

UNIVERZA V LJUBLJANI  
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Niki Bizjak

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

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

MENTOR: doc. dr. Boštjan Slivnik

Ljubljana, 2023



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



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



## ZAHVALA

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

*Niki Bizjak, 2023*



Vsem rožicam tega sveta.

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

— Albert Einstein





# Kazalo

**Povzetek**

**Abstract**

<b>1</b>	<b>Uvod</b>	<b>1</b>
1.1	Upravljanje s pomnilnikom . . . . .	1
1.2	Rustov model lastništva . . . . .	3
1.2.1	Premik . . . . .	4
1.2.2	Izposoja . . . . .	5
<b>2</b>	<b>STG</b>	<b>11</b>
2.1	Leni izračun . . . . .	12
2.2	Prevajalnik GHC . . . . .	18
2.3	STG jezik . . . . .	20
2.3.1	Definicija jezika . . . . .	20
2.3.2	Operacijska semantika . . . . .	24
2.3.3	Abstraktni STG stroj . . . . .	30
2.3.4	Primer . . . . .	32
<b>3</b>	<b>Sorodno delo</b>	<b>33</b>
3.1	Sistemi tipov . . . . .	34
3.2	Linearni tipi . . . . .	36
3.3	Unikatni tipi . . . . .	39
3.4	Programski jezik Granule . . . . .	41

## KAZALO

<b>4</b>	<b>Življenjske dobe in leni izračun</b>	<b>45</b>
4.1	Izpeljava življenjskih dob . . . . .	45
4.2	Primer . . . . .	47
<b>5</b>	<b>Zaključek</b>	<b>49</b>

# Seznam uporabljenih kratic

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



# Povzetek

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

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

## Ključne besede

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



# Abstract

**Title:** Ownership model instead of garbage collection during lazy evaluation

This sample document presents an approach to typesetting your BSc thesis using L<sup>A</sup>T<sub>E</sub>X. 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 števcov, 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 posodablajo metapodatkov na pomnilniku ob vsaki spremembi, temveč se izvedejo le, kadar se

prekorači velikost kopice. Algoritem pregleda kopico in ugotovi, na katere objekte ne kaže več noben kazalec ter jih odstrani. Nekateri algoritmi podatke na kopici tudi defragmentirajo in s tem zagotovijo boljšo lokalnost ter s tem boljše predpomnjenje [4]. Problem metode označi in pometi pa je njeno nedeterministično izvajanje, saj programer oziroma sistem ne moreta predvideti, kdaj se bo izvajanje glavnega programa zaustavilo, in tako ni primerna za časovno-kritične (angl. real-time) aplikacije. 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 sprostí, 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 (angl. exclusion principle) [7]. V poljubnem trenutku za neko vrednost na pomnilniku velja natanko ena izmed dveh možnosti:

- Vrednost lahko *spreminjamo* 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.

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

Rust ✗

Spremenljivka `a` prevzame lastništvo nad vrednostjo 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 spremenljivo ali da so vse izposoje nespremenljive.

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

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

Rust ✓

Zaradi principa izključitve je v Rustu prepovedano ustvariti več kot eno spremenljivo referenco na objekt. V primeru, da bi bilo dovoljeno ustvariti več spremenljivih 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 spremenljivi referenci, zaradi česar prevajalnik ustrezno javi napako.

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

Rust ✗

Prav tako v Rustu ni veljavno ustvariti spremenljive 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.

```
1 let mut number = Complex(0, 0);
2 let a = &number;
3 let b = &mut number; // Napaka: cannot borrow `number` as mutable
4                       // 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 spremenljiva referenca ali da so vse reference nanjo nespremenljive. 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`.

```
1 let outer;
2 {
3     let inner = 3;
4     outer = &inner; // Napaka: `inner` does not live
5                     // long enough
6 }
```

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 spremenljivi 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 nekeje 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 curryjinga

oziroma delne aplikacije [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.

