [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 4

Zaključek

Pri magistrskem delu nam je uspelo implementirati izvajalno okolje za STG jezik.

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

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

[8] 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

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

[10] 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.

[11] 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).

[12] 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

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

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

[15] 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.

[16] 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.

[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] 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.

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

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

[21] 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.

[22] 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.

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

[24] 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.

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

[26] 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.

[27] 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.

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

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

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

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

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

[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