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, 2024

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, 2024

Vsem rožicam tega sveta.

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

— Albert Einstein

Kazalo

Seznam uporabljenih kratic

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

Povzetek

Naslov: Lastništvo objektov namesto avtomatskega čistilca pomnilnika med lenim izračunom

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

Ključne besede

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

Abstract

Title: Ownership model instead of garbage collection during lazy evaluation

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

Keywords

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

Poglavje 1

Uvod

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

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

Posredne metode, npr. označi in pometi [3], ne posodabljajo metapodatkov na pomnilniku ob vsaki spremembi, temveč se izvedejo le, kadar se prekorači velikost kopice. Algoritem pregleda kopico in ugotovi, na katere

objekte ne kaže več noben kazalec ter jih odstrani. Nekateri algoritmi podatke na kopici tudi defragmentirajo in s tem zagotovijo boljšo lokalnost ter s tem boljše predpomnjenje [4]. Problem metode označi in pometi 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 Leni izračun

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

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

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

Haskell ✓

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

Če semantika jezika dopušča izračun vrednosti izraza, kljub temu, da njegovi podizrazi nimajo vrednosti, jo imenujemo za nestrogo semantiko [6]. Bolj formalno lahko uvedemo vrednost $\bot$, ki jo imenujemo tudi dno (angl. bottom) in predstavlja vrednosti izrazov, katerim ni mogoče izračunati normalne oblike (tj. izrazi, katerih vrednosti ne moremo izračunati oziroma izrazi, ki vrnejo napako). Če izraza *expr* ne moremo izračunati, lahko tako zapišemo, da zanj velja $\textbf{eval}(expr) = \bot$. Za funkcijo enega argumenta pravimo, da je *stroga*, če velja pogoj STRICT.

$$f \bot = \bot \qquad \text{(STRICT)}$$

Za stroge funkcije enega argumenta torej velja, da če pride pri izračunu argumenta pri klicu funkcije do napake (tj. vrednosti $\bot$), bo tudi rezultat funkcije napaka $\bot$. Če funkcija ni stroga, pravimo da je nestroga. Za nestroge funkcije torej velja, da lahko vrednost funkcije vseeno izračunamo, kjub temu, da ne moremo nujno izračunati vrednosti argumenta (tj. da velja $f \bot \neq \bot$).

Če programski jezik za izračun vrednosti izrazov uporablja strogo semantiko, pravimo, da je *neučakan* (angl. eager). Vrednosti izrazov se pri

neučakanem izračunu izračunajo takoj, ko se v programu pojavijo. Pri tem lahko izračun argumenta povzroči stranske učinke, katerih vrstni red in čas izvedbe sta ključna za nadaljnjo delovanje programa, zato večina imperativnih jezikov uporablja neučakani izračun [6]. Funkcijski programski jeziki kot so npr. Scheme, ML ali OCaml, omogočajo uporabo stranskih učinkov, in prav tako implementirajo neučakani izračun.

Če jezik implementira nestrogo semantiko, pravimo, da je *len* (angl. lazy). Pri lenem izračunu se torej vrednost izraza ne izračuna takoj, ko je ta definiran, ampak šele, ko je njegova vrednost dejansko potrebna.

Ena glavnih pomanjkljivosti neučakanega izračuna je, da se pri klicu funkcije vedno izračunajo vse vrednosti argumentov, ne glede na to, ali bo vrednost posameznega argumenta v telesu funkcije sploh uporabljena. To pomeni, da lahko pride do nepotrebnega računanja, kar vpliva na učinkovitost programa. Prednost neučakanega izračuna pa je večja predvidljivost poteka programa [6], saj za razliko od lenega izračuna natanko vemo kdaj se bo vrednost nekega izraza izračunala. Glavna prednost lenega izračuna je v tem, da se argumenti izračunajo *največ enkrat*. Če argument v telesu funkcije ni uporabljen, se tudi nikoli ne izračuna. V primeru, da se uporabi, pa se njegova vrednost izračuna *enkrat* in se nato memoizira za vse nadaljnje uporabe. Slabost lenega izračuna pa je težja implementacija in počasnejše izvajanje v primerjavi z jeziki, ki uporabljajo neučakan izračun [6, 7].

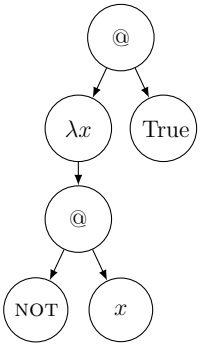
Redukcija grafa

Zgodnejši programski jeziki so bili namenjeni neposrednemu upravljanju računalnika oziroma komuniciranju z vhodno izhodnimi napravami in so kot taki precej odražali delovanje računalniške arhitekture na kateri so se izvajali [8]. 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 zakasnje-

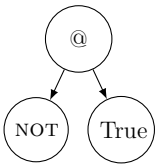
nimi izrazi (angl. thunks) in zagotavljanje, da se ti izrazi izračunajo le, ko so resnično potrebni.

Leni izračun najpogosteje implementiramo s pomočjo redukcije grafa [6, 8]. 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 1.1a 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 1.1b 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

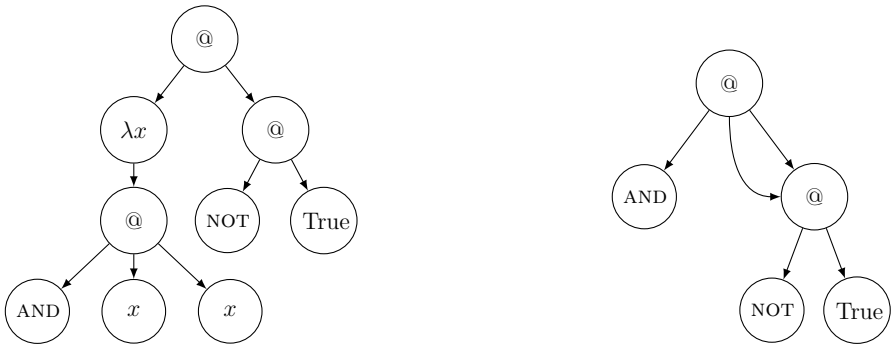


(b) Abstraktno sintaksno drevo *po* redukciji

Slika 1.1: Redukcija grafa izraza $(\lambda x \,.\, \text{NOT}\ x)$ `True`

Slika 1.2 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.

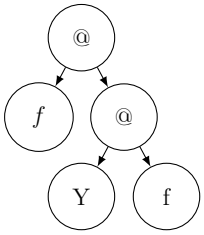


Slika 1.2: Redukcija grafa izraza $(\lambda x\,.\,\text{AND}\ x\ x)$ (NOT `True`)

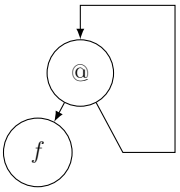
Slika 1.3 prikazuje dve možni implementaciji ciklične funkcije $Y\ f = f\ (Y\ f)$. Za razliko od primerov na slikah 1.1 in 1.2, pri katerih je bilo reducirano sintaktično drevo še vedno usmerjen acikličen graf, pa temu pri funkciji Y ni več tako. Funkcija Y je namreč rekurzivna, kar pomeni, da se sama pojavi kot vrednost svojega argumenta. Na sliki 1.3a 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 1.3b je funkcija implementirana neposredno s pomnilniškem ciklu, kjer je vrednost argumenta kar vozlišče samo.

Zapis grafa na pomnilniku

Leni izračun najpogosteje implementiramo s pomočjo zakasnjenih izrazov oziroma zakasnitev (angl. thunks). Te so na pomnilniku predstavljene kot ovojnice, tj. strukture s kazalcem na kodo, ki izračuna njihovo vrednost in polj, ki vsebujejo vezane in proste spremenljivke. Ob izračunu zakasnitve



(a) Funkcija Y implementirana z uporabo proste spremenljivke



(b) Funkcija Y implementirana kot cikličen usmerjen graf

Slika 1.3: Graf funkcije $Y\ f = f\ (Y\ f)$

(angl. forcing a thunk) se najprej izračuna njena vrednost, izračunano vrednost, nato pa se izračunana vrednost shrani v strukturo na pomnilniku, da je ob naslednji evalvaciji ni potrebno ponovno računati. Pravimo, da se vrednost na pomnilniku *posodobi*. Tako v programskem jeziku zagotovimo nestrogo semantiko, pri kateri 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.

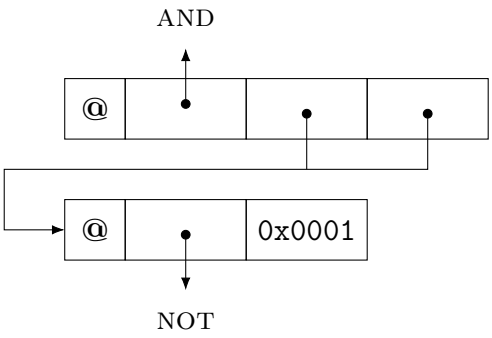
Slika 1.4 prikazuje eno izmed možnih predstavitev vozlišča grafa. Sestavljena je iz oznake vozlišča in polj z vsebino oziroma argumenti $a_1, \ldots, a_n$. Oznaka je vrednost, ki predstavlja vrsto vozlišča: aplikacija funkcije, primitivna operacija, celoštevilska vrednost, preusmeritev, ipd. V poglavju 4.2 bomo videli, da sta si struktura 1.4 in pomnilniški zapis objektov v jeziku STG precej podobna.

Oznaka	a_1	a_2	...	a_n

Slika 1.4: Pomnilniška predstavitev vozlišča grafa

Slika 1.5 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. Pri tem se navadno na pomnilniku ustvari nova struktura, ovojnico pa se prepiše s preusmeritvijo na novonastalo strukturo. Tako ob vseh nadaljnjih dostopih, vrednosti ovojnice ni potrebno ponovno računati.



Slika 1.5: 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]. 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 [9, 10]. Danes vsi večji funkcijski programski jeziki, ki omogočajo leni izračun, uporabljajo avtomatični čistilec pomnilnika [11, 12, 13, 14, 15].

Poglavje 2

Sorodno delo

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 [16] (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) [17]. Za upravljanje s pomnilnikom se v trenutni različici uporablja čistilec pomnilnika [18].

Programski jezik micro-mitten [19] je programski jezik, podoben Rustu, ki za upravljanje s pomnilnikom uporablja princip ASAP (angl. As Static As Possible) [20]. 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 do-

datnih 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.