```
1  konjunkcija :: Bool -> Bool -> Bool
2  konjunkcija p q =
3      case p of
4          False -> False
5          True  -> q
```

Haskell

Kot primer si oglejmo razliko pri evalvaciji 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 potrebvana. 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  $\text{eval}(\text{expr}) = \perp$ . 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.

Ena izmed slabosti neučakanega 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 neučakanega 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 memoizirana v vseh preostalih uporabah. Slabost lenega izračuna pa je v tem, da ga je težje implementirati in da se navadno izvaja počasneje od jezikov z neučakanim izračunom.

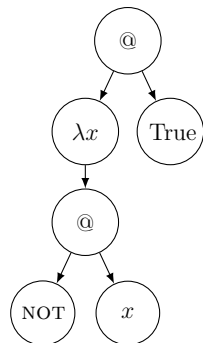
## Redukcija grafa

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.

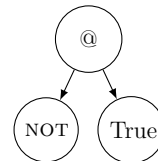
Leni izračun najpogosteje implementiramo s pomočjo redukcije grafa [15, 14]. Pri tej metodi prevajalnik na pomnilniku najprej sestavi abstraktno sintaksno drevo programa, kjer vozlišča predstavljajo aplikacije funkcij oziroma operacije nad podatki, njihova podvozlišča pa odvisnosti med njimi. V procesu *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 sintaksno 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.



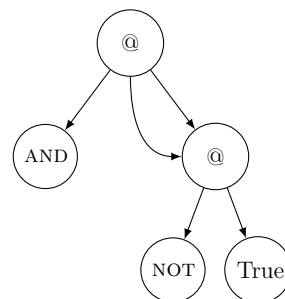
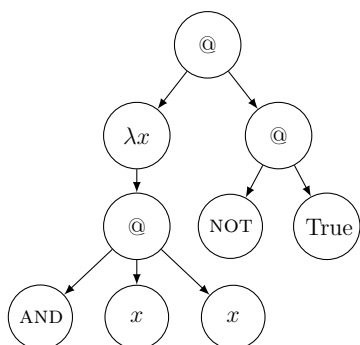
(a) Abstraktno sintaksno drevo *pred* redukcijo



(b) Abstraktno sintaksno drevo *po* redukciji

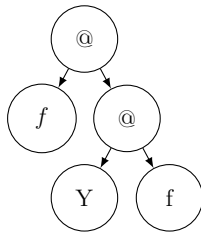
**Slika 2.1:** Redukcija grafa izraza  $(\lambda x . \text{NOT } x) \text{ True}$

Slika 2.2 prikazuje en korak redukcije abstraktnega sintaksnega drevesa izraza  $(\lambda x . \text{AND } x x) (\text{NOT True})$ . 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.

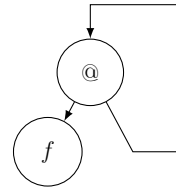


**Slika 2.2:** Redukcija grafa izraza  $(\lambda x . \text{AND } x x) (\text{NOT True})$

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 aciklični 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 pomnilniškim ciklom, kjer je vrednost argumenta kar vozlišče samo.



(a) Funkcija  $Y$  implementirana z uporabo proste spremenljivke



(b) Funkcija  $Y$  implementirana kot ciklični 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 spremenljivke. 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.

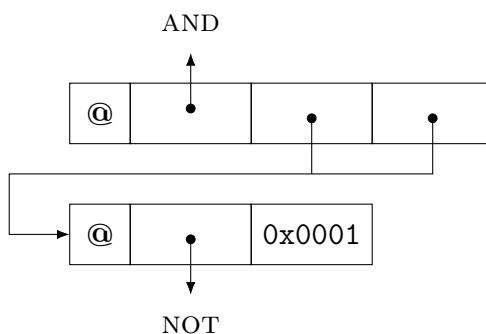


Slika 2.4 prikazuje eno izmed možnih predstavitev vozlišča grafa. Sestavljena je iz oznake vozlišča in polj z vsebino oziroma argumenti  $a_1, \dots, a_n$ . V poglavju 2.3 bomo videli, da sta si struktura 2.4 in pomnilniški zapis objektov v jeziku STG precej podobna.

Oznaka	$a_1$	$a_2$	$\dots$	$a_n$
--------	-------	-------	---------	-------

**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.



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

Ena izmed slabosti jezikov z lenim izračunom je, da je, za razliko od neučakanih 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

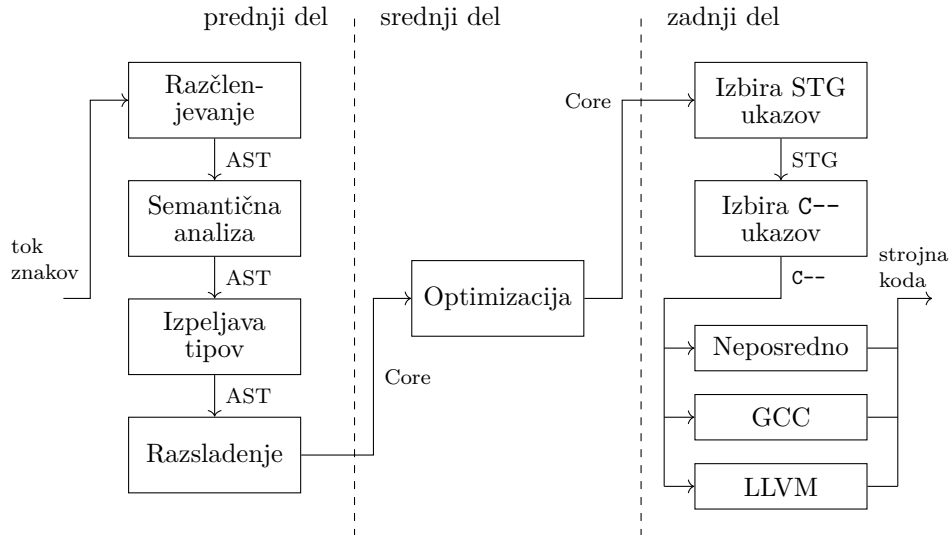
ustvarila vrednost argumenta ali rezultata. Zato klicnih zapisov takih funkcij ni mogoče hraniti na skladu, temveč na kopici [1]. Kot smo že lahko videli v poglavju 2.1, so zakasnitve in funkcije na kopici shranjene kot ovojnice, tj. podatkovne strukture, v katerih se poleg kode hranijo še kazalci na 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 [16, 17]. Danes vsi večji funkcijski programski jeziki, ki omogočajo leni izračun, uporabljajo avtomatični čistilec pomnilnika [18, 19, 20, 21, 22].

V nadaljevanju se bomo osredotočili na delovanje prevajalnika GHC (angl. Glasgow Haskell compiler), ki izvirno kodo napisano v programskem jeziku Haskell prevede v strojno kodo. Pri prevajanju se program transformira v več različnih vmesnih predstavitev (angl. intermediate representation), mi pa se bomo v magistrskem delu osredotočili predvsem na vmesno kodo imenovano STG jezik (angl. Spineless tagless G-Machine language), katerega delovanje bomo podrobneje opisali v razdelku 2.3.

## 2.2 Prevajalnik GHC

Prevajalnik GHC prevajanje iz izvirne kodo v programskem jeziku Haskell v strojno kodo izvaja v več zaporednih fazah oziroma modulih. Vsaka faza kot vhod prejme izhod prejšnje, nad njim izvede določeno transformacijo in rezultat posreduje naslednji fazi. Faze glede na njihovo funkcijo v grobem delimo na tri dele. V prednjem delu (angl. front-end) se nad izvirno kodo najprej izvede leksikalna analiza, pri kateri se iz toka znakov, ki predstavljajo vhodni program pridobi abstraktno sintaksno 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 podmnžica 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) [23, 24], 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) [17].

### 2.3.1 Definicija jezika

Novejši članek [24] 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 [23]), 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  $C$ .

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

STG jezik podpira dva neokvirjena (angl. unboxed) podatkovna tipa: celoštevilске vrednosti in števila s plavajočo vejico. Neokvirjene primitivne

tipe označujemo s simbolom **lojtre**:  $n\#$ . Poleg tega jezik omogoča uvajanje novih algebrskih podatkovnih tipov, ki jih lahko tvorimo oziroma inicializiramo s pomočjo konstruktorjev  $C$ . Pri tem je vredno omeniti, da so algebrski 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.

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

Vsi argumenti pri aplikaciji funkcij in primitivnih operacij so v A-normalni obliki (angl. A-normal form) [25], kar pomeni, da so **atomarni** (angl. atomic). Tako je vsak argument ali primitivni podatkovni tip ali pa spremenljivka. Pri prevajanju v STG jezik lahko prevajalnik sestavljene argumente funkcij priredi novim spremenljivkam z ovijanjem v `let` izraz in spremenljivke uporabi kot argumente pri klicu funkcije. Pri tem je potrebno zagotoviti, da so definirane spremenljivke unikatne oziroma, da se ne pojavijo v ovitem izrazu. Aplikacijo funkcije  $f (\oplus x y)$  bi tako ovili v `let` izraz `let a =  $\oplus x y$  in f a`, s čemer bi zagotovili, da so vsi argumenti atomarni.

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

Prevajalnik lahko med prevajanjem za določene funkcije določi njihovo mestnost (angl. arity), tj. število argumentov, ki jih funkcija sprejme. Ker pa je STG funkcijski jezik, lahko funkcije nastopajo tudi kot argumenti drugih funkcij, zato včasih določevanje mestnosti ni mogoče. 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.

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

Primitivne operacije so funkcije implementirane v izvajalnem okolju (angl. runtime) in so namenjene izvajanju računskih operacij nad neokvirjenimi primitivnimi podatki. Jezik podpira celoštevilске 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  $\eta$ -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 podrobneje 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.

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

Izraz **case** je sestavljen iz podizraza  $e$ , ki se izračuna 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 kontinuiracijo (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 celoštevilске vrednosti, ki se hranijo v objektu CON. Pri razstavljanju sestavljenega podatkovnega tipa v **case** izrazu lahko tako primerjamo cela števila in ne celih nizov.

$obj \quad := \quad \text{FUN}(x_1 \dots x_n \rightarrow e)$	aplikacija
$\quad \quad \quad   \quad \text{PAP}(f \ a_1 \dots a_n)$	delna aplikacija
$\quad \quad \quad   \quad \text{CON}(C \ a_1 \dots a_n)$	konstruktor
$\quad \quad \quad   \quad \text{THUNK } e$	zakasnitev
$\quad \quad \quad   \quad \text{BLACKHOLE}$	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 ovojnicu na pomnilniku.

Objekt FUN predstavlja funkcijsko ovojnicu (angl. closure) z argumenti  $x_1, \dots, x_n$  in telesom  $e$ , ki pa se lahko poleg argumentov  $x_i$  sklicuje še na

druge proste spremenljivke. Pri tem velja, da je lahko funkcija aplicirana na več kot  $n$  ali manj kot  $n$  argumentov, tj. je curryrana.

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

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

Objekt THUNK predstavlja zakasnitev izraza  $e$ . Kadar se vrednost izraza uporabi, tj. kadar se izvede **case** izraz, se izračuna vrednost  $e$ , THUNK objekt na kopici pa se nato posodobi s preusmeritvijo (angl. indirection) na vrednost  $e$ . Pri izračunu zakasnitve se objekt THUNK na kopici zamenja z objektom BLACKHOLE, s čemer se preprečuje puščanje pomnilnika [26] in neskončnih rekurzivnih struktur. Objekt BLACKHOLE se lahko pojavi le kot rezultat evalvacije zakasnitve, nikoli pa v vezavi v **let** izrazu.

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

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 kontinuiranj,  $H$  pa kopico.

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

Na skladu  $s$  hranimo kontinuiranje, 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. Kontinuiranje so lahko ene izmed naslednjih oblik:



$$\begin{aligned}
k &:= \text{case } \bullet \text{ of } \{alt_1; \dots; alt_n\} \\
&| \quad Upd\ t\ \bullet \\
&| \quad (\bullet\ a_1 \dots a_n)
\end{aligned}$$

Kontinuuacija  $\text{case } \bullet \text{ of } \{alt_1; \dots; alt_n\}$  izvede glede na trenutno vrednost  $e$  natanko eno izmed alternativ  $alt_1, \dots, alt_n$ . Pri kontinuuaciji  $Upd\ t\ \bullet$  se zakasnitev na naslovu  $t$  posodobi z vrnjeno vrednostjo, tj. trenutnim izrazom  $e$ . Zadnja kontinuuacija  $(\bullet\ a_1 \dots a_n)$  pa uporabi vrnjeno funkcijo nad argumenti  $a_1, \dots, a_n$ . Sestavimo jo, 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 dodana kontinuuacija  $(\bullet\ a_{n+1} \dots a_{n+k})$  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 funkcij-ska 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.

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

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

### Pogojna izbira s stavkom **case**

V primeru, da je izraz, ki ga računamo 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  $x_1, \dots, x_n$  zamenjajo z argumenti  $a_1, \dots, a_n$  (pravilo CASECON).

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

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.

$$\frac{(v \text{ je literal}) \vee (H[v] \text{ je vrednost, ki ne ustreza nobeni drugi alternativni})}{\text{case } v \text{ of } \{ \dots; x \rightarrow e \}; s; H \Rightarrow e[v/x]; s; H} \quad (\text{CASEANY})$$

