Sinteza programa

Seminarski rad u okviru kursa
Metodologija stručnog i naučnog rada
Matematički fakultet

Anja Ivanišević, Ivan Ristović, Milana Kovačević, Vesna Katanić
kontakt email prvog, drugog (trećeg) autora

??. ?? 2018.

Sažetak

U ovom tekstu je ukratko prikazana osnovna forma seminarskog rada. Obratite pažnju da je pored ove .pdf datoteke, u prilogu i odgovarajuća .tex datoteka, kao i .bib datoteka korišćena za generisanje literature. Na prvoj strani seminarskog rada su naslov, apstrakt i sadržaj, i to sve mora da stane na prvu stranu! Kako bi Vaš seminarski zadovoljio standarde i očekivanja, koristite uputstva i materijale sa predavanja na temu pisanja seminarskih radova. Ovo je samo šablon koji se odnosi na fizički izgled seminarskog rada (šablon koji *morate* da ispoštujete!) kao i par tehničkih pomoćnih uputstava. Molim Vas da kada budete predavali seminarski rad, imenujete datoteke tako da sadrže temu seminarskog rada, kao i imena i prezimena članova grupe (ili samo temu i prezimena, ukoliko je sa imenima predugačko). Predaja seminarskih radova biće isključivo preko web forme, a NE slanjem mejla.

Sadržaj

1 Uvod

Uz sve novouvedene termine u zagradi naglasiti od koje engleske reči termin potiče. Naredni primeri ilustruju način uvođenja enlegskih termina kao i citiranje.

Teorema 1.1 *Problem zaustavljanja (eng.* halting problem*) je neodlučiv [?].*

Teorema 1.2 *Za prevođenje programa napisanih u programskom jeziku C može se koristiti GCC kompajler [?].*

Teorema 1.3 *Da bi se ispitivala ispravost softvera, najpre je potrebno precizno definisati njegovo ponašanje [?].*

Reference koje se koriste u ovom tekstu zadate su u datoteci *literature.bib*. Prevođenje u pdf format u Linux okruženju može se uraditi na sledeći način:

```
pdflatex TemaImePrezime.tex
bibtex TemaImePrezime.aux
pdflatex TemaImePrezime.tex
pdflatex TemaImePrezime.tex
```

Prvo latexovanje je neophodno da bi se generisao *.aux* fajl. *bibtex* proizvodi odgovarajući *.bbl* fajl koji se koristi za generisanje literature. Potrebna su dva prolaza (dva puta pdflatex) da bi se reference ubacile u tekst (tj da ne bi ostali znakovi pitanja umesto referenci). Dodavanjem novih referenci potrebno je ponoviti ceo postupak.

Broj naslova i podnaslova je proizvoljan. Neophodni su samo Uvod i Zaključak. Na poglavlja unutar teksta referisati se po potrebi.

Teorema 1.4 *U odeljku* **??** *precizirani su osnovni pojmovi, dok su zaključci dati u odeljku 6.*

Još jednom da napomenem da nema razloga da pišete:

```
\v{s} i \v{c} i \'c ...
```

Možete koristiti srpska slova

```
š i č i ć ...
```

Ovde pišem uvodni tekst. Ovde pišem uvodni tekst. Ovde pišem uvodni tekst. Ovde pišem uvodni tekst.

2 Primene

Sinteza programa je moćan alat i mogućnosti primene su raznolike. Programski sintezeri se mogu naći u velikom broju današnjih aplikacija. Automatsko generisanje programskog koda čini proces programiranja manje repetitivnim i manje podložnim greškama. Moderni sintezeri omogućavaju programerima da zadaju šablon koda ili par *primera* koda, dok se ostatak generiše automatski. Ovaj proces se naziva *Programiranje vođeno primerima* (eng. *Programming Based on Examples)* [3], u daljem tekstu *PBE.*

Neke od oblasti primene sinteze programa koje će biti pokrivene u ovom radu su:

- Priprema podataka
- Grafika
- Popravka koda

- Sugestije prilikom kodiranja
- Superoptimizacija
- Konkurentno programiranje

2.1 Priprema podataka

Priprema podataka predstavlja proces čišćenja, transformacije i pripreme podataka iz polu-struktuiranog formata u format pogodan za analizu i prezentovanje. Problem je skupoća dovođenja podataka u oblik pogodan za primenu algoritama iz oblasti mašinskog učenja ili istraživanja podataka radi izvlačenja korisnih zaključaka. PBE čini čitav ovaj proces bržim [3].

Proces pripreme podataka često obuhvata sledeće korake:

- izvlačenje
- transformacija
- formatiranje

Transformacija podataka se često svodi na manipulisanje niskama ili izmenama samih tipova podataka. Iako savremeni programski jezici pomažu korisnicima uvodeći širok spektar komandi, korisnici se zamaraju pisanjem skriptova ili makroa kako bi obavili posao. Alati koji koriste PBE su idealni za ovakav posao [3, 6]

[slika ovde ?]

