Niki Bizjak
Cesta 1, 1234 Mesto, Slovenija
Študijski program: Računalništvo in informatika, MAG
Vpisna številka: 60606060

Komisija za študijske zadeve
Univerza v Ljubljani, Fakulteta za računalništvo in informatiko
Večna pot 113, 1000 Ljubljana

Vloga za prijavo teme magistrskega dela
Kandidat: Niki Bizjak

Niki Bizjak, študent/-ka magistrskega programa na Fakulteti za računalništvo in informatiko, zaprošam Komisijo za študijske zadeve, da odobri predloženo temo magistrskega dela z naslovom:

Slovenski: **Lastništvo objektov namesto avtomatskega čistilca pomnilnika med lenim izračunom**
Angleški: **Ownership model instead of garbage collection during lazy evaluation**

Tema je bila že potrjena lani in je ponovno vložena: ***NE***

Izjavljam, da so spodaj navedeni mentorji predlog teme pregledali in odobrili ter da se z oddajo predloga strinjajo.

Magistrsko delo nameravam pisati v slovenščini.

Za mentorja/mentorico predlagam:

Ime in priimek, naziv: Boštjan Slivnik, doc. dr.
Ustanova: Fakulteta za Računalništvo in Informatiko, Ljubljana
Elektronski naslov: bostjan.slivnik@fri.uni-lj.si

V Ljubljani, 4. december 2023.

PREDLOG TEME MAGISTRSKEGA DELA

1 Področje magistrskega dela

slovensko: prevajalniki, funkcijski programski jeziki, nestrog izračun, upravljanje s pomnilnikom
angleško: compilers, functional programming languages, lazy evaluation, memory management

2 Ključne besede

slovensko: prevajalnik, nestrog izračun, upravljanje s pomnilnikom, avtomatični čistilec pomnilnika, lastništvo objektov
angleško: compiler, lazy evaluation, memory management, garbage collector, ownership model

3 Opis teme magistrskega dela

Pretekle potrditve predložene teme:
Predložena tema ni bila oddana in potrjena v preteklih letih.

3.1 Uvod in opis problema

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

Pri avtomatičnem upravljanju s pomnilnikom zna sistem sam dodeljevati in sproščati pomnilnik. Tukaj ločimo posredne in neposredne metode. Ena izmed neposrednih metod je npr. štetje referenc [2], pri kateri za vsak objekt na kopici hranimo metapodatek o

številu kazalcev, ki se sklicujejo nanj. V tem primeru moramo ob vsakem spreminjanju referenc zagotavljati še ustrezno posodabljanje števcev, kadar pa število kazalcev pade na nič, objekt izbrišemo iz pomnilnika. 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].

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

3.2 Pregled sorodnih del

Leni funkcijski programski jeziki funkcije obravnavajo kot prvorazredne objekte (angl. first-class objects), kar pomeni, da so lahko funkcije argumenti drugim funkcijam in da lahko funkcije kot rezultate vračajo druge funkcije. Taki jeziki pogosto omogočajo in spodbujajo tvorjenje novih funkcij z uporabo delne aplikacije [6], pri kateri je funkciji podanih le del njenih argumentov. Leni funkcijski programski jeziki uporabljajo nestrogo semantiko, ki deluje na principu prenosa po potrebi (angl. call-by-need) [7], pri kateri se pri klicu funkcij ne izračuna najprej vrednosti argumentov, temveč se računanje izvede šele takrat, ko telo funkcije vrednost dejansko potrebuje. Nestroga semantika je običajno implementirana z ovijanjem izrazov v zakasnitve (angl. thunks) [8], tj. funkcije brez argumentov, ki se evalvirajo šele, kadar je njihova vrednost dejansko zahtevana.

Ker je v funkcijskih jezikih lahko funkcija vrednost argumenta ali rezultata, je lahko izvajanje take funkcije zamaknjeno v čas po koncu izvajanja funkcije, ki je ustvarila vrednost argumenta ali rezultata. Zato klicnih zapisov takih funkcij ni mogoče hraniti na skladu, temveč na kopici [1]. Na kopici so zakasnitve in funkcije shranjene kot *zaprtja* (angl. closures), tj. podatkovne strukture, v katerih se poleg kode hranijo še kazalci na podatke, ki so zahtevani za izračun telesa. Pri izvajanju se tako na kopici nenehno ustvarjajo in brišejo nova zaprtja, ki imajo navadno zelo kratko življenjsko dobo, zato je nujna učinkovita implementacija dodeljevanja in sproščanja pomnilnika. Haskell za to uporablja *generacijski* avtomatični čistilec pomnilnika [9, 10]. Danes vsi večji funkcijski programski jeziki, ki omogočajo leni izračun, uporabljajo avtomatični čistilec pomnilnika [11, 12, 13, 14, 15].