Na podlagi principa lastništva in izposoje iz Rusta je nastal len funkcijski programski jezik Blang [21]. Interpreter jezika zna pomnilnik za *ovojnice izrazov* in *spremenljivk* med izvajanjem sproščati samodejno brez uporabe avtomatičnega čistilca pomnilnika. Jezik na podlagi deklaracij tipov (angl. type declarations) v fazi semantične analize izvede analizo premikov in izposoj, pri katerih se za spremenljivke določi življenjska doba in preveri ali izposojene spremenljivke živijo dlje od vrednosti, ki si jo izposojajo. Težava jezika je v nezmožnosti čiščenja pomnilnika funkcij in delnih aplikacij, implementiran prevajalnik pa ima v določenih primerih težave s puščanjem pomnilnika. Kako deluje Blang?

2.1 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 nezdružljivostjo tipov, od-

krijejo 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 4 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 Haskella v STG. Izrek o varnosti sistemov tipov Haskella zagotavlja, da *med izvajanjem* STG jezika z redukcijo grafa ne bo prišlo do napak, povezanih z nezdružljivostjo tipov.

Pri izpeljavi oziroma preverjanju tipov, v kontekstu Γ 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.

Večina programskih jezikov ima 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 Γ, potem bo uspešno tudi v kateremkoli drugem kontekstu, ki je sestavljen kot permutacija predpostavk iz Γ.
- *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) [22] so

sistemi tipov, pri katerih ne velja ena izmed treh strukturni 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.

2.2 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 [22].

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 je linearna, če *uporabi* svoj argument natanko enkrat. Linearno funkcijo f, ki kot vhod sprejme argument tipa α in vrne rezultat tipa β, označimo z oznako $f :: \alpha \multimap \beta$.

Če sistem tipov ne omogoča oslabitve, potem je zagotovljeno, da nobene vrednosti ne moremo zavreči [23]. 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 [24]. 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 [22, 24, 25]. 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 [26]. 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 to-

rej 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 pretvorbo nelinearnega tipa v linearnega [23]. S pravilom zoženja omogočimo, da lahko nelinearne vrednosti uporabimo večkrat, s pravilom oslabitve omogočimo, da vrednost sploh ni uporabljena, pravilo derelikcije pa omogoči, da nelinearno vrednost uporabimo natanko enkrat. tj. linearno. Brez derelikcije v jeziku namreč nelinearnih vrednosti ni mogoče uporabljati kot argumente linearnih funkcij. Toda zaradi pravila derelikcije 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 [24] 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 derelikcije. 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.

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

Wadlerjev len programski jezik

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 posodabljati ali

izbrisati iz pomnilnika.

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) [23]. 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 [27]. 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.

2.3 Unikatni 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) [25]. V literaturi se unikatni tipi pogosto kar enačijo z linearnimi tipi

oziroma se obravnavajo kot posebna vrsta linearnih tipov [22, 27]. 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 [25, 28].

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 2.2, 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 2.4, 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 [28]. 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 [25].

V sistemu tipov, kjer je mora biti *vsaka* vrednost linearna, je zagotovljeno tudi, da je vsaka vrednost unikatna [25]. Linearni tipi namreč ne dovoljujejo podvajanja, zaradi česar je zagotovljeno, da bo referenca na nek vrednost vedno le ena. Wadlerjev sistem steadfast linearnih tipov [24] 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 [29]. Za razliko od Haskella, 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 hitrejše 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.

```
possible :: *Cake -> (Happy, Cake)
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.

2.4 Programski jezik Granule

Programski jezik Granule [30] 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 v svojem sistemu tipov uporablja princip podatkov kot virov (angl. data as a resource). 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) [31]. Tako lahko namesto neomejene uporabe $!A$, določimo zgornjo mejo uporabe vrednosti. Tip $!_2\, A$ npr. označuje vrednost tipa A, ki je lahko 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 [30], nato pa se trditve dokaže s pomočjo dokazovalnika Z3 [32]. 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 $a \to b$. Neomejeni tipi kot jih poznamo iz linearne logike, so označeni s pripono `[]`. Taka oznaka je ekvivalentni oznaki `!`, ki jo je definiral Girard [26], 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 $t \to t$), 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 številnostjo `t [0]`, funkcija `copy` pa svoj argument uporabi dva-

krat, kar je označeno s številnostjo `t [2]`.

```
drop : ∀ {t : Type} . t [0] → ()
drop [x] = ()

copy : ∀ {t : Type} . t [2] → (t, t)
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 [25]. Linearnost v takem jeziku omogoča boljši nadzor nad upravljanjem s sredstvi, 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 izposojo na podlagi tistih iz Rusta [28]. 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 unikaten 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 unikaten 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) [33]. 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 [28]. Tako lahko spremenljivo referenco s pomočjo izraza `split` razdelimo na dve nespremenljivi referenci in s pomočjo izraza `join` ponovno združimo v spremenljivo referenco.

Poglavje 3

Rustov model upravljanja s pomnilnikom

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 [34], 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 (angl. exclusion principle) [35]. 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)`.

3.1 Premik

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

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

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

Pravila lastništva [34] 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.

Rustov model lastništva lahko predstavimo tudi kot graf, v katerem vozlišča predstavljajo spremenljivke oziroma objekte na pomnilniku, povezava med vozlišči $u \to v$ pa označuje, da si spremenljivka u lasti spremenljivko v. Ker ima vsaka vrednost natanko enega lastnika, lahko sklepamo, da je tak graf ravno drevo. Kadar gre spremenljivka v izven dosega, lahko Rust počisti celotno poddrevo s korenom v tako, da rekurzivno sprosti pomnilnik

za spremenljivke, ki si jih vozlišče v lasti, nato pa počisti še svoj pomnilnik. Preverjanje veljavnost pravil poteka v fazi analize premikov (angl. move check), v program pa se v tej fazi na ustrezna mesta dodajo tudi ukazi za sproščanje pomnilnika.

3.1.1 Prenos lastništva pri klicu 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.

```rust
fn identiteta(x: Complex) -> Complex { x }
let number = Complex(0, 1);
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 x.

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

Rust ✓

3.2 Izposoja

Drugi koncept, ki ga definira Rust je *izposoja*. Ta omogoča *deljenje* (angl. aliasing) vrednosti na pomnilniku. Izposoje so lahko spremenljive (angl. muta-

ble) `&mut x` ali nespremenljive (angl. immutable) `&x`. Po principu izključitve, je lahko v danem trenutku 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 check), 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. Prav tako je prepovedano spreminjati vrednost spremenljivke *number*, dokler nanjo obstaja aktivna izposoja.

```
let number = Complex(2, 1);
let a = &number;
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 tveganih stanj (angl. data races). V naslednjem primeru skušamo ustvariti dve spremenljivi referenci, zaradi česar prevajalnik ustrezno javi napako.

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

Rust ✗

Prav tako v Rustu ni veljavno ustvariti 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.

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

Rust ✗

3.2.1 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. Življenjske dobe označujemo z oznakami `'a`, `'b`, ..., najdaljša življenjska doba pa nosi oznako `'static`, ki označuje, da je objekt živ tekom celotnega izvajanja programa.

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

Rust ✗

Zgornji 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`.

Ker je notranji doseg uveden znotraj zunanjega dosega, si prevajalnik tudi označi, da je življenjska doba `'a` vsaj tako dolga kot doba `'b` (oziroma da mora biti življenjska doba `'b` največ tako dolga kot `'a`). To pomeni, da bodo vse spremenljivke, ki so definirane znotraj notranjega dosega živele manj časa od spremenljivk definiranih v zunanjem dosegu. Rust počisti pomnilnik spremenljivke *inner*, kadar se notranji doseg konča. 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 v tem primeru Rust vrne napako.

3.2.2 Eksplicitno navajanje življenjskih dob

V Rustu so eksplicitne življenjske dobe potrebne, kadar prevajalnik ne more samodejno določiti razmerij med življenjskimi dobami referenc v podpisih funkcij, metod ali struktur. Do tega pride pri funkcijah, ki sprejmejo več argumentov in vrnejo rezultat, ki se sklicuje na nekatere izmed njih ali pri strukturah, ki v poljih vsebujejo reference. V teh primerih mora Rust zagotoviti, da so vrnjene reference veljavne vsaj tako dolgo, kot je potrebno, česar pa ne zna izpeljati avtomatično, zato mora programer navesti življenjske dobe eksplicitno.

Naslednji primer prikazuje program, pri katerem pride do napake zaradi nepravilno definiranih življenjskih dob. Funkcija *longest* vrne referenco na daljšo besedo. Po definiciji ta sprejme dve referenci z *enako* življenjsko dobo `'a` in vrne referenco z življenjsko dobo `'a`, ki živi tako dolgo kot oba argumenta. V funkciji *main*, so definirane tri spremenljivke: *first* in *result* sta deklarirani v istem bloku in imata zato enako življenjsko dobo, spremenljivka *second* pa je deklarirana v ugnezdenem bloku in ima tako krajšo življenjsko dobo. Kadar se notranji blok zaključi, se počisti tudi pomnilnik spremenljivke *second*. Toda, v spremenljivko *result* se shrani kazalec na daljšo izmed besed, kar je v našem primeru spremenljivka *second*, ki pa je izbrisana pre-

den se rezultat izpiše na ekran. Rust zna s pomočjo eksplicitno navedenih življenjskih dob napako tudi odkriti in vrniti napako.

```rust
fn longest<'a>(first: &'a str, second: &'a str) -> &'a str;
fn main() {
    let first = String::from("Rust");
    let result;
    {
        let second = String::from("Haskell");
        result = longest(first.as_str(), second.as_str());
        // Spremenljivka 'second' gre tukaj izven dosega
    }
    println!("{}", result);
}
```

Rust ✗

V določenih preprostih primerih zna Rust izpeljati življenjske dobe sam. Predvsem zaradi pisanja krajše kode, Rust namreč podpira izpuščanje življenjskih dob (angl. lifetime elision) v določenih primerih. Pri tej na podlagi treh pravil prevajalnik samodejno ustvari oziroma dopolni življenjske dobe vhodnih in izhodnih argumentov. Če npr. funkcija kot vhod sprejme referenco in vrne referenco, prevajalnik predpostavlja, da imata obe enaki življenjski dobi. Kadar prevajalnik po pravilih življenjskih dob ne zna izpeljati, prevajalnik vrne napako, odgovornost za eksplicitno navajanje življenjskih dob pa preda programerju.

```rust
fn id(x: &i32) -> &i32;
fn id<'a>(x: &'a i32) -> &'a i32;
```

Rust ✓

3.2.3 Opombe

Od leta 2022 Rust podpira neleksikalne življenjske dobe (angl. non-lexical lifetimes) [36, 37], pri katerih se preverjanje izposoj in premikov izvaja nad grafom poteka programa (angl. control flow graph) namesto nad abstraktnim sintaktičnim drevesom programa [38, 39]. Prevajalnik zna s pomočjo neleksi-

kalnih 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. Ker deklarirane spremenljivke nikjer v prihodnosti niso več uporabljene, zna Rust z analizo živosti prepoznati, da sta npr. dve zaporedni spremenljvi izposoji varni, saj vrednost ni nikoli več spremenjena. Zato za vse primere v tem poglavju predpostavljamo, da so tako premaknjene, kot tudi izposojene spremenljivke v nadaljevanju programa še nekje uporabljene.