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

$$\frac{}{\text{case } e \text{ of } \{ \dots \}; s; H \Rightarrow e; (\text{case } \bullet \text{ of } \{ \dots \}) : s; H} \quad (\text{CASE})$$

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.

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

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  $v \mapsto \text{CON}(C a_1 \dots a_n)$ . 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 nova kontinuacija oziroma posodobitveni okvir  $\text{Upd } t \bullet$ , 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.

$$\frac{x; s; H[x \mapsto \text{THUNK}(e)] \Rightarrow e; (\text{Upd } t \bullet) : s; H[x \mapsto \text{BLACKHOLE}]}{\quad} \quad (\text{THUNK})$$

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

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

V dejanski implementaciji se pri posodobitvi na pomnilniku prejšnji objekt prepíše s preusmeritvijo. 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.

$$\frac{f^n a_1 \dots a_n; s; H[f \mapsto \text{FUN}(x_1 \dots x_n \rightarrow e)]}{\Rightarrow e[a_1/x_1 \dots a_n/x_n]; s; H} \quad (\text{KNOWNCALL})$$

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.

$$\frac{a = \oplus a_1 \dots a_n}{\oplus a_1 \dots a_n; s; H \Rightarrow a; s; H} \quad (\text{PRIMOP})$$

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.

$$\frac{f^\bullet a_1 \dots a_n; s; H[f \mapsto \text{FUN}(x_1 \dots x_n \rightarrow e)]}{\Rightarrow e[a_1/x_1 \dots a_n/x_n]; s; H} \quad (\text{EXACT})$$

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  $a_{n+1}, \dots, a_m$  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.

$$\frac{m > n}{f^k a_1 \dots a_m; s; H[f \mapsto \text{FUN}(x_1 \dots x_n \rightarrow e)]} \quad (\text{CALLK})$$

$$\Rightarrow e[a_1/x_1 \dots a_n/x_n]; (\bullet a_{n+1} \dots a_m) : s; H$$

$$\frac{m < n, p \text{ je sveža spremenljivka}}{f^k a_1 \dots a_m; s; H[f \mapsto \text{FUN}(x_1 \dots x_n \rightarrow e)]} \quad (\text{PAP2})$$

$$\Rightarrow p; s; H[p \mapsto \text{PAP}(f a_1 \dots a_m)]$$

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 kontinuiracija  $(\bullet a_1 \dots a_m)$ , ki bo sprožila klic funkcije, ko bo vrednost zakasnitve do konca izračunana.

$$\frac{f^\bullet a_1 \dots a_m; s; H[f \mapsto \text{THUNK}(e)]}{\Rightarrow f; (\bullet a_1 \dots a_m) : s; H} \quad (\text{TCALL})$$

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.

$$\frac{f^k a_{n+1} \dots a_m; s; H[f \mapsto \text{PAP}(g a_1 \dots a_n)]}{\Rightarrow g^\bullet a_1 \dots a_n a_{n+1} \dots a_m; s; H} \quad (\text{PCALL})$$

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

$$\frac{(H[f] \text{ je FUN}(\dots)) \vee (H[f] \text{ je PAP}(\dots))}{f; (\bullet a_1 \dots a_n) : s; H \Rightarrow f^\bullet a_1 \dots a_n; s; H} \quad (\text{RETFUN})$$

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

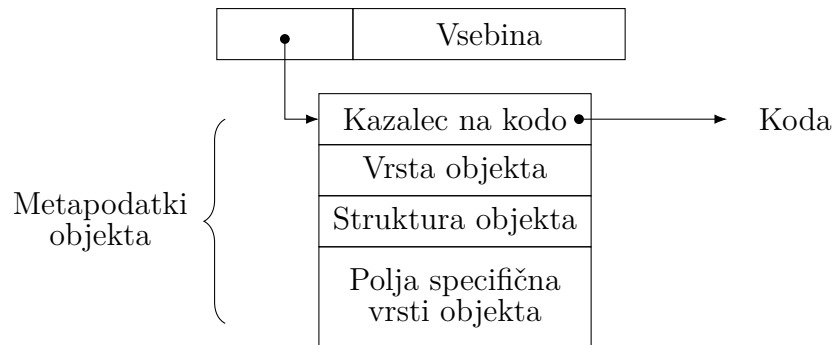
### 2.3.3 Abstraktni STG stroj

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, tj. celih števil oziroma števil s plavajočo vejico. Različni objekti nosijo različno vsebino. Objekt  $\text{CON}(C a_1 \dots a_n)$  je npr. predstavljen s strukturo, ki ima na prvem mestu kazalec do metapodatkov objekta tipa CON, vsebina pa ima  $n$  polj, z vrednostmi argumentov  $a_1, \dots, a_n$  (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 celoštevilsko 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 obiše 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 [24], 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

Mislim, da je trenutna vsebina tega poglavja to, kar bi o sorodnem delu želel povedati. Potrebno se mi zdi še napisati nek boljši uvod v poglavje.

Da je upravljanje s pomnilnikom v funkcijskih programskih jezikih še vedno aktualno področje raziskovanja, priča mnogo eksperimentalnih jezikov razvitih v zadnjih nekaj letih. V sledečem poglavju bomo na kratko predstavili nekaj funkcijskih jezikov, ki za upravljanje s pomnilnikom uporabljajo manj konvencionalne pristope, nato pa se bomo podrobneje posvetili sistemom tipov in na kratko predstavili *neučakan* programski jezik Granule, ki za upravljanje s pomnilnikom že uporablja princip lastništva in izposoje na podlagi tistega iz Rusta.

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].

Na podlagi principa lastništva in izposoje iz Rusta je nastal len funkcijski programski jezik Blang [30]. Interpreter jezika zna pomnilnik za ovojnice izrazov in spremenljivk med izvajanjem samodejno sproščati brez uporabe

čistilcev, zatakne pa se pri sproščanju funkcij in delnih aplikacij. **Potrebno je bolj podrobno opisati kako deluje Blang.**

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 Sistemi tipov

Sistemi tipov formalno definirajo pravila za določanje podatkovnega tipa poljubnega izraza v programu. Poleg tega postavljajo omejitve, ki jih morajo vsi izrazi v programu izpolnjevati. Med njimi določijo katere operacije se lahko izvajajo nad izrazi z določenimi tipi in kakšnega tipa je izračunan rezultat. Tipični primer tovrstnih pravil je, da se aritmetične operacije, kot sta seštevanje in množenje, lahko izvajajo le nad numeričnimi tipi, medtem ko je seštevanje nizov in števil prepovedano.

Glede na način preverjanja tipov, ločimo statično in dinamično tipiziranje. Pri jezikih s statičnim tipiziranjem (angl. *static typing*) se podatkovni tipi izpeljejo oziroma izračunajo med prevajanjem, medtem ko se pri dinamičnem tipiziranju (angl. *dynamic typing*) preverjanje tipov izvaja med samim izvajanjem programa. Tako je pri statično tipiziranih jezikih, kot sta Java in Haskell, zagotovljeno, da se napake, povezane z nezdržljivostjo tipov, odkrijejo že med prevajanjem in ne med izvajanjem programa. Na drugi strani so jeziki z dinamičnim tipiziranjem, kot sta Python in JavaScript, podvrženi večjemu tveganju za napake med izvajanjem.

V poglavju 2 smo lahko videli, da STG jezik ne vključuje preverjanja tipov, vendar to ne pomeni, da ni tipiziran. Podatkovni tipi so namreč izpeljani in preverjeni pred prevajanjem Haskell v STG. Izrek o varnosti sistemov tipov Haskell zagotavlja, da *med izvajanjem* STG jezika z redukcijo grafa ne bo prišlo do napak, povezanih z nezdržljivostjo tipov.

Pri izpeljavi oziroma preverjanju tipov, v kontekstu  $\Gamma$  hranimo predpostavke o tipih spremenljivk, na katere smo že naleteli. Te predpostavke pomagajo pri določitvi tipov kompleksnejših izrazov in funkcij, saj lahko z uporabo že znanih tipov preverjamo in izpeljujemo tipe novih izrazov. Na ta način lahko v vsakem koraku preverimo, ali so tipi skladni s pravili sistema tipov.

Programski jeziki imajo običajno neomejen (angl. *unrestricted*) sistem tipov, ki omogoča, da lahko do spremenljivk dostopamo poljubno mnogokrat in v poljubnem vrstnem redu. To je zagotovljeno s pomočjo treh strukturnih lastnosti:

- *zamenjava* (angl. *exchange*) zagotavlja, da vrstni red spremenljivk v kontekstu tipov ni pomemben. Če je preverjanje tipov uspešno v nekem kontekstu  $\Gamma$ , potem bo uspešno tudi v kateremkoli drugem kontekstu, ki je sestavljen kot permutacija predpostavk iz  $\Gamma$ .
- *oslabitev* (angl. *weakening*) zagotavlja, da se lahko, kljub dodajanju neuporabnih predpostavk v kontekst tipov, izrazu še vedno določi tip.
- *zoženje* (angl. *contraction*) ki zagotavlja, da če lahko preverimo tip izraza z uporabo dveh enakih predpostavk, lahko isti izraz preverimo tudi z uporabo samo ene predpostavke. To pomeni, da lahko spremenljivko v izrazu uporabimo večkrat, ne da bi morali v kontekstu imeti več kopij te predpostavke.

Substrukturni sistemi tipov (angl. *substructural type systems*) [33] so sistemi tipov, pri katerih ne velja ena izmed treh strukturnih lastnosti. Mednje sodi npr. *urejen* (angl. *ordered*) sistem tipov, pri katerem ne velja nobena izmed treh lastnosti, kar v praksi pomeni, da morajo biti vse spremenljivke

uporabljene natanko enkrat in to v vrstnem redu kot so bile deklarirane. V nadaljevanju si bomo podrobneje ogledali linearen sistem tipov, ki dovoljuje zamenjavo, ne pa tudi oslabitve in zoženja.

## 3.2 Linearni tipi

Linearen sistem tipov (angl. linear type system) je substrukturni sistem tipov, ki se od urejenega razlikuje v tem, da dovoljuje *zamenjavo*. To pomeni, da zahteva, da je vsak objekt uporabljen *natanko enkrat*, ne velja pa, da morajo biti ti uporabljeni v vrstnem redu, kot so bili deklarirani. Jeziki s sistemom linearnih tipov omogočajo pisanje programov, ki bolj varno upravljajo s pomnilnikom, saj zagotavljajo, da bo na vsako vrednost kazal natanko en kazalec, kar pomeni, da ne more priti do nenadzorovanih ali nepričakovanih sprememb podatkov, tudi če se program izvaja hkrati na več nitih. Prav tako pa omogočajo bolj natančen nadzor nad življenjskimi cikli spremenljivk, kar preprečuje pisanje programov, ki puščajo pomnilnik ali programov, ki ne sprostijo virov operacijskega sistema po uporabi [33].

Pri linearnih sistemih tipov gre referenca izven dosega takoj, ko se pojavi na desni strani prirejanja ali ko je posredovana kot argument funkciji. S tem je zagotovljeno, da na en objekt na kopici vedno kaže natanko ena referenca. Funkcija  $f$  linearna, če *uporabi* svoj argument natanko enkrat. Oznaka  $f :: \alpha \multimap \beta$  predstavlja tip linearne funkcije  $f$ , ki sprejme argument tipa  $\alpha$  in vrne rezultat tipa  $\beta$ .

Če sistem tipov ne omogoča oslabitve, potem je zagotovljeno, da nobene vrednosti ne moremo zavreči [34]. V tem primeru leni izračun sploh ni potreben, saj bo vsaka vrednost zagotovo vsaj enkrat uporabljena. Če prepovemo pravilo zoženja, potem v jeziku vrednosti ne moremo podvajati, kar pomeni, da na vsako vrednost na pomnilniku kaže natanko ena referenca, zaradi česar tak jezik ne potrebuje avtomatičnega čistilca pomnilnika. V linearnem sistemu tipov sta obe pravili prepovedani, s čemer je zagotovljeno, da bo vsaka vrednost uporabljena *natanko enkrat*, po svoji edini uporabi pa lahko tako

sistem ustrezno sprosti pomnilnik. Težava, ki se pojavi pri takem jeziku pa je v tem, da je pogosto preveč omejujoč, saj spremenljivk ni mogoče podvajati ali zavreči [35]. Naslednji program prikazuje dve funkciji, ki se v jeziku z linearnim sistemom tipov ne bi prevedli. Funkcija `duplicate` namreč argument  $x$  uporabi dvakrat, funkcija `fst` pa argumenta  $y$  sploh ne uporabi, kar krši pravila linearnosti.