Lene funkcijske programske jezike najpogosteje implementiramo s pomočjo redukcije grafa [8]. Eden izmed načinov za izvajanje redukcije je abstraktni STG stroj (angl. Spineless Tagless G-machine) [16], 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].

Ena izmed alternativ STG stroja za izvajanje jezikov z nestrogo semantiko je prevajalnik GRIN [17] (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) [18]. Za upravljanje s pomnilnikom se v trenutni različici uporablja čistilec pomnilnika [19].

Programski jezik Rust za upravljanje s pomnilnikom uvede princip lastništva (angl. ownership model) [5], pri katerem ima vsak objekt na kopici natančno enega *lastnika* [20, 21, 22]. Kadar gre spremenljivka, ki si lasti objekt, izven dosega (angl. out of scope), se pomnilnik za objekt sprosti. Rust definira pojem *premika* (angl. move), pri katerem druga spremenljivka prevzame lastništvo (in s tem odgovornost za čiščenje pomnilnika) in *izposoje* (angl. borrow), pri katerem se ustvari (angl. read-only) referenca na objekt, spremenljivka pa *ne* prevzame lastništva. Preverjanje pravilnosti sproščanja pomnilnika se izvaja *med prevajanjem* v posebnem koraku analize izposoj in premikov (angl. borrow checker). Prevajalnik zna v strojno kodo dodati ustrezne ukaze, ki ustrezno sproščajo pomnilnik in tako na predvidljiv, varen in učinkovit način zagotovi upravljanje s pomnilnikom.

Na podlagi principa lastništva in izposoje iz Rusta je nastal len funkcijski programski jezik Blang [23]. Interpreter jezika zna pomnilnk za zaprtja izrazov in spremenljivk med izvajanjem samodejno sproščati brez uporabe čistilcev, zatakne pa se pri sproščanju funkcij in delnih aplikacij.

Programski jezik micro-mitten [24] je programski jezik, podoben Rustu, ki za upravljanje s pomnilnikom uporablja princip ASAP (angl. As Static As Possible) [25]. Prevajalnik namesto principa lastništva izvede zaporedje analiz pretoka podatkov (angl. data-flow), namen katerih je aproksimirati statično živost spremenljivk na kopici. Pri tem prevajalnik ne postavi dodatnih omejitev za pisanje kode, kot jih poznamo npr. v Rustu, kjer mora programer za pisanje delujoče in učinkovite kode v vsakem trenutku vedeti, katera spremenljivka si objekt lasti in kakšna je njena življenjska doba. Metoda ASAP še ni dovolj raziskana in tako še ni primerna za produkcijske prevajalnike.

3.3 Predvideni prispevki magistrske naloge

V magistrskem delu se bomo primarno ukvarjali s STG jezikom. Operacijska semantika tega veleva, da so vsi izrazi v izvorni kodi v pomnilniku predstavljeni kot zaprtja. Jezik vsebuje izraz `let`, ki na kopici ustvari novo zaprtje, izbirni izraz `case` pa šele dejansko izračuna njegovo vrednost. Jezik STG za čiščenje zaprtij iz kopice uporablja generacijski čistilec pomnilnika [16, 26].

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

3.4 Metodologija

Za potrebe naše magistrske naloge bomo v izbranem programskem jeziku implementirali simulator STG stroja. V programskem jeziku STG bomo napisali zbirko programov, s pomočjo katerih bomo testirali uspešnost implementirane metode. Merili bomo količino dodeljenega pomnilnika in količino sproščenega pomnilnika in skušali ugotoviti, ali je ves pomnilnik pravočasno sproščen. Cilj magistrskega dela ni izdelava učinkovite implementacije čiščenja pomnilnika, temveč skušati ugotoviti, kakšne spremembe in analize je potrebno dodati v STG stroj, da bo lahko uporabljal princip lastništva namesto čistilca pomnilnika.

3.5 Literatura in viri

[1] R. Jones, A. Hosking, E. Moss, The garbage collection handbook: the art of automatic memory management, CRC Press, 2023.

[2] G. E. Collins, A method for overlapping and erasure of lists, Communications of the ACM 3 (12) (1960) 655–657.

[3] J. McCarthy, Recursive functions of symbolic expressions and their computation by machine, part I, Communications of the ACM 3 (4) (1960) 184–195.

[4] R. R. Fenichel, J. C. Yochelson, A LISP garbage-collector for virtual-memory computer systems, Communications of the ACM 12 (11) (1969) 611–612.

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

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

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

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

[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] S. L. P. Jones, Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine, Journal of functional programming 2 (2) (1992) 127–202.

[17] 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.

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

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

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

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

[22] 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.

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

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

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

[26] 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.