3.2.4 Sorodno delo

Kljub temu, da Rust v svoji dokumentaciji [34] 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 [40] formalizira delovanje analize izposoj v Rustu in dokaže, da velja izrek o varnosti (angl. soundness) za varno (angl. safe) podmnožico jezika Rust. Model RustBelt [41] poda prvi formalen in strojno preverjen dokaz o varnosti za jezik, ki predstavlja realistično podmnožico jezika Rust. Model Stacked borrows [39] definira operacijsko semantiko za dostopanje do pomnilnika v Rustu in model za prekrivanje (angl. aliasing model) in ga strojno dokaže. Model Oxide [38] je formalizacija Rustovega sistema tipov in je tudi prva semantika, ki podpira tudi neleksikalne življenjske dobe.

Poglavje 4

Implementacija lenega izračuna z uporabo 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 oziroma stroja na katerem se izvajajo. Ukazi zelo podrobno opisujejo samo izvajanje programa, 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, pri katerih je glavna operacija nad podatki ravno funkcija. Funkcijski programi so sestavljeni iz zaporedja aplikacij (angl. application) in kompozicij funkcij. V modernejših funkcijskih programskih jezikih, so funkcije višjenivojske (angl. higher order) [8], kar pomeni, da jih je mogoče prirediti v spremenljivke, jih uporabiti kot argumente in kot rezultate drugih funkcij, ter da omogočajo rekurzijo in polimorfizem. Prav tako taki jeziki pogosto omogočajo in celo spodbujajo tvorjenje novih funkcij z uporabo curryjinga oziroma delne apli-

kacije [8], pri kateri je funkciji podanih le del njenih argumentov.

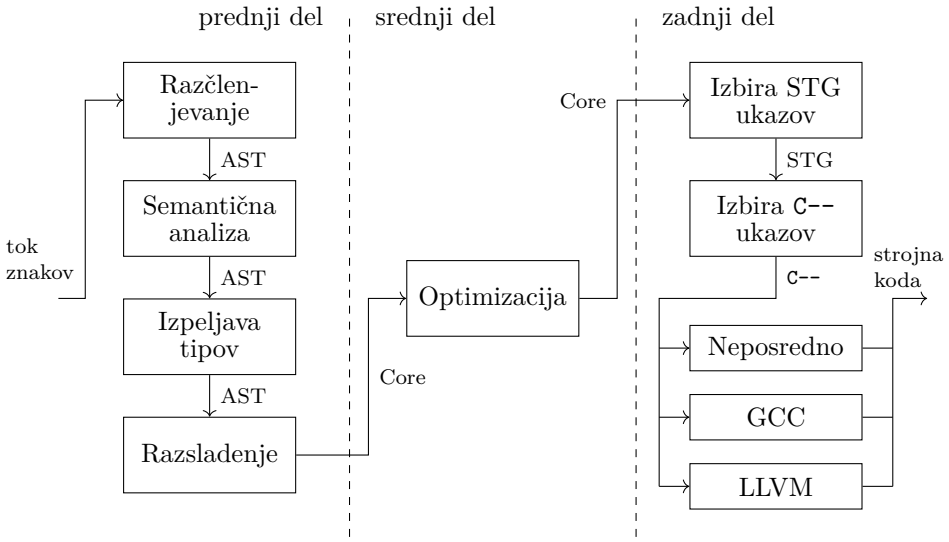
Čisti funkcijski programski jeziki spodbujajo uporabo čistih funkcij (angl. pure functions) [8]. Te za izračun vrednosti rezultata uporabljajo le vrednosti svojih argumentov, ne pa tudi drugega globalnega stanja in ne povzročajo stranskih učinkov (angl. side effects). Med take jezike spada npr. Haskell, pri katerem program opišemo kot kompozicijo čistih funkcij, ki vhod preslikajo v izhod. Vhodno-izhodne operacije so potisnjene na rob samega računanja, saj povzročajo stranske učinke, ki pa so v jeziku neželene. Stranski učinki so v jeziku predstavljeni eksplicitno z uporabo monad.

V nadaljevanju se bomo podrobneje posvetili funkcijskemu programskemu jeziku Haskell, ki temelji na čistih funkcijah in lenosti. Osredotočili se bomo na delovanje prevajalnika GHC (angl. Glasgow Haskell compiler), ki izvorno kodo 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 4.2.

4.1 Prevajalnik GHC

Prevajalnik GHC prevajanje iz izvorne kode v programskem jeziku Haskell v strojno kodo izvaja v več zaporednih fazah oziroma modulih [42, 43]. 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 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 4.1: 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.

4.2 Definicija STG jezika

Kot smo si podrobneje pogledali v poglavju 1.2, 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) [44, 45], 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) [10]. 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.

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

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

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

Vsi argumenti pri aplikaciji funkcij in primitivnih operacij so v A-normalni obliki (angl. A-normal form) [46], 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 izraz `let` $a = \oplus\ x\ y$ `in` $f\ \ a$, s čemer bi zagotovili, da so vsi argumenti atomarni.

k	$:=$	$\bullet$	neznana mestnost funkcije
	$\mid$	n	znana mestnost $n \geq 1$

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
	$\mid$	$f^k\ a_1 \ldots a_n$	aplikacija funkcije ($n \geq 1$)
	$\mid$	$\oplus\ a_1 \ldots a_n$	primitivna operacija ($n \geq 1$)
	$\mid$	`let` $x = obj$ `in` e	
	$\mid$	`case` e `of` $\{alt_1; \ldots; alt_n\}$	

Primitivne operacije so funkcije implementirane v izvajalnem okolju (angl. runtime) in so namenjene izvajanju računskih operacij nad neokvirjenimi primitivnimi podatki. Jezik podpira celoštevilske operacije `+#`, `-#`, `*#`, `/#`, in operacije nad števili s plavajočo vejico. Pri tem velja, da so vse primitivne operacije *zasičene*, kar pomeni, da sprejmejo natanko toliko argumentov, kot je mestnost (angl. arity) funkcije. Če programski jezik omogoča delno aplikacijo primitivnih funkcij, potem je potrebno take delne aplikacije z η-dopolnjevanjem razširiti v nasičeno obliko. Pri tem delno aplikacijo ovijemo v nove lambda izraze z uvedbo novih spremenljivk, ki se ne pojavijo nikjer v izrazu. Tako npr. izraz `(+ 3)`, ki predstavlja delno aplikacijo vgrajene funkcije za seštevanje prevedemo v funkcijo $\lambda x.(+\ 3\ x)$ in s tem zadostimo pogoju zasičenosti.

Izraz `let` na kopici ustvari nov objekt in je kot tak tudi edini izraz, ki omogoča alokacijo novih objektov 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.

$$
\begin{array}{rcll}
alt & \coloneqq & C\ x_1 \ldots x_n \to expr & \text{algebraična alternativa} \\
 & | & x \to expr & \text{privzeta alternativa}
\end{array}
$$

Izraz `case` je sestavljen iz podizraza e, ki se izračuna (angl. scrutinee) 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 kontinuacijo (angl. con-

tinuation), tj. naslov kjer se bo izvajanje nadaljevalo in nato začne računati vrednost izraza e. Zaradi bolj učinkovitega izvajanja, se pri prevajanju za konstruktorje vsotnih podatkovnih tipov generirajo oznake, navadno kar nenegativne celoštevilske vrednosti, ki se hranijo v objektu CON. Pri razstavljanju sestavljenega podatkovnega tipa v `case` izrazu lahko tako primerjamo cela števila in ne celih nizov.

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

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

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

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

Objekt CON predstavlja nasičeno aplikacijo konstruktorja C na argumente $a1, \ldots a_n$. Pri tem je število argumentov, ki jih prejme konstruktor natančno enako številu parametrov, ki jih zahteva.

Objekt THUNK predstavlja zakasnitev izraza e. Kadar se vrednost izraza uporabi, tj. kadar se izvede `case` izraz, se izračuna vrednost e, THUNK objekt na kopici pa se nato posodobi s preusmeritvijo (angl. indirection) na vrednost e. Pri izračunu zakasnitve se objekt THUNK na kopici zamenja z objektom BLACKHOLE, s čemer se preprečuje puščanje pomnilnika [47] in neskončnih

rekurzivnih struktur. Objekt BLACKHOLE se lahko pojavi le kot rezultat evalvacije zakasnitve, nikoli pa v vezavi v `let` izrazu.

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

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

4.3 Operacijska semantika

Operacijska semantika malih korakov bo opisana s pravili podanimi v obliki, ki jo prikazuje enačba KORAK. Pri tem s P označujemo predikate, ki morajo biti izpolnjeni, da se pravilo izvede, z e označujemo izraze (iz poglavja 4.2), oznaka s predstavlja sklad kontinuacij, H pa kopico.

$$\frac{P}{e_1;\, s_1;\, H_1 \;\Rightarrow\; e_2;\, s_2;\, H_2} \qquad \text{(KORAK)}$$

Na skladu s hranimo kontinuacije, 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. Kontinuacije so lahko ene izmed naslednjih oblik:

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

Kontinuacija $\texttt{case} \bullet \texttt{of} \; \{alt_1; \ldots; alt_n\}$ izvede glede na trenutno vrednost e natanko eno izmed alternativ $alt_1, \ldots, alt_n$. Pri kontinuaciji $\mathit{Upd}\, t \bullet$ se zakasnitev na naslovu t posodobi z vrnjeno vrednostjo, tj. trenutnim izrazom e. Zadnja kontinuacija $(\bullet \; a_1 \ldots a_n)$ pa uporabi vrnjeno funkcijo nad argumenti $a_1, \ldots, a_n$. 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 kontinuacija $(\bullet\ a_{n+1} \ldots a_{n+k})$ s k argumenti.

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

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

Dodeljevanje novih objektov na pomnilniku

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

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

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 $x_1, \ldots x_n$ zamenjajo z argumenti $a_1, \ldots, a_n$ (pravilo CASECON).

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

Če se noben izmed konstruktorjev v algebraičnih alternativah ne ujema s konstruktorjem na naslovu v, ali če je v konstanta, 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.

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

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

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

Zadnje pravilo RET se izvede, ko se izraz v `case` izrazu do konca izračuna v konstanto 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.

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

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 $v \mapsto \text{CON}(C\, a_1 \ldots a_n)$. Brisanje elementov iz kopice je implementirano s pomočjo avtomatičnega čistilca pomnilnika.