```
1 duplicate x = (x, x)
2 fst (x, y) = x
```

Programski jezik z linearnim sistemom tipov

Zaradi teh omejitev, se v programske jezike poleg linearnih tipov pogosto uvede še nelinearne [33, 35, 36]. Nelinearne vrednosti so v takih jezikih posebej označene in omogočajo, da je vrednost uporabljena poljubno mnogokrat, tj. nič ali večkrat. Vendar pa uvedba nelinearnih tipov prinaša izzive pri upravljanju s pomnilnikom. Prevajalnik namreč ne more zanesljivo določiti, kdaj na določeno vrednost ne kaže več nobena referenca, kar pomeni, da je za čiščenje nelinearnih vrednosti v jeziku še vedno potrebna implementacija avtomatičnega čistilca pomnilnika.

### Girardova linearna logika

Sam sistem linearnih tipov temelji na Girardovi linearni logiki [37]. Ta vsebuje tako linearne, kot tudi nelinearne tipe, prehajanje med njimi pa je omogočeno s pomočjo pravil **promocije** (angl. promotion) in **derelikcije** (angl. dereliction).

**Promocija** je pravilo, ki omogoča deljenje vrednosti, če je zagotovljeno, da je mogoče deliti tudi vse proste spremenljivke, ki se v vrednosti pojavijo. Če bi bila katera izmed prostih spremenljivk linearna, bi z deljenjem nanjo ustvarili več referenc, kar pa krši pravila linearnosti. Promocija torej omogoča, da linearno vrednost pretvorimo v nelinearno in jo kot tako uporabimo večkrat oziroma sploh ne.

**Derelikcija** je, v kontekstu linearnih tipov, operacija, ki omogoča pre-

tvorbo nelinearnega tipa v linearnega [34]. S pravilom zoženja omogočimo, da lahko nelinearne vrednosti uporabimo večkrat, s pravilom oslabitve omogočimo, da vrednost sploh ni uporabljena, pravilo derelikecije pa omogoči, da nelinearno vrednost uporabimo natanko enkrat. tj. linearno. Brez **derelikecije** v jeziku namreč nelinearnih vrednosti ni mogoče uporabljati kot argumente linearnih funkcij. Toda zaradi pravila **derelikecije** ni mogoče zagotoviti, da ima linearen tip le eno referenco, kar pomeni, da tudi pomnilnika za linearne tipe ni mogoče sprostiti takoj po njihovi prvi uporabi.

### Wadlerjev sistem **steadfast** tipov

Wadler v svojem delu [35] predstavi *len* programski jezik z linearnim sistemom tipov. Tipi so razdeljeni na dve družini, med njima pa ni mogoče implicitno prehajati z uporabo **promocije** oziroma **derelikecije**. Linearni tipi v jeziku predstavljajo reference z možnostjo pisanja (angl. write access), medtem ko nelinearni tipi omogočajo le dostop za branje (angl. read-only access). Za sestavljene podatkovne tipe v jeziku velja, da nelinearni tipi ne smejo vsebovati referenc na linearne tipe. Ker lahko nelinearne tipe podvajamo, bi se v tem primeru namreč lahko zgodilo, da bi podvojili tudi referenco na linearen tip, s čemer pa bi prekršili pravila linearnosti v jeziku.

Prehajanje med linearnimi in nelinearnimi tipi je omogočeno le na en način, *eksplicitno* s pomočjo izraza **let!**. Pri tem je mogoče znotraj izraza *u*, vrednost *x* uporabljati *nelinearno*, a le za branje. Več kot ena referenca na vrednost na pomnilniku je namreč varna, dokler obstaja *v trenutku posodobitve* nanjo samo ena referenca. V izrazu *v* pa je tip spremenljivke *x* ponovno linearen, kar pomeni, da je vrednost mogoče neposredno posodabljanje ali izbrisati iz pomnilnika.