2.2 Grafika

Programski opis grafičkih objekata dovodi do bržih proračuna koordinata zavisnih tačaka od tačaka koje imaju slobodne koordinate, što omogućava interaktivne izmene i efikasne animacije. Tehnike sinteze programa mogu uspešno generisati rešenja geometrijskih problema srednješkolske težine [12].

Slike i crteži (u daljem tekstu *grafike*) nekada sadrže ponovljene šablone, teksture ili objekte. Konstrukcija takvih grafika zahteva pisanje skriptova ili ponovljenih operacija, što može biti jako neprijatno i podložno greškama. Korišćenjem PBE, moguće je mogućiti korisniku da prikaže par primera i ostavi posao sintezeru da predvidi naredne objekte u nizu [5]. Štaviše, korišćenjem grafičkog interfejsa za domen vektorske grafike, moguće je interaktivno omogućiti korisniku crtanje isključivo pomoću grafičkih alata a generisanje programskog koda ostaviti sintezeru.

[slika ovde ?]

2.3 Popravka koda

Za dat program P i specifikaciju ϕ, problem popravke koda zahteva računanje modifikacija programa P koje stvaraju nov program P' takav da zadovoljava ϕ. Osnovna ideja ovih tehnika je da se prvo ubace alternativni izbori za izraze u programu, a onda tehnikama programske sinteze izraza pronađu izrazi koji program dovode u oblik koji zadovoljava ϕ. Postoji mnogo tehnika sinteze napravljenih specifično za problem popravke koda [7, 1].

2.4 Sugestije prilikom kodiranja

Većina današnjih programerskih okruženja omogućava neku vrstu automatske dopune koda (*IntelliSense* za *MS Visual Studio* ili *Content Assist* za *Eclipse*). Programski sintesizeri mogu potencijalno generisati ne samo tokene već i čitave jedinice koda. Dva najbitnija pristupa ovom problemu su *statistički modeli* [13] i *type-directed completion* [2], ali je važno napomenuti da postoje i alati koji su uspešni iako ne koriste ove tehnike kao npr. *InSynth* **??** i *Bing Developer Assistant* **??**.

2.5 Superoptimizacija

Superoptimizacija predstavlja proces kreiranja optimalnog poretka instrukcija mašinskog koda tako da je dobijeni fragment koda ekvivalentan polaznom uz dobijanje na performansama [9].

Kao primer, uzmimo računanje proseka dva broja x i y. Formula $prosek = \frac{x+y}{2}$ može dovesti do prekoračenja. Takođe koristi skupu aritmetičku operaciju deljenja. Alternativa je formula $(x \mid y) - ((x \oplus y) \gg 1)$. Ova formula koristi veoma brze bitovske operatore i operaciju oduzimanja, rešavajući probleme prethodne formule.

Jedan od načina da se kod optimizuje automatski je korišćenje *enumerativne pretraga* (videti 3.2.1 na strani 6), sa *LENS* algoritmom kao predstavnikom [10].

2.6 Konkurentno programiranje

Sinteza programa se u oblasti konkurentnog programiranja koristi kao pomoć programerima u pisanju bezbednog kompleksnog koda. Postoji više tehnika sinteze koje uspešno automatski postavljaju minimalne konstrukte za sinhronizaciju u programski kod. Predstavnik u ovom polju je *Sinteza vođena apstrakcijom* (eng. *Abstraction-Guided Synthesis)* (u daljem tekstu *AGS*) [8].

AGS je tehnika kojom se program i njegova apstrakcija zajedno menjaju sve dok apstrakcija programa ne postane dovoljno precizna da verifikuje program. AGS algoritam pravi apstraktnu reprezentaciju programa u apstraktnom domenu, i proverava da li postoji prekršenje postavljene specifikacije programa. Specifikacija se obično vezuje za trku za podacima. Ukoliko postoji prekršenje specifikacije, AGS algoritam nedeterministički bira da li da menja apstrakciju (npr. sužavanje domena) ili da menja sam program dodavajući sinhronizacione konstrukte. Ovaj postupak se ponavlja sve dok se ne nađe program koji može biti verifikovan apstrakcijom.

3 Izazovi

Pisanje programa koji može da sintetiše drugi program predstavlja veliki izazov. Naime, ovaj problem se može razložiti na dva potproblema:

- Definisanje specifikacija željenog programa,
- Pretraživanje prostora mogućih programa u potrazi za onim koji zadovoljava definisane specifikacije.

Prostor programa se povećava eksponencijalno brzo u odnosu na veličinu željenog programa. Zbog toga postoje različiti pristupi njegovog pretraživanja, a neke od tih tehnika su opisane u poglavlju 3.2.

3.1 Definisanje specifikacija

Generisani program treba da se ponaša na način koji to korisnik definiše. Međutim, precizno definisanje zahteva je zapravo mnogo teže nego što izleda na prvi pogled. Postoje različiti načini na koje se to može program može opisati. Može se opisati formalnim logičkim izrazima, kao i neformalnim metodama ili primerima ulaza i izlaza programa.