Zakasnitve in njih posodobitve

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

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

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

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

V dejanski implementaciji se pri posodobitvi na pomnilniku prejšnji objekt prepiše s preusmeritvijo. To je objekt, ki je (podobno kot ostali objekti v STG jeziku glede na sliko 4.2) 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.

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

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

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

Pri pravilih KNOWNCALL in PRIMOP gre za aplikacijo funkcije z znano mestnostjo n na natanko n argumentov. Kaj pa če argumentov ni dovolj ali pa jih je preveč? V primeru, da je argumentov premalo, gre za delno aplikacijo funkcije. Če pa je argumentov preveč, je potrebno funkcijo najprej aplicirati na ustrezno število argumentov. Ko se bo aplikacija do konca izvedla, bo rezultat funkcija, ki jo apliciramo na preostanek argumentov.

Klici funkcij z neznano mestnostjo

Pravilo EXACT je zelo podobno pravilu KNOWNCALL, le ga gre v tem primeru za funkcijo neznane mestnosti. V tem primeru je število argumentov shranjeno v tabeli metapodatkov objekta FUN na kopici, kot bomo videli v poglavju 4.4.

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

Pravili CALLK in PAP2 se ukvarjata z aplikacijami funkcij, pri katerih je število argumentov različno od njihove mestnosti. Če je število argumentov

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

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

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

Pravilo TCALL pokrije možnost, da funkcija še ni izračunana, torej da je zakasnitev. V tem primeru se prične računanje zakasnitve f, obenem pa se na sklad doda kontinuacija $(\bullet\ a_1 \ldots a_m)$, ki bo sprožila klic funkcije, ko bo vrednost zakasnitve do konca izračunana.

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

Pri pravilu PCALL gre za uporabo delne aplikacije nad preostankom argumentov. V tem primeru se funkcija g uporabi nad vsemi m argumenti. Ker je g zaradi definicije v poglavju 4.2 zagotovo funkcija, se bo v naslednjem koraku zagotovo zgodilo eno izmed pravil EXACT, CALLK ali PAP2.

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

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 kontinuacija z argumenti, se ustrezno izvede aplikacija funkcije nad argumenti.

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

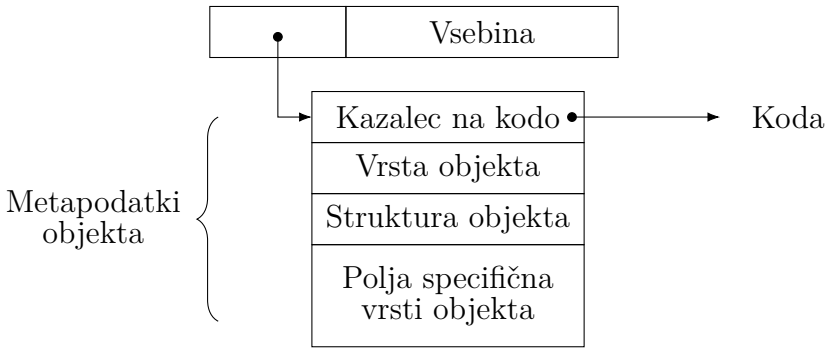
V nadaljevanju bomo na kratko opisali delovanje abstraktnega STG stroja in predstavitev objektov na pomnilniku.

4.4 Abstraktni STG stroj

Slika 4.2 prikazuje strukturo STG objektov na pomnilniku. Vsak objekt je sestavljen iz kazalca na tabelo metapodatkov in polj z vsebino objekta (angl. payload). V poljih z vsebino so shranjene proste spremenljivke, ki so lahko ali konstantne vrednosti ali pa kazalci, ki kažejo na druge objekte na kopici. Ker STG jezik podpira 6 različnih vrst objektov (tj. 5 osnovnih in preusmeritev INDIRECTION), se v tabeli metapodatkov hrani polje z vrsto objekta. Različni objekti nosijo različno vsebino. Objekt CON($C\ a_1\ \ldots\ a_n$) je npr. predstavljen s strukturo, ki ima na prvem mestu kazalec do metapodatkov objekta tipa CON, vsebina pa ima n polj, z vrednostmi argumentov $a_1, \ldots, a_n$ (ki so, kot smo videli v poglavju 4.2 atomarne vrednosti).

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

Pri implementaciji nestroge semantike v programskih jezikih obstajata



Slika 4.2: Struktura STG objektov na pomnilniku

dva glavna pristopa, ki se razlikujeta glede na način, kako se obravnavajo klici funkcij in njihovi argumenti. Modela se razlikujeta predvsem v prelaganju odgovornosti za obravnavo argumentov med klicočo in klicano funkcijo, kar vpliva na način obdelave funkcijskih klicev ter na učinkovitost izvajanja jezika.

- 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 obravnavo argumentov. 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 [45], zato je ta model tudi uporabljen v Haskellovem prevajalniku GHC, kot bomo videli v poglavju 5 pa smo ga uporabili tudi pri naši implementaciji prevajalnika.

4.5 Primer

Manjka primer izvajanja manjšega programa v STG jeziku.

Poglavje 5

Težave pri prenosu lastništva in življenjskih dob v STG

Za potrebe naše magistrske naloge smo pripravili simulator abstraktnega STG stroja[1], ki zna razčleniti kodo napisano v STG jeziku in jo izvesti. Sintaksa STG jezika temelji na tisti iz originalnega članka [45], za izračun aplikacije funkcij je uporabljen model izračunaj / uporabi, pri preverjanju pravilnosti implementacije pa smo si pomagali s spletnim simulatorjem STG jezika[2]. Namen našega dela je bil v simulator dodati analizo premikov in izposoj, kot jo poznamo v Rustu. Za osnovo smo vzeli *len* funkcijski programski jezik Blang [21], ki naj bi za upravljanje s pomnilnikom že uporabljal model lastništva in izposoje.

5.1 Implementiran prevajalnik

Slika 5.1 prikazuje zamišljeno strukturo našega prevajalnika. Z odebeljeno pisavo so označene faze, ki smo jih v naš prevajalnik uspešno implementirali. Implementacija vhodni program prebere iz datoteke, ter v fazi razčlenjevanja (angl. parsing) sestavi abstraktno sintaksno drevo. Nad drevesom se izvedeta

[1]Dostopen na naslovu `https://github.com/nikibizjak/magistrsko-delo`.
[2]Dostopen na naslovu `https://github.com/MrTipson/webstg`.

dve vrsti semantične analize:

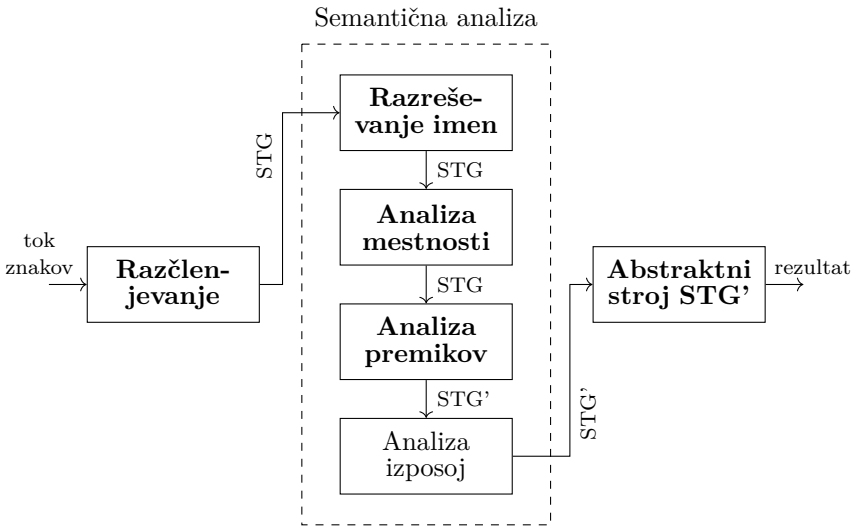
- Pri razreševanju imen (angl. name resolution) prevajalnik preveri, ali so bile vse spremenljivke definirane preden so bile uporabljene, s čemer se zagotovi, da pri izvajanju ne bo prišlo do napak zaradi dostopa do spremenljivk, ki ne obstajajo.
- Pri analizi mestnosti (angl. arity analysis) se pri aplikacijah funkcij $f^k\ a_1 \ldots a_n$ za funkcije izračuna njihovo mestnost k. Pri sestavljanju abstraktnega sintaksnega drevesa prevajalnik namreč vse aplikacije označi z neznano mestnostjo $f^\bullet\ a_1 \ldots a_n$. Za funkcije, ki so bile uvedene s pomočjo konstruktorja FUN, lahko prevajalnik mestnost določi na podlagi števila parametrov. Za tiste aplikacije funkcij, ki jim mestnosti ni mogoče določiti (npr. funkciji f v izrazu `apply = FUN(f x -> f x)`), se uporabi neznana mestnost $\bullet$, za izvajanje pa poskrbijo ustrezna pravila (EXACT, TCALL, ...) operacijske semantike.

Po semantični analizi se začne izvajanje STG programa s pomočjo abstraktnega STG stroja, ki implementira pravila operacijske semantike opisana v poglavju 4.3. Implementacija sestoji iz izraza, ki ga trenutno računamo, sklada, ki hrani nadaljevanja in kopice, na kateri so shranjeni *objekti*, ki jih STG stroj obdeluje (tj. objekti THUNK, FUN, CON, PAP, BLACKHOLE). Vsak objekt na kopici vsebuje še okolje (angl. environment), ki imena spremenljivk slika v naslove povezanih objektov na kopici. Prevajalnik zna okolja optimizirati tako, da v njih hrani le nujno potrebne spremenljivke, tj. spremenljivke, ki so v telesu objekta proste (angl. free variables).

5.2 Vrstni red izračuna izrazov

5.2.1 Neučakanost jezika Rust

Programski jezik Rust implementira strogo semantiko, kar pomeni, da se pri klicu funkcije najprej izračunajo vrednosti argumentov, ki so nato ob klicu



Slika 5.1: Predvidena struktura implementiranega prevajalnika

posredovane v telo funkcije. Prav tako se zaradi neučakanosti pri prirejanju v spremenljivko (tj. izrazu `let lhs = rhs;`), najprej do konca izračuna vrednost desnega dela prirejanja *rhs*. V splošnem velja, da če se stavek s_1 v programu pojavi leksikalno pred stavkom s_2, potem se bo stavek s_1 zagotovo izvedel pred s_2 [?] TODO: Najdi citat.