```
1  let! (x) y = u in v
```

Wadlerjev len programski jezik

Toda, zaradi lenosti jezika, bi lahko spremenljivka *y* preživela **let!** izraz, pri tem pa vsebovala kazalec na spremenljivko *x*, ki je linearno uporabljena

v telesu izraza  $v$ . Izraz `let!` je zato edini izraz, ki se ne izvaja leno, temveč neučakano. Nujno je namreč izračunati *celoten* izraz  $u$  preden se začne izvajati izraz  $v$ , da zagotovimo, da bodo vse reference na  $x$  odstranjene preden se bo začel izračun  $v$ , ki bo mogoče sprostil vrednost spremenljivke  $x$ . To imenujemo tudi za *zelo neučakani izračun* (angl. hyperstrict evaluation).

Wadler v svojem delu torej uvede dva povsem ločena “svetova” tipov, med katerimi je moč prehajati z uporabo izraza `let!`. Tak sistem tipov poimenuje tudi za `steadfast` (angl. steadfast) [34]. Pri tem pokaže, da je zaradi uvedbe nelinearnosti, še vedno potreben avtomatski čistilec pomnilnika, saj so nelinearne vrednosti lahko poljubno podvojene. Pokaže tudi, da je potrebno zagotoviti, da se vsi nelinearni dostopi do objekta na pomnilniku izvedejo pred dostopom za pisanje, kar pa je pri lenem izračunu skoraj nemogoče izvesti, zato v `let!` izraze ponovno uvede neučakan izračun.

Linearni tipi so bili tudi že dodani v Haskell kot razširitev sistema tipov [38]. Najpomembnejša pridobitev članka je vpeljava linearnih tipov v Haskell, ki omogoča varno in učinkovito posodabljanje podatkovnih struktur ter zagotavljanje pravilnega dostopa do zunanjih virov, kot so datoteke in omrežni viri. Avtorji so dokazali, da je mogoče linearne tipe vključiti v obstoječi programski jezik s spreminjanjem algoritma za preverjanje in izpeljavo tipov, ne pa tudi s samim spreminjanjem abstraktnega STG stroja na katerem se izvaja redukcija grafa.

### 3.3 Unikadni tipi

**Unikatni tipi** (angl. uniqueness types) so namenjeni zagotavljanju, da na vsako vrednost kaže natanko ena referenca, kar omogoča učinkovito implementacijo sistema, ki omogoča posodobitve na mestu (angl. in-place updates) [36]. V literaturi se unikatni tipi pogosto kar enačijo z linearnimi tipi oziroma se obravnavajo kot posebna vrsta linearnih tipov [33, 38]. Za vrednosti linearnih tipov velja, da v *prihodnosti* zagotovo ne bojo podvojene ali zavržene, medtem ko je za unikatne vrednosti zagotovljeno, da v *preteklosti*

še niso bile podvojene [36, 39].

Ker včasih želimo da na eno vrednost kažeta dva kazalca, se tudi pri unikatnih tipih, podobno kot pri linearnih, uvede neomejene (angl. *unrestricted*) vrednosti. Razlika med linearnimi in unikatnimi tipi je v zmožnosti prehajanja med neomejenimi in omejenimi vrednostmi. Kot smo videli v poglavju 3.2, lahko pri linearnem sistemu tipov med linearnimi in nelinearnimi vrednostmi prehajamo s pomočjo pravil *promocije* in *derelikcije*. S pomočjo derelikcije je lahko nelinearna spremenljivka v nadaljevanju uporabljena linearno, pri tem pa ni mogoče zagotoviti, da na to nelinearno spremenljivko kaže natanko ena referenca. To pa tudi pomeni, da v linearnem sistemu ne moremo zagotoviti unikatnosti vrednosti.

Pri unikatnih sistemih tipov ni pravila, ki bi omogočala pretvorbo vrednosti neomejenega tipa nazaj v vrednost unikatnega tipa. Ker na neomejene tipe namreč lahko kaže poljubno mnogo kazalcev, jih ni mogoče obravnavati kot unikatne. Kot bomo lahko videli v poglavju 3.4, programski jezik Granule tako pretvorbo omogoča s ključno besedo `clone`, ki globoko kopira (angl. *deep copy*) vrednost na pomnilniku. S tem je sicer zagotovljena unikatnost, a je kopiranje precej neučinkovito, saj je potrebno klonirati celoten podgraf na pomnilniku [39]. Pri sistemih unikatnih tipov torej velja, da je vrednost unikatnega tipa mogoče pretvoriti v neomejeno vrednost, obratno pa ne. Linearni tipi so tako bolj uporabni pri zagotavljanju pravilne uporabe računalniških *sredstev* (angl. *resource*), medtem ko sistemi unikatnih tipov omogočajo ponovno uporabo struktur na pomnilniku in posodabljanje le-teh na mestu [36].

V sistemu tipov, kjer je mora biti *vsaka* vrednost linearna, je zagotovljeno tudi, da je vsaka vrednost unikatna [36]. Linearni tipi namreč ne dovoljujejo podvajanja, zaradi česar je zagotovljeno, da bo referenca na nek vrednost vedno le ena. Wadlerjev sistem *steadfast* linearnih tipov [35] omeji pravili *promocije* in *derelikcije*, s čemer v jezik ponovno uvede pogoj za unikatnost reference. Sistem tipov jezika glede na definicijo bolj ustreza sistemu linearnih tipov, ki pa takrat še ni bil definiran.



### Programski jezik Clean

Eden izmed programskih jezikov, ki uporabljajo sistem unikatnih tipov, je len funkcijski jezik Clean [40]. Za razliko od Haskell, ki za mutacije notranjega stanja in vhodno-izhodne operacije uporablja monade, Clean le-te implementira s pomočjo sistema unikatnih tipov. Prav tako zna prevajalnik unikatne vrednosti spreminjati na mestu, kar zmanjša porabo pomnilnika in omogoča hitrejšo izvajanje programov.

Spodnji primer prikazuje program v jeziku Clean. Konstruktor tipa  $*T$  predstavlja unikaten tip  $T$ . Če predpostavljamo, da je `eat` tipa `Cake -> Happy` in `have` tipa `Cake -> Cake`, potem je naslednji program veljaven.

```
1 possible :: *Cake -> (Happy, Cake)
2 possible cake = (eat cake, have cake)
```

Clean

Kot lahko vidimo, se argument `cake` v telesu funkcije pojavi dvakrat. Funkcija vzame unikaten kazalec na vrednost `Cake`, ker pa jo v telesu dvakrat uporabi, vrnjena vrednost izgubi unikatnost. Vrnjena vrednost je tako neomejenega tipa.

## 3.4 Programski jezik Granule

Programski jezik Granule [41] je *neučakan močno tipiziran* (angl. strongly typed) funkcijski jezik, ki v svojem sistemu tipov združuje linearne, indeksne in *graded modal* (angl. graded modal) tipe. Granule uporablja princip podatkov kot virov (angl. data as a resource) v svojem sistemu tipov. Za upravljanje s pomnilnikom je uporabljen avtomatičen čistilec.

S pomočjo linearnih tipov je v jeziku zagotovljen pogled na podatke kot na fizičen vir, ki mora biti uporabljen enkrat, nato pa nikoli več. Neomejena uporaba nekega vira mora biti v jeziku označena z eksponentnim *graded modalityjem*  $!A$ , ki omogoča, da je vrednost lahko deljena poljubno mnogokrat. Jezik poleg neomejene uporabe omogoča še določanje zgornje meje uporabe

podatkov s pomočjo omejene linearne logike (angl. bounded linear logic) [42]. Tako lahko namesto neomejene uporabe  $!A$ , določimo zgornjo mejo uporabe vrednosti, tj. s  $!_2 A$  označimo, da bo vrednost  $A$  uporabljena *največ* dvakrat.

Preverjanje tipov je v jeziku Granule implementirano v dveh stopnjah: najprej se za izraze v programu izpelje trditve in omejitve glede njihovih tipov [41], nato pa se trditve dokaže s pomočjo dokazovalnika Z3 [43]. Izpeljava tipov (angl. type inference) za **statične** (angl. top-level) funkcije ni podprta, zato morajo biti označeni tipi vseh funkcij na statičnem nivoju. V Granule so vse funkcije privzeto linearne, zato se namesto operatorja za linearne funkcije  $a \multimap b$ , uporablja kar zapis  $a \rightarrow b$ . Neomejeni tipi kot jih poznamo iz linearne logike, so označeni s pripono  $[]$ . Taka oznaka je ekvivalentni oznaki  $!$ , ki jo je definirал Girard [37], omogoča pa poljubno mnogo uporab spremenljivke, tako da omogoči pravili **oslabitve** in **zoženja**. Jezik prav tako omogoča omejevanje števila uporab neke spremenljivke s pomočjo pripone  $[n]$ , ki določa, da je lahko število uporab spremenljivke največ  $n$ .

Naslednji primer prikazuje identiteto, implementirano v jeziku Granule. Iz **prototipa** (angl. type annotation) lahko prevajalnik razbere, da je `id` funkcija, ki sprejme spremenljivko poljubnega tipa in jo zaradi linearnosti (konstruktor tipa  $t \rightarrow t$ ), uporabi natanko enkrat.