V tem poglavju bomo predpostavljali, da se dosegi spremenljivk računajo glede na leksikalen doseg. Kot smo že omenili v poglavju 3, zna Rust življenjske dobe računati neleksikalno. To pomeni, da se pri analizi življenjskih dob izvede še analiza živosti spremenljivk (angl. liveness analysis) s pomočjo katere se lahko življenjske dobe določijo bolj natančno. Posledično imajo lahko spremenljivke krajšo življenjsko dobo kot funkcija, ki jih je ustvarila. V tem poglavju bomo predpostavljali, da je spremenljivka živa od njene uvedbe, do konca bloka v katerem je bila deklarirana, oziroma do prvega (in tudi edinega) premika spremenljivke.

Spodnji primer prikazuje program v Rustu, ki vsebuje tri zaporedne klice funkcij. Zaradi neučakanosti jezika 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 izvedel pred klicem *add*.

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

Rust ✓

Ker je vrstni red operacij natanko določen, je zelo preprosto izračunati tudi življenjske dobe. V zgornjem primeru ima spremenljivka a zagotovo daljšo življenjsko dobo od spremenljivke 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.

5.2.2 Lenost jezika STG

Zgornji program bi v STG jeziku napisali na zelo podoben način. Pri deklaraciji spremenljivke z `let` izrazom, se na kopici ustvari nov objekt. Ta je lahko konstruktor algebraičnega tipa CON, funkcija FUN ali še neizračunan izraz THUNK. Objekta z delno aplikacijo PAP in črna luknja BLACKHOLE se neposredno v programu nikoli ne moreta pojaviti, temveč ju lahko na kopici ustvari le prevajalnik oziroma izvajalno okolje. V programu še upoštevamo, da vse tri funkcije *mul*, *div* in *add* kot vhodna argumenta sprejmejo uokvirjeni (angl. boxed) vrednosti. Funkcije najprej izračunajo levo vrednost in jo odpakirajo (angl. unbox), nato pa enako storijo še za desno vrednost. Pri klicu funkcij *mul*, *div* in *add* je torej potrebno vrednosti najprej uokviriti (angl. box). V našem programu so *six*, *two* in *eight* globalne spremenljivke, ki vsebujejo uokvirjena cela števila, tj. spremenljivka *two* hrani npr. kazalec na objekt `CON(Integer 2)` na kopici.

```
izracunaj = THUNK(
    let a = THUNK(mul six two) in
    let b = THUNK(div eight two) in
    let r = THUNK(add b a) in
        r
)
```

STG jezik ✓

Pri zgornjem programu se na kopici najprej ustvarijo trije novi objekti z neizračunanimi izrazi THUNK. Po definiciji jezika STG, se objekti ustvarijo v takem zaporedju, kot se v programu pojavijo, a se tukaj, za razliko od Rusta, pri prirejanju še ne izračunajo. Izračun se sproži šele ko se začne evalvacija izraza r v telesu `let` izraza. Pri tem se začne izračun zakasnitve `THUNK(add b a)`, ki glede na definicijo funkcije *add* najprej evalvira levi, nato pa še desni izraz. Pri neučakanem računu se vrednosti spremenljivk izračunajo v vrstnem redu, kot so bile definirane, tj. a, b, r, medtem ko se pri lenem izračunu vrednosti izračunajo v vrstnem redu b, a, r. Pri jezikih z lenim izračunom, je vrstni red računanja odvisen od samih implementacij funkcij, kar oteži semantične analize v prevajalniku.

Kljub temu, da je program na pogled dokaj preprost, za izračun v našem simulatorju potrebujemo 61 korakov redukcije.

5.3 Analiza premikov v STG jeziku

V poglavju 4.2 smo videli, da jezik STG sestoji iz le peščice operacij: atomarne operacije, aplikacij funkcij oziroma primitivnih operacij, izraza `let`, ki na kopici ustvari nove objekte in izraza `case`, s katerim se sproži izračun zakasnitve (angl. forcing a thunk) in glede na rezultat izvede eno izmed možnih alternativ.

Pri izrazu `let` se najprej na kopici ustvari nov objekt, nato pa se izvede še telo izraza. Spodnji primer prikazuje dva zaporedna `let` izraza, ki ustvarita spremenljivki a in b. S pravili izračuna zakasnitev je zagotovljeno, da se bo objekt, na katerega kaže spremenljivka a, na kopici ustvaril *pred* objektom,

na katerega kaže spremenljivka b.

```
main = THUNK(
    let a = THUNK(...) in -- let1
    let b = THUNK(...) in -- let2
        -- body2
)
```

STG jezik ✓

Toda, ker je jezik len, je vrstni red računanja vrednosti objektov a in b odvisen od samega izraza v telesu `body2`. Lahko se zgodi, da sta spremenljivki uporabljeni kot argumenta pri klicu funkcije, ki ju izračuna v poljubnem vrstnem redu (oziroma ju sploh ne), zaradi česar je pri prevajanju skoraj nemogoče predvideti v katerem vrstnem redu se bodo objekti evalvirali.

5.3.1 Formalna definicija analize premikov

Ta lastnost `let` izraza, nam da idejo za implementacijo analize premikov (angl. move check), ki zna zagotoviti, da na vsak objekt na kopici kaže natanko ena referenca. Pri tem upoštevamo, da se lastništvo prenese ob *ustvarjanju* objekta na kopici. Tako analizo lahko izvedemo leksikalno nad abstraktnim sintaksnim drevesom v času prevajanja.

Enačba 5.1 prikazuje obliko pravil analize premikov. Kot lahko vidimo, ta kot vhod sprejme izraz *expr* in množico trenutno veljavnih spremenljivk Δ. Množica vsebuje vse spremenljivke, do katerih lahko v danem trenutku dostopamo, tj. spremenljivk, ki še niso bile premaknjene. Rezultat analize je nova množica spremenljivk, ki so veljavne po izvajanju izraza *expr*. Če se izraz *expr* ne ujema z nobenim izmed pravil, prevajalnik vrne napako.

$$\Delta;\ expr \rightsquigarrow \Delta' \tag{5.1}$$

Enačbi INT in VAR podajata pravili za analizo premikov za atomarne operacije. Pravilo INT ne spreminja konteksta, saj izraz ne uporabi nobene spremenljivke in tako ne prenese lastništva. Pravilo VAR obravnava primer

uporabe spremenljivke, pri katerem pride do prenosa lastništva. Spremenljivka x v nadaljevanju ni več veljavna, zato je odstranjena s konteksta Δ. Pogoj $x \in \Delta$ zagotavlja, da lahko v izrazu dostopamo le do veljavnih spremenljivk. Če spremenljivka ne nastopa kot element množice Δ, se izraz ne ujema z nobenim izmed pravil, zato prevajalnik vrne napako.

$$\frac{}{\Delta;\ \texttt{n\#} \ \rightsquigarrow\ \Delta} \quad \text{(INT)}$$

$$\frac{x \in \Delta}{\Delta;\ x \ \rightsquigarrow\ \Delta \setminus \{x\}} \quad \text{(VAR)}$$

Pravili APP in PRIMOP obravnavata aplikacije funkcij oziroma primitivnih operacij. Ker je f spremenljivka, aplikacija prevzame lastništvo nad njo, prav tako pa se vrednosti argumentov premaknejo v samo funkcijo. Pri klicu funkcije namreč parametrom $x_1, \ldots, x_n$ implicitno priredimo vrednosti argumentov $a_1, \ldots, a_n$ in jih s tem *premaknemo* v telo funkcije. Podobno obravnavamo tudi primitivne operacije, le da pri tem premaknemo le argumente. Pri tem je potrebno poudariti, da s praviloma uvedemo vrstni red obdelave argumentov, ti se namreč v kontekst dodajo oziroma iz njega odstranijo od leve proti desni, ne glede na to ali so v telesu funkcije sploh uporabljeni. Z analizo premikov tako v STG jezik uvedemo vrstni red prenosa argumentov.

$$\frac{f \in \Delta \qquad \Delta_0 \coloneqq \Delta \setminus \{f\} \qquad \Delta_0;\ a_1 \rightsquigarrow \Delta_1 \qquad \ldots \qquad \Delta_{n-1};\ a_n \rightsquigarrow \Delta_n}{\Delta;\ f^k\, a_1 \ldots a_n \ \rightsquigarrow\ \Delta_n} \quad \text{(APP)}$$

$$\frac{\Delta;\ a_1 \rightsquigarrow \Delta_1 \qquad \Delta_1;\ a_2 \rightsquigarrow \Delta_2 \qquad \ldots \qquad \Delta_{n-1};\ a_n \rightsquigarrow \Delta_n}{\Delta;\ f^k\, a_1 \ldots a_n \ \rightsquigarrow\ \Delta_n} \quad \text{(PRIMOP)}$$

Enačba LET podaja pravilo za analizo premikov v `let` izrazu. Pri tem se najprej na kopici ustvari nov objekt x, ki lahko prevzame lastništvo nad spremenljivkami v trenutnem kontekstu. Izraz `let` ni rekurziven, zato v kontekst Δ ne dodamo spremenljivke x. Nadaljujemo z analizo telesa `let` izraza, ki

lahko premakne vse tiste spremenljivke, ki še niso bile premaknjene v novonastali objekt (tj. ravno spremenljivke v kontekstu Δ_1) in spremenljivko, ki je bila definirana v `let` izrazu (tj. spremenljivko x).

$$\frac{\Delta;\ obj \rightsquigarrow \Delta_1 \qquad \Delta_1 \cup \{x\};\ e \rightsquigarrow \Delta_2}{\Delta;\ \texttt{let}\ x = obj\ \texttt{in}\ e \ \rightsquigarrow\ \Delta_2} \quad (\textsc{let})$$

Enačba CASE podaja pravilo za analizo premikov `case` izraza. Pri tem se najprej izvede izraz e (angl. scrutinee), ki lahko prevzame lastništvo nad spremenljivkami v kontekstu Δ. Spremenljivke, ki niso bile premaknjene v izrazu e (tj. spremenljivke v množici Δ'), so lahko premaknjene v telesu `case` izraza, tj. eni izmed n možnih alternativ. Ker se vedno izvede *natanko ena* izmed alternativ, lahko upoštevamo, da lahko alternative premaknejo iste spremenljivke. V novem kontekstu so torej vse tiste spremenljivke, ki niso bile premaknjene v nobeni izmed alternativ.

$$\frac{\Delta;\ e \rightsquigarrow \Delta' \qquad \Delta';\ alt_1 \rightsquigarrow \Delta_1 \qquad \ldots \qquad \Delta';\ alt_n \rightsquigarrow \Delta_n}{\Delta;\ \texttt{case}\ e\ \texttt{of}\ \{alt_1; \ldots; alt_n\} \ \rightsquigarrow\ \Delta_1 \cap \cdots \cap \Delta_n} \quad (\textsc{case})$$