```

1  id : ∀ {t : Type} . t → t
2  id x = x

```

Granule

V naslednjem primeru sta implementirani funkciji **drop** in **copy**, ki v jeziku z le linearnimi tipi, nista mogoči. Pri obeh funkcijah je označeno število uporab argumenta. Funkcija **drop** svojega argumenta ne uporabi, zato je označena z števnostjo  $t [0]$ , funkcija **copy** pa svoj argument uporabi dvakrat, kar je označeno s števnostjo  $t [2]$ .

```

1 drop : ∀ {t : Type} . t [0] → ()
2 drop [x] = ()
3
4 copy : ∀ {t : Type} . t [2] → (t, t)
5 copy [x] = (x, x)

```

Granule

### Unikatni tipi v jeziku Granule

V programski jezik Granule je bila eksperimentalno poleg linearnih tipov dodana tudi podpora za unikatne tipe [36]. Linearnost v takem jeziku omogoča boljši nadzor nad upravljanjem z resursi, medtem ko je unikatnost uporabljena za varno posodabljanje podatkov na mestu. Prevajalnik jezika Granule zna s pomočjo izpeljanih unikatnih tipov, generirati optimizirano Haskell kodo, ki **mutira** sezname na mestu. Avtorji so pokazali, da je uporaba unikatnih tipov za delo s tabelami učinkovitejša od uporabe nespremenljivih tabel (angl. immutable arrays). Rezultati so pokazali, da je različica z unikatnimi tabelami hitrejša in porabi bistveno manj časa za upravljanje s pomnilnikom. To je posledica dejstva, da se unikatni podatki **alocirajo** izven kopice GHC prevajalnika in se lahko eksplicitno sprostijo po njihovi uporabi. Avtorji tudi poudarijo, da sistem unikatnih tipov omogoča varno in učinkovito mutacijo podatkov neposredno v funkcijskem jeziku, brez potrebe po uporabi nepreverjene kode (angl. unsafe code), ki je npr. prisotna v Haskell knjižnicah za učinkovite operacije nad tabelami.

### Model lastništva v jeziku Granule

Pozneje je bil sistem tipov jezika Granule še dodatno razširjen s pravili za lastništvo in izposajo na podlagi tistih iz Rusta [39]. Avtorji v članku povežejo koncepta linearnih in unikatnih tipov in vgradijo sistem lastništva in izposoje v sistem tipov za funkcijski programski jezik.

Sistem lastništva v Rustu določa, da z vsako vrednostjo v pomnilniku upravlja natanko ena referenca. To se ujema z definicijo unikatnih vrednosti, zato avtorji osnovo za lastništvo objektov postavijo na sistem unikatnih tipov.

Ker vrednosti unikatnega tipa ni mogoče podvajati, avtorji uvedejo ključno besedo `clone`, ki omogoča globoko kopiranje (angl. deep copy) objekta v pomnilniku. Po operaciji na novonastali objekt zagotovo kaže le en kazalec, s čemer je omogočeno, da lahko vrednost poljubnega tipa pretvorimo v unikatni tip. Dodana je še ključna beseda `share`, ki omogoča deljenje izraza. Pri deljenju se vrednost unikatnega tipa  $*A$  pretvori v neomejeno vrednost  $!A$ , ki je od takrat naprej ni mogoče ponovno pretvoriti v unikatni tip. Za čiščenje pomnilnika neomejenih tipov se še vedno uporablja avtomatski čistilec, medtem ko se za čiščenje unikatnih in linearnih tipov uporablja Rustov model upravljanja s pomnilnikom.

Izposoje so v jeziku implementirane s pomočjo delnih pravic (angl. fractional permissions) [44]. Tipi z delnimi pravicami so označeni  $\&_p A$ , kjer  $p$  predstavlja ali vrednost  $*$  ali pa ulomek na intervalu  $[0, 1]$ . Vrednosti tipa  $\&_* A$  predstavljajo unikatne izposoje in so z vrednostmi unikatnih tipov povezane s pomočjo enakosti  $*A \equiv \&_* A$ . Spremenljive izposoje so označene s tipom  $\&_1 A$ , pri nespremenljivih izposojah pa je  $p < 1$ . Z izrazom `split` je omogočeno, da se referenca  $\&_p A$  razdeli na dve novi referenci, ki kažeta na isti objekt kot prvotna referenca. Novi referenci sta označeni s polovico dovoljenj prvotne reference, tj. referenci imata tipa  $\&_{\frac{p}{2}} A$ . Z izrazom `join` se dve obstoječi referenci združita v eno novo referenco, pri čemer se dovoljenja združenih referenc seštejeta [39]. Tako lahko spremenljivo referenco s pomočjo izraza `split` razdelimo na dve nespremenljivi referenci in s pomočjo izraza `join` ponovno združimo v spremenljivo referenco.

Na tem mestu je še vredno omeniti, da je programski jezik Granule *neučakan*. “Call-by-value was chosen for simplicity and since the effective part of Gr is necessarily CBV (to avoid pitfalls of laziness with side effects).“ [41]. Preveriti moram še, zakaj je nujno imeti prenos po vrednosti.

## Poglavje 4

# Težave pri izračunu življenjskih dob pri lenem izračunu

### 4.1 Izpeljava življenjskih dob

Programski jezik Rust je neučakan, kar pomeni, da je vrstni red operacij v programu natanko določen. Če se stavek  $s_1$  v programu pojavi pred stavkom  $s_2$ , potem se bo stavek  $s_1$  zagotovo izvedel pred  $s_2$ .

V spodnjem primeru se bodo stavki zagotovo izvajali glede na vrstni red, v katerem so zapisani. Tako se bo klic funkcije `mul` zagotovo izvedel pred klicem `div`, ta pa se bo zagotovo izvedla pred klicem `add`.

```
1  fn izracunaj() -> i32 {  
2      let a = mul(6, 2);  
3      let b = div(8, 2);  
4      let r = add(b, a);  
5      return r;  
6  }
```

Rust ✓

V jezikih z lenim izračunom pa temu ni tako. Pri prirejanju v spremenljivko  $a$ , bi se na kopici najprej ustvarila nova zakasnitev za poznejši izračun vrednosti izraza `mul(6, 2)`. V spremenljivki  $a$  bi se hranil kazalec na na-

slov na katerem se nahaja zakasnitev. Na podoben način bi se izvedlo še prirejanje v  $b$ , pri prirejanju v spremenljivko  $r$  pa bi se v ovojnico zakasnitve shranila še kazalca na zakasnitvi  $a$  in  $b$  na kopici. Ker je izračun len, se pri klicu funkcije `izracunaj` na kopici samo ustvarijo novi objekti, ne izračuna pa se njihova dejanska vrednost. Rezultat klica `izracunaj` je tako naslov ovojnice zakasnitve  $r$  na kopici.

V poglavju bomo predpostavljali, da pri prevajanju niso uporabljene nobene optimizacije, in da se dosegi spremenljivk računajo glede na leksikalen doseg. V poglavju 1.2 smo že omenili, da se v Rustu dosegi oziroma življenjske dobe računajo neleksikalno. Pri računanju življenjskih dob, se namreč upošteva analiza živosti spremenljivk (angl. *liveness analysis*), kar pomeni, da imajo lahko spremenljivke krajšo življenjsko dobo kot funkcija, ki jih je ustvarila. V tem poglavju predpostavimo, da je spremenljivka živa od njene uvedbe, do konca funkcije v kateri je bila definirana, oziroma do prvega (edinega) premika spremenljivke.

Ker je vrstni red operacij natanko določen, je zelo preprosto izračunati tudi življenjske dobe. V zgornjem primeru ima spremenljivka  $a$  zagotovo daljšo življenjsko spremenljivko  $b$ , saj je bila deklarirana pred uvedbo spremenljivke  $b$ . Ker bi se spremenljivka  $b$  lahko sklicevala na spremenljivko  $a$ , je potrebno zagotoviti, da spremenljivka  $a$  živi vsaj toliko časa kot  $b$ , kar pomeni da je življenjska doba spremenljivke  $a$  večja od  $b$ .

Zaradi lenosti jezika je izračun življenjskih dob praktično nemogoč, kar pomeni, da Rustov princip čiščenja pomnilnika ne pride v poštev. Ker ne moremo izračunati življenjskih dob, v STG ne moremo implementirati izposoje. Kar se tiče pa samega lastništva iz Rusta, je pa to zelo podobno linearnim oziroma edinstvenim tipom (angl. *uniqueness types*). Ti so bili v Haskell že implementirani (glej "Linear Haskell: practical linearity in a higher-order polymorphic language" [38]). Če v jezik uvedemo le linearne tipe, jezik zelo omejimo in postane dokaj neuporaben. Zato navadno dodamo še nelinearne tipe, zaradi katerih pa: 1. jezik ni več povsem len (glej poglavje 3.1.1) 2. v jeziku še vedno potrebujemo avtomatičen čistilec pomnilnika (linearne tipe

lahko počistimo brez, nelinearnih pa na žalost ne)

## 4.2 Primer

Za vsakršno vrsto analize je potrebno v jezik STG ponovno potrebno uvesti oznake tipov.





## Poglavje 5

## Zaključek

V sklopu magistrskega dela nam je uspelo implementirati delujoč abstraktni STG stroj.



# 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).
  - [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] 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.
  - [17] 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.

- 
- [18] 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.
  - [19] E. Czaplicki, Elm : Concurrent FRP for functional GUIs, Senior thesis, Harvard University 30 (2012).
  - [20] 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.
  - [21] D. Syme, The f# compiler technical overview (3 2017).
  - [22] 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.
  - [23] 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.
  - [24] 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.
  - [25] 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.
  - [26] R. Jones, Tail recursion without space leaks, Journal of Functional Programming 2 (1) (1992) 73–79.
  - [27] U. Boquist, T. Johnsson, The GRIN project: A highly optimising back end for lazy functional languages, in: Implementation of Functional Lan-

- guages: 8th International Workshop, IFL'96 Bad Godesberg, Germany, September 16–18, 1996 Selected Papers 8, Springer, 1997, pp. 58–84.
- [28] P. Podlovics, C. Hruska, A. Péntzes, 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 Högskola* (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] B. C. Pierce, *Advanced topics in types and programming languages*, MIT press, 2004.
- [34] P. Wadler, Is there a use for linear logic?, *ACM SIGPLAN Notices* 26 (9) (1991) 255–273.
- [35] P. Wadler, Linear types can change the world!, in: *Programming concepts and methods*, Vol. 3, Citeseer, 1990, p. 5.
- [36] D. Marshall, M. Vollmer, D. Orchard, Linearity and uniqueness: An entente cordiale, in: I. Sergey (Ed.), *Programming Languages and Systems*, Springer International Publishing, Cham, 2022, pp. 346–375.
- [37] J.-Y. Girard, Linear logic, *Theoretical computer science* 50 (1) (1987) 1–101.

- 
- [38] J.-P. Bernardy, M. Boespflug, R. R. Newton, S. Peyton Jones, A. Spiwack, Linear haskell: practical linearity in a higher-order polymorphic language, in: *Principles of Programming Languages 2018 (POPL 2018)*, ACM, 2018.
  - [39] D. Marshall, D. Orchard, Functional ownership through fractional uniqueness, *Proc. ACM Program. Lang.* 8 (OOPSLA1) (apr 2024).
  - [40] S. Smetsers, E. Barendsen, M. van Eekelen, R. Plasmeijer, Guaranteeing safe destructive updates through a type system with uniqueness information for graphs, in: *Graph Transformations in Computer Science: International Workshop Dagstuhl Castle, Germany, January 4–8, 1993 Proceedings*, Springer, 1994, pp. 358–379.
  - [41] 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.
  - [42] J.-Y. Girard, A. Scedrov, P. J. Scott, Bounded linear logic: a modular approach to polynomial-time computability, *Theoretical computer science* 97 (1) (1992) 1–66.
  - [43] L. De Moura, N. Bjørner, Z3: An efficient smt solver, in: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2008, pp. 337–340.
  - [44] J. Boyland, Checking interference with fractional permissions, in: *International Static Analysis Symposium*, Springer, 2003, pp. 55–72.