V nadaljevanju podamo še pravila za analizo premikov objektov *obj* in alternativ *alt*. Pri zakasnitvah (pravilo THUNK) preprosto analiziramo izraz v zakasnitvi. Konstruktorji CON prevzamejo lastništvo nad vsemi spremenljivkami, ki se pojavijo kot argumenti (pravilo CON). Pri funkcijah (pravilo FUN) pa se uvedejo nove spremenljivke (parametri $x_1, \ldots, x_n$), ki jih dodamo v kontekst in nadaljujemo z analizo telesa funkcije.

$$\frac{\Delta;\ e \rightsquigarrow \Delta'}{\Delta;\ \textsc{thunk}(e) \ \rightsquigarrow\ \Delta'} \quad (\textsc{thunk})$$

$$\frac{\Delta_0;\ a_1 \rightsquigarrow \Delta_1 \qquad \ldots \qquad \Delta_{n-1};\ a_n \rightsquigarrow \Delta_n}{\Delta_0;\ \textsc{con}(C\ a_1 \ldots a_n) \ \rightsquigarrow\ \Delta_n} \quad (\textsc{con})$$

$$\frac{\Delta \cup \{x_1, \ldots, x_n\};\ e \rightsquigarrow \Delta'}{\Delta;\ \textsc{fun}(x_1 \ldots x_n \to e) \ \rightsquigarrow\ \Delta'} \quad (\textsc{fun})$$

Objekta PAP in BLACKHOLE se nikoli ne pojavita na desni strani izraza za prirejanje. Uporabljena sta za hranjenje stanja sistema in ju kot taka

lahko na kopici ustvari samo izvajalno okolje. Ker se analiza premikov izvaja v času prevajanja, tako pravila za PAP in BLACKHOLE niso potrebna.

$$\frac{\Delta \cup \{x_1, \ldots, x_n\};\ e \rightsquigarrow \Delta'}{\Delta;\ C\ \ x_1\ \ \ldots\ \ x_n \to e\ \ \rightsquigarrow\ \ \Delta'} \qquad \text{(CONALT)}$$

$$\frac{\Delta \cup \{x\};\ e \rightsquigarrow \Delta'}{\Delta;\ x \to e\ \ \rightsquigarrow\ \ \Delta'} \qquad \text{(CONDEF)}$$

Pravila za alternative podobno kot pravila LET in FUN uvedejo nove spremenljivke, ki jih dodamo v kontekst, nato pa nadaljujemo z analizo premikov izraza e v alternativi.

5.3.2 Lastništvo objektov

S pravili analize premikov je v jeziku zagotovljeno, da je vsaka spremenljivka uporabljena *največ enkrat* (tj. nič ali enkrat), kar ustreza sistemu afinih tipov tipov [22]. Če program uspešno prestane analizo lastništva, torej velja, da na vsak objekt na kopici kaže največ en kazalec. V tem poglavju bomo opisali sistem lastništva, ki je uporabljen med samim izvajanjem programa in skrbi za ustrezno čiščenje pomnilnika. Pri sistemu lastništva se z neokvirjenimi (angl. unboxed) vrednostmi ne bomo ukvarjali, ker lahko te navadno spravimo v en register in je posledično njihovo kopiranje zelo preprosto in učinkovito.

Pri sistemu lastništva obravnavamo dva ločena pojma: spremenljivke in objekte. Objekti *obj* so strukture (oziroma ovojnice) na kopici, ki so bile definirane v poglavju 4.2. Vsak objekt je lahko ena izmed petih možnosti (THUNK, FUN, CON, PAP ali BLACKHOLE) in je sestavljen iz kode, ki se izvede ob evalvaciji objekta, in okolja v katerem se hranijo vrednosti prostih spremenljivk. Pri uvedbi sistema lastništva vsak objekt na kopici razširimo še s seznamom kazalcev na objekte, ki si jih le-ta lasti. V programskem jeziku Blang [21] je tak seznam imenovan za *seznam močnih kazalcev*, zato bomo tako terminologijo uporabljali tudi v našem delu. Med izvajanjem kode objekta, lahko pride do prenosa lastništva, pri čemer je potrebno se-

znam močnih kazalcev dopolniti z novo vrednostjo, oziroma kakšen kazalec odstraniti. Kadar se objekt do konca izračuna (npr. ko se THUNK do konca evalvira), se sprosti pomnilnik vseh objektov na katere kažejo močni kazalci. Pri tem pa si tudi objekti na seznamu močnih kazalcev lastijo druge objekte, zaradi česar je potrebno operacijo sproščanja izvajati rekurzivno: vsak objekt na katerega kaže seznam iz močnih kazalcev najprej sprosti pomnilnik vseh objektov, ki si jih lasti, nato pa odstrani še sam sebe. V učinkoviti implementaciji bi vsak objekt poleg kode za izračun, vseboval še kodo za čiščenje pomnilnika, ki bi poskrbela za odstranjevanje vseh vrednosti, ki si jih lasti.

Kadar nek objekt ustvari novo spremenljivko x, tj. z `let` izrazom, se na kopici ustvari nov objekt, njegov naslov pa se shrani v spremenljivko x. Ker je objekt spremenljivko ustvaril, je posledično tudi lastnik objekta na katerega spremenljivka kaže, zato se ob prirejanju v spremenljivko, v seznam močnih kazalcev doda še naslov objekta na katerega spremenljivka kaže.

V nadaljevanju bomo neformalno opisali pravila za prenašanje lastništva. Pri tem bomo upoštevali, da vsaka *uporaba* neke spremenljivke preda lastništvo objekta, na katerega spremenljivka kaže nekemu drugemu objektu.

- Pri aplikaciji $f^k\ a_1 \ldots a_n$ so argumenti atomarni, kar pomeni, da so ali neokvirjene vrednosti, pri katerih se z lastništvom ne ukvarjamo ali pa spremenljivke, ki kažejo na druge objekte na kopici. Lastništvo objektov na katere kažejo argumenti $a_1, \ldots, a_n$ se preda objektu na katerega kaže spremenljivka f.
- Pri primitivnih operacijah $\oplus\ a_1 \ldots a_n$, operator $\oplus$ prevzame lastništvo nad objekti na katere kažejo argumenti $a_1, \ldots, a_n$.
- Pri izrazu `let` $x = obj$ `in` e se na kopici najprej ustvari objekt obj, ki prevzame lastništvo nad vsemi prostimi spremenljivkami, ki se v njem pojavijo. Izraz `let` uvede novo spremenljivko x, pri čemer trenutni objekt prevzame lastništvo nad novo ustvarjenim objektom obj. Po prenosu lastništva sledi izračun telesa e.
- Pri `case` izrazu se najprej izračuna vrednost izraza e, kar lahko vodi do

prenosa lastništva. Rezultat izračuna je naslov v konstruktorja CON, ki se ujema z eno izmed alternativ. Če so vsi dosedanji izrazi lastništvo predali, pa ga `case` izraz prevzame. Trenutnemu objektu, se v seznam lastništva dodajo kazalci na vse objekte, ki si jih objekt v lasti. Objekt na katerega kaže v, lahko izbrišemo iz pomnilnika, saj nanj zagotovo kaže le kazalec v.

Sprememba operacijske semantike

Operacijsko semantiko je potrebno dopolniti s pravili za čiščenje pomnilnika. V nadaljevanju neformalno opišemo nekaj najpomembnejših sprememb operacijske semantike.

- Pravilo UPDATE posodobi objekt THUNK na kopici s preusmeritvijo INDIRECTION, ki kaže na objekt z rezultatom. Ker je pri tem THUNK objekt uničen, lahko pri tem počistimo pomnilnik vseh objektov, ki si jih le-ta lasti.
- Pri pravilu CASECON trenuten objekt prevzame lastništvo nad vsemi argumenti konstruktorja. Zatem je lahko pomnilnik CON objekta sproščen, saj s tem vrednosti argumentov niso izbrisane pred uporabo.
- Pri pravilu PRIMOP najprej izračunamo rezultat primitivne operacije, nato pa lahko počistimo pomnilnik vseh objektov, ki so bili uporabljeni za izračun rezultata.

Primer

Naslednji program prikazuje zakasnitev t, ki ustvari dva nova objekta na kopici in ju priredi spremenljivkama a in b. Pri prirejanju v spremenljivko r, sta spremenljivki a in b v telesu funkcije *prosti*, kar pomeni, da se lastništvo (in s tem odgovornost za čiščenje pomnilnika) prenese v objekt FUN. Po premiku postaneta spremenljivki a in b neveljavni, kar pomeni, da se v telesu zakasnitve ne smeta več pojaviti. Telo najbolj ugnezdenega `let` izraza iz same zakasnitve t vrne tudi lastništvo nad spremenljivko r. To pomeni, da

THUNK na katerega kaže spremenljivka t, ni več odgovoren za čiščenje pomnilnika nobene izmed spremenljivk, ki jih je definiral. Če program vsebuje kakršnokoli prirejanje, kjer se spremenljivka t pojavi na desni strani (angl. right-hand-side), potem objekt v katerem se prirejanje nahaja, prevzame lastništvo nad objektom FUN($f \to f\ a\ b$).

```
t = THUNK(
    let a = THUNK(...)      in
    let b = THUNK(...)      in
    let r = FUN(f -> f a b) in  -- Premik
        r
)
```

STG jezik s sistemom lastništva ✓

Ena izmed optimizacij, ki jo lahko prevajalnik izvede, da s pomočjo analize prostih spremenljivk prepozna katere izmed spremenljivk, definiranih v nekem objektu (z `let` izrazom) v nadaljevanju programa niso nikjer več uporabljene. Take definicije lahko prevajalnik iz programa odstrani ter s tem zmanjša količino objektov, ki se ustvarijo oziroma brišejo iz kopice.

Prednosti in slabosti

Prednost jezika STG, ki vsebuje le premike (tj. zgolj sistem lastništva) je zmožnost izvajanja avtomatičnega čiščenja pomnilnika brez uporabe tradicionalnega avtomatičnega čistilca pomnilnika. Vsak objekt na kopici dopolnimo s seznamom objektov, ki si jih le-ta lasti (tj. seznamom močnih kazalcev), operacijsko semantiko jezika pa dopolnimo tako, da se ob posodobitvah zakasnitev sprosti pomnilnik za objekte, ki so v seznamu. Kot smo že omenili, v jezik tako uvedemo sistem *afinih* tipov, ki zagotavlja, da na vsak objekt kaže največ ena referenca, oziroma da je lahko vsaka spremenljivka uporabljena *največ enkrat*. Če spremenljivka v nekem objektu ni nikjer uporabljena, potem lahko objekt, na katerega se le-ta sklicuje, izbrišemo s pomnilnika. Če je spremenljivka nekje uporabljena, potem se lastništvo prenese na nek drug objekt, ki postane odgovoren tudi za čiščenje pomnilnika. Z uvedbo sistema

afinih tipov, pa v STG vpeljemo tudi omejitve afinega sistema tipov. V takem jeziku lahko vsako spremenljivko (oziroma vsak objekt) uporabimo največ enkrat, kar pomeni, da ne moremo napisati naslednjega programa.

```
main = THUNK(
    let a = ... in
    let b = CON(Pair a a) in -- (a, a)
        ...
)
```

STG jezik s sistemom lastništva ✗

Ker STG jezik obravnava funkcije kot prvorazredne objekte, lahko spremenljivke hranijo reference na funkcije. To pa pomeni, da v afinem sistemu izgubimo možnost večkratne uporabe funkcij (tj. v takem jeziku ne smemo napisati izraza `f (f x)`, ki funkcijo f uporabi dvakrat), prav tako pa nam tak sistem tipov prepoveduje uporabo rekurzije.

5.4 Globoko kloniranje objektov

Glavna težava, ki se pojavi pri sistemu, ki vsebuje le princip lastništva, je nezmožnost deljenja objektov. Ena izmed možnih rešitev (implementirana tudi v delu [28]) je vpeljava izraza `clone`, ki zna globoko klonirati (angl. deep clone) objekt na kopici. Pri globokem kopiranju se ustvari nova kopija objekta, vključno z vsemi njegovimi notranjimi strukturami in podatki. To pomeni, da se rekurzivno kopirajo tudi vsi ugnezdeni objekti in podatkovne strukture, ki jih vsebuje izvorni objekt. Rezultat je popolnoma neodvisna kopija originalnega objekta, kjer spremembe na kopiji ne vplivajo na original in obratno.

S pomočjo analize premikov lahko tako za vsak objekt zagotovimo, da na vsak objekt kaže natanko ena referenca, s pomočjo kloniranja pa lahko objekte podvajamo, ne da bi to predpostavko prekršili. Slabost takega pristopa je v tem, da je pri kloniranju velikih oziroma bolj kompleksnih objektov, postopek kloniranja izredno časovno in prostorsko potraten [24, 48]. Do

druge težave pride zaradi same lenosti izračuna.

```
main = THUNK(
    let a = ... in
    let b = THUNK(clone a) in
        (f a) + b
)
```

STG jezik z globokim kloniranjem ✗

Zgornji primer prikazuje program, ki bi se v neučakanem jeziku izvedel brez težav. Ob prirejanju v spremenljivko `b`, bi se namreč celoten objekt na kopici, na katerega kaže spremenljivka `a`, kopiral, kar pomeni, da sta objekta na katera kažeta spremenljivki `a` in `b` povsem disjunktna (tj. njuna grafa na pomnilniku si ne delita nobenega vozlišča). Funkcija `f` prevzame lastništvo nad `a` in s tem tudi odgovornost za čiščenje pomnilnika. Če se v izrazu `(f a) + b` torej najprej izvede aplikacija `f a`, potem se pomnilnik spremenljivke a tudi ustrezno počisti. Toda, spremenljivka b kaže na zakasnitev, ki pa še ni bila izračunana, ta pa vsebuje referenco na sedaj neobstoječo spremenljivko a, kar pomeni, da pride pri izvajanju programa do napake.

Če bi želeli, da se program pravilno izračuna, bi torej morali v STG jezik ponovno uvesti neučakanost pri računanju, na podoben način kot je to pri `let!` izrazu Wadlerjevem jeziku s sistemom steadfast tipov [24]. Pri tem se desna stran prirejanja izračuna *v celoti*, preden se izvede telo izraza, kar Wadler imenuje za hiperstriktni izračun (angl. hyperstrict evaluation). Ker je leni izračun osnova STG jezika, ta pristop ne pride v poštev.

5.5 Izposoja

Druga rešitev za ponovno uvedbo deljenja izrazov v programskem jeziku s sistemom lastništva temelji na konceptu izposoje, kot ga poznamo iz programskega jezika Rust. Implementiran sistem lastništva zahteva, da je vsaka spremenljivka uporabljena največ enkrat. Koncept izposoje pa omogoča, da je spremenljivka lahko nespremenljivo izposojena (in s tem uporabljena)

večkrat, ali pa spremenljivo izposojena le enkrat. Programski jezik Rust [34] uvede koncept življenjskih dob, s pomočjo katerih v fazi analize izposoj zagotovi, da je življenjska doba objekta daljša od življenjske dobe izposoje. Len funkcijski jezik Blang [21] na podlagi Rustovega modela lastništva implementira tako analizo premikov, kot tudi analizo izposoj. Jezik (podobno kot npr. Ocaml in Scala) podpira mutacijo in na podlagi principa izključitve v Rustu, tudi spremenljive in nespremenljive izposoje. Podobno kot Rust, tudi Blang definira pojem življenjskih dob, ki jih zmore v določenih primerih prevajalnik izračunati sam, v drugih primerih pa zahteva, da programer življenjske dobe in tipe navede eksplicitno.

Mutacija v STG jeziku

Pri STG jeziku lahko nove spremenljivke definiramo s pomočjo `let` izraza, ki na kopici ustvari nov objekt, v spremenljivko pa se shrani pomnilniški naslov le-tega. Jezik STG ne vsebuje nobenega konstrukta, s pomočjo katerega bi obstoječi spremenljivki priredili drugo vrednost. Pri definiciji spremenljivke z enakim imenom, se prejšnja vrednost ne prepiše, temveč zasenči (angl. shadow). Na kopici se ustvari nov objekt, pri čemer se vse nadaljnje uporabe spremenljivke x sklicujejo na novonastali objekt, pri vseh objektih, ki so nastali prej, pa spremenljivka vsebuje naslov prejšnjega objekta. Jezik STG torej ne omogoča neposredne mutacije vrednosti obstoječih spremenljivk.

Jezik STG učinkovito implementacijo lenega izračuna doseže s pomočjo principa prenosa po potrebi (angl. call by need), pri katerem se vrednost zakasnitve izračuna prvič, ko je potrebovana, nato pa se v pomnilniku objekt zamenja s preusmeritvijo na rezultat. Ob vsaki nadaljnji uporabi, se vrednost enostavno prebere s pomnilnika. Torej so edini vir mutacij v STG jeziku posodobitve zakasnitev (angl. thunk updates). Rust podpira spremenljive in nespremenljive izposoje, ker pa v STG jeziku ni mutacije, se bomo v našem delu ukvarjali le z nespremenljivimi izposojami.

V poglavju 5.3 smo definirali analizo premikov pri kateri pride pri uporabi spremenljivke do premika in s tem do prenosa odgovornosti za čiščenje

pomnilnika premaknjenega objekta. Pri nespremenljivi izposoji, ki jo bomo označevali z `&x`, je omogočen dostop do spremenljivke x, pri tem pa se *ne* prenese odgovornost za čiščenje pomnilnika. Podobno kot pri premikih, bomo tudi pri izposojah predpostavljali, da do izposoje pride ob prirejanju in ne ob dejanski uporabi spremenljivke. Naslednji primer prikazuje program, ki najprej na kopici ustvari objekt in ga priredi spremenljivki a, nato pa se na kopici ustvarita še dve zakasnitvi, ki si izposodita spremenljivko a.

```
1  main = THUNK(
2      let a = ... in
3      let b = THUNK(... &a ...) in -- Izposoja
4      let c = THUNK(... &a ...) in -- Izposoja
5          ...
6  )
```

STG jezik s sistemom lastništva in izposoje

Po pravilih lastništva, je zakasnitev *main*, lastnik spremenljivk a, b in c, kar pomeni, da lahko ob posodobitvi, počisti tudi pomnilnik spremenljivk a, b in c. Objekta, na katera kažeta spremenljivki b in c, si izposojata spremenljivko a. Ker je spremenljivka a prosta znotraj zakasnitev, se pojavi v okolju znotraj ovojnice obeh zakasnitev, ker pa je spremenljivka izposojena in ne premaknjena, pa se *ne* pojavi v seznamu močnih kazalcev zakasnitev in tako njen pomnilnik ni počiščen ob posodobitvi zakasnitev.

Uvedba izposoje v STG

Za uvedbo izposoje v jezik STG, je najprej potrebno razširiti semantiko jezika z dodajanjem operacije za izposojo.

$$a, v \quad := \quad \ldots \mid \&x \qquad\qquad \text{atomarni izrazi}$$

V analizo premikov dodamo pravilo BORROW, ki deluje na podoben način kot pravilo VAR, le da pri tem iz konteksta Δ ne odstrani spremenljivke, saj pri izposoji vrednost ni premaknjena.

$$\frac{x \in \Delta}{\Delta;\, \&x \;\rightsquigarrow\; \Delta} \quad \text{(BORROW)}$$

Naslednji primer prikazuje program, ki najprej ustvari spremenljivko a, nato pa ustvari spremenljivko b in pri tem prenese lastništvo v objekt THUNK, na katerega kaže spremenljivka b. Pri tem spremenljivka a ni več veljavna in je zato pri analizi premikov odstranjena iz konteksta Δ. Pri analizi objekta na katerega kaže c, pride tako do napake, saj se kontekst Δ in izraz $\&a$ ne ujemata s pravilom BORROW (ker velja $a \notin \Delta$).

```
main = THUNK(
    let a = ... in
    let b = THUNK(... a ...) in
    -- Neveljavna izposoja, spremenljivka a je bila že premaknjena
    let c = THUNK(... &a ...) in
        ...
)
```

STG jezik s sistemom lastništva in izposoje ✗

Naslednji primer prikazuje program, ki bi uspešno prestal analizo premikov. Objekt b si namreč spremenljivko a le izposodi in je ne odstrani iz konteksta Δ. Objekt c spremenljivko a dejansko premakne in jo odstrani iz konteksta, pri čemer ta postane neveljavna v telesu `let` izraza.

```
main = THUNK(
    let a = ... in
    let b = THUNK(... &a ...) in
    -- Neveljaven premik, spremenljivka a je že bila izposojena
    let c = THUNK(... a ...) in
        ...
)
```

STG jezik s sistemom lastništva in izposoje ✓

Tak program bi v Rustu javil napako, saj spremenljivka ne sme biti premaknjena, dokler je izposojena (tj. dokler nanjo obstaja živa referenca). V Rustu lahko življenjsko dobo izposoj upravljamo s pomočjo blokov. Nov blok namreč uvede novo življenjsko dobo, spremenljivke definirane znotraj bloka

pa so veljavne do konca bloka. Kadar se blok zaključi, se pomnilnik za spremenljivke sprosti, izposoje pa se zaključijo. V naslednjem programu uvedemo spremenljivko a in si jo znotraj bloka izposodimo v spremenljivko b. Kadar se blok zaključi, se zaključi tudi izposoja, zato lahko nato izvedemo premik v spremenljivko c.

```
let a = ...;
{
    let b = &a;
}
let c = a;
```

Rust ✓

Jezik STG nima koncepta blokov (tj.). Z izrazom `let` lahko ustvarimo nove zakasnitve, ki se izračunajo po potrebi, izraz `case` pa je edini izraz, ki sproži izračun neke zakasnitve (angl. forcing a thunk). Je res? Torej je `case` izraz tudi edini način, da se izposoja zaključi. Naslednji primer prikazuje zakasnitev, v kateri definiramo spremenljivko a in si jo izposodimo v spremenljivko b. Nato sprožimo izračun zakasnitve b s pomočjo `case` izraza. Odvisno od izraza v zakasnitvi THUNK, je lahko spremenljivka a še vedno izposojena (če je bila vrednost `&a` uporabljena pri tvorjenju rezultata). Če spremenljivka a ni več izposojena (tj. ni bila uporabljena pri tvorjenju rezultata zakasnitve), potem jo lahko premaknemo.

```
main = THUNK(
    let a = ... in
    let b = THUNK(... &a ...) in
    case b of {
        x ->
            -- Premik je veljaven samo, če pri tvorjenju rezultata
            ↪  'x' ni bila uporabljena vrednost '&a'.
            let c = THUNK(... a ...) in c
    }
)
```

STG jezik s sistemom lastništva in izposoje

Na tem mestu se pojavi vprašanje, ali je mogoče izposoje prepoznati med samim prevajanjem programa.

Zakaj je nujna uvedba podatkovnih tipov

V naslednjem primeru funkcija f ni znana, kar pomeni, da ne vemo, ali bo spremenljivka *result* vsebovala referenco na *item*. Če bo, potem je program neveljaven, ker se bo po izvajanju funkcije f počistil pomnilnik spremenljivke *item* (ker ni bila premaknjena), rezultat pa bo vseboval referenco na neobstoječo spremenljivko.

```
1  g = FUN(f ->
2      let item = CON(Integer 10) in
3      case f &item of {
4          x ->
5              let result = CON(Just x) in
6                  result
7      }
8  )
9
10 f1 :: &'A a -> 'A (Maybe b)
11 f1 = FUN(x ->
12     let result = CON(Nothing) in
13         result
14 )
15
16 f2 :: &'A a -> &'A a
17 f2 = FUN(x -> x)
```

STG jezik s sistemom lastništva in izposoje

Pri aplikaciji funkcije `g f1`, je program povsem veljaven, saj je pomnilnik vseh objektov ustrezno počiščen. V primeru aplikacije `g f2`, pa bo rezultat vseboval referenco na spremenljivko *item*, ki pa bo izbrisana po izhodu iz funkcije g.

Še en primer, ki je veljaven, če je f funkcija ki kot vhod sprejme dve referenci in vrne nek nov objekt skupaj z lastništvom. Če bi bil rezultat

funkcije f sestavljen z uporabo referenc, bi bil program neveljaven.

```
g :: (Pair (&'A a) (&'B b) -> c) -> c
g = FUN(f ->
    let item = CON(Integer 10) in
    let pair = CON(Pair &item &item) in
        f pair
)

-- Veljavna funkcija
f1 :: (Pair (&'A a) (&'B b) -> Integer) -> Integer
f1 = FUN(x y -> let r = CON(Integer 10) in r)
```

STG jezik s sistemom lastništva in izposoje

Za analizo izposoj, je torej nujno potrebno v jezik STG uvesti podatkovne tipe. V poglavju 3 smo omenili, da tudi Rust ne zna sam izpeljati življenjskih dob pri funkcijah z več kot enim argumentom. Naša razširitev STG jezika bi podobno kot Rust od programerja zahtevala, da za vsako funkcijo in vsak objekt ustvarjen z `let` izrazom, napiše še prototip (angl. type annotation).

```
-- Operacija 'add' sprejme dve referenci, ustvari nov objekt z
↪  rezulatom in ga vrne (in s tem vrne tudi lastništvo)
add :: &'A Integer -> &'B Integer -> Integer
add = FUN(x y -> ...)

main = THUNK(
    let a = CON(Integer 12) in
    let b = THUNK(add &a &a) in
    -- Tukaj ne moremo premakniti 'a', ker je izposojena
    -- let c = THUNK(a) in
    case b of {
        result ->
            -- Na tem mestu zakasnitev 'main' prevzame lastništvo
            ↪  nad objektom CON(Integer x), ki jo je ustvarila
            ↪  funkcija 'add'.
            -- Prav tako pa vemo, da je zakasnitev 'b' do konca
            ↪  izračunana in ne usebuje več referenc na 'a' (tj.
            ↪  izposoja 'a' je zaključena)
            -- => Tukaj lahko spremenljivko 'a' ponovno premaknemo
            let d = THUNK(a) in
                ...
    }
```

STG jezik s sistemom lastništva in izposoje

```
id :: &'A a -> &'A a
id = FUN(x -> x)

ignore :: &'A a -> b
ignore = FUN(x -> let r = CON(Integer 10) in r)

main = THUNK(
    let a = CON(Integer 12) in
    let b = THUNK(id &a) in -- Se še ne izračuna
    let c = CON(Just b) in
    case c of {
        -- Izraz case izračuna do konca zakasnitve, konstruktorje
        ↪ CON pa spravi v a normalno obliko.
        Just value ->
            -- Tukaj ne smemo premakniti spremenljivke 'a', ker je
            ↪ izposoja še vedno aktivna. Dokler se namreč ne
            ↪ izračuna vrednost argumenta (Just value), ta
            ↪ vsebuje kazalec na spremenljivko 'a'.
            case (ignore value) of {
                result ->
                    -- Na tem mestu izposoja ni več veljavna, spet
                    ↪ lahko premaknemo spremenljivko 'a'.
                    let d = THUNK(a) in
                        ...
            }
    }
```

STG jezik s sistemom lastništva in izposoje

Poglavje 6

Zaključek

V sklopu magistrskega dela nam je uspelo implementirati delujoč abstraktni STG stroj. Ugotovitve našega magistrskega dela: (alineje je potrebno prepisati v enotno besedilo)

- V jezik lahko dodamo analizo premikov, s katero zagotovimo, da je vsaka spremenljivka lahko uporabljena največ enkrat. S pomočjo analize v jezik uvedemo lastništvo objektov, ki nam omogoča avtomatično sproščanje pomnilnika.
- Problem takega jezika je, da postane dokaj neuporaben. Vsak objekt (tj. spremenljivka, funkcija, ...) je lahko uporabljen največ enkrat, kar pomeni, da izgubimo možnost deljenja, ki nam omogoča učinkovito implementacijo lenega izračuna.
- Ena izmed možnosti s katero lahko v jezik ponovno uvedemo podvojevanje objektov je globoko kloniranje, s pomočjo katerega se podvoji celotna struktura na kopici. Problem globokega kloniranja je v neučinkovitosti kopiranja velikih objektov, prav tako pa je za pravilno implementacijo potrebno v len jezik ponovno uvesti neučakani izračun, kar za STG jezik ni sprejemljivo.
- Druga možnost temelji na konceptu izposoje iz programskega jezika Rust [34] in je bila implementirana v programski jezik Blang [21]. Tu-

kaj v jezik dodamo poseben operator, ki omogoča izposojo objektov. Če pride pri premiku do spremembe lastništva in posledično tudi do prenosu odgovornosti za čiščenje pomnilnika, potem se pri izposoji lastništvo ne preda.

- Za implementacijo izposoj je potrebno uvesti (oziroma izračunati) življenjske dobe objektov. Zaradi same lenosti STG jezika pa je življenjske dobe praktično nemogoče izračunati. Že sam vrstni red računanja izrazov je težko predvideti, saj je odvisen od dejanske implementacije funkcij.
- Ena izmed možnih rešitev (za katero nismo prepričani, če bi sploh delovala), je ponovna uvedba oznak tipov (prototipov oziroma angl. type annotations), ki bi jih razširili z življenjskimi dobami, s katerimi bi moral programer označiti funkcije in tako prevajalniku povedati, kakšne so odvisnosti med življenjskimi dobami parametrov in rezultata funkcije.
- Problem takega pristopa je, da je izredno kognitivno naporen za samega programerja in da povzroči uvedbo tipov v STG jezik, kar pa ni v duhu magistrskega dela, saj jezika nismo želeli povsem spremeniti.
- V splošnem uvedba principa lastništva in izposoje na podlagi tiste iz Rusta za lene programske jezike ni izvedljiva oziroma smiselna.

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

[7] R. W. Sebesta, Concepts of programming languages, Pearson Education India, 2004.

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

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

[14] D. Syme, The F# compiler technical overview (3 2017).

[15] 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.

[16] 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.

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

[18] U. Boquist, Code optimization techniques for lazy functional languages, Ph.D. thesis, Chalmers University of Technology, Sweden (April 1999).

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

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

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

[22] B. C. Pierce, Advanced topics in types and programming languages, MIT press, 2004.

[23] P. Wadler, Is there a use for linear logic?, ACM SIGPLAN Notices 26 (9) (1991) 255–273.

[24] P. Wadler, Linear types can change the world, Ifip Tc (1990).

[25] 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.

[26] J.-Y. Girard, Linear logic, Theoretical computer science 50 (1) (1987) 1–101.

[27] J.-P. Bernardy, M. Boespflug, R. R. Newton, S. Peyton Jones, A. Spiwack, Linear haskell: practical linearity in a higher-order polymorphic language, Proc. ACM Program. Lang. 2 (POPL) (dec 2017).

[28] D. Marshall, D. Orchard, Functional ownership through fractional uniqueness, Proc. ACM Program. Lang. 8 (OOPSLA1) (apr 2024).

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] J. Boyland, Checking interference with fractional permissions, in: International Static Analysis Symposium, Springer, 2003, pp. 55–72.

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

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

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

[37] 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`

[38] A. Weiss, O. Gierczak, D. Patterson, A. Ahmed, Oxide: The essence of rust (2021). `arXiv:1903.00982`.

[39] 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.

[40] 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).

[41] 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).

[42] 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, proceedings of joint framework for information technology technical conference, keele Edition, DTI/SERC, 1993, pp. 249–257.

[43] A. Brown, G. Wilson, The Architecture of Open Source Applications, Volume II, The Achrictecture of Open Source Applications, CreativeCommons, 2012.

[44] 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.

[45] 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.

[46] 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.

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

[48] Y. Lafont, The linear abstract machine, Theoretical computer science 59 (1-2) (1988) 157–180.