Formalno definisanje zahteva može često da izgleda komplikovano (možda čak i da deluje komplikovanije neko pisanje samog programa). Nasuprot tome, neformalne metode su mnogo prirodnije korisniku, ali dovode do drugih problema. Na primer, neka se željeni program definiše na osnovu primera njegovog ulaza i izlaza na sledeći način:
"John Smith" $\rightarrow$ *"Smith, J."*.
Ovaj program vrši nama intuitivnu transformaciju niski, ali, na primer, da bi se on automatski generisao korišćenjem FlashFill [4] programa, tj. program treba da pretraži prostor koji sadrži milione mogućih rešenja. Problem je u tome što programi nemaju ljudsku intuiciju, već se preprilagođavaju datim primerima ulaza i izlaza.

Većina programa koji se danas koriste su previše komplikovani da bi se u potpunosti opisali bilo formalnim bilo neformalnim metodama. Čak i ako bi se to nekako uspelo, opis programa bi mogao da bude toliko obiman kao i sama implementacija programa. Kako bi senteza ovakvih, realnih programa bila moguća, potrebno je omogućiti korisniku da na početku definiše željeni program do neke tačke, a da kasnije tokom sinteze, interaktivno sa računarom, postepeno dolazi do rešenja.

Upravo ovakvu napradnu pretragu koristi gore spomenuti program FlashFill. Tokom pretrage, on uključuje dodatnu komunikaciju sa korisnikom. Ovako on usmerava pretragu, te na kraju ipak uspeva da nađe rešenje u realnom vremenu.

3.2 Prostor programa

Svaka uspešna sinteza programa vrši neki vid pretrage prostora mogućih programa (eng. *search space*). Ovo je težak kombinatorni problem. Broj mogućih rešenja raste eksponencijalno sa veličinom programa, te pretraga svih kandidata nije moguća u realnom vremenu. Potrebno je pažljivo vršiti odsecanja dela prostora pretrage kako bi se došlo do rešenja u realnom vremenu.

Tehnike pretrage se mogu zasnivati na enumerativnoj pretrazi, dedukciji, tehnikama sa ograničenjima, statističkim tehnikama, kao i na kombinaciji nekih od nih.

3.2.1 Enumerativna pretraga

Tehnike enumerativne pretrage za sintezu prigrana su se pokazale kao jedne od najefikasnijih tehnika za generisanje malih programa. Razlog ove efikasnosti je u pametnim tehnikama *čišćenja* (eng. *pruning*) u prostoru programa koji se pretražuje. Glavna ideja je da se prvo na neki način opiše prostor pretrage u kome se nalazi žejleni program. To može da se postigne korišćenjem meta-podataka kao što su veličina programa ili njegova složenost. Kada se mogući programi numerišu po osobinama, mogu da se odmah odbace oni koji ne zadovoljavaju prethodno definisane specifikacije.

Kako na osnovu pretpostavki vrši velika odsecanja, može da se dođe do toga da pogrešno numeriše neki od programa i time izgubi neka od

mogućih rešenja. Zato je enumerativna tehnika polu-odlučiva, ali u opštem slučaju je upotrebljiva i daje dobre rezultate i to relativno brzo.

3.2.2 Deduktivna pretraga

Deduktivna sinteza programa je tradicionalni pogled na sintezu programa. Ovakvi pristupi pretpostavljaju da postoji celokupna formalna specifikacija željenog programa. Ovo je vrlo jaka pretpostavka imajući u vidu da ta specifikacija može da bude veoma velika ukoliko je program kompleksan. Rešenje se sintetiše postupkom dokazivanja teorema, logičkim zaključivanjem i razrešavanjem ograničenja.

Deduktivna pretraga je pretraga odozgo - na dole. Koristi tehniku podeli-pa-vladaj (eng. *divide-and-conquer*). Program sintetiše tako što se prvo podeli na potprobleme tako da svaki od njih ima svoju specifikaciju. Rekurzivno se obrade potproblemi, a zatim iskombinuju podrešenja kako bi se dobilo glavno rešenje.

Deljenje problema na potprobleme koji mogu da se sintetišu odvojeno nije moguće u opštem slučaju. Ovo zavisi od prirode problema. U tom slučaju se deduktivna pretraga može iskombinovati sa enumerativnom. Kada deduktivna pretraga više ne može da razloži problem, enumerativnom pretragom (koja je odozdo - na gore) tada treba pretražiti prostor rešenja potproblema, a nakon toga spojiti dobijene rezultate.

Deduktivna pretraga može lako da zaključi vrednosti konstanti u programu. To je bitno jer ukoliko program sadrži veliki broj konstanti, sama enumerativna pretraga bi se izgubila pokušavajući da pogodi njihove prave vrednosti.

3.2.3 Induktivna pretraga

Prethodno smo videli da se u slučaju deduktivne pretrage program sintetiše dokazivanjem teorema i razrešavanjem ograničenja. Za razliku od nje, induktivna pretraga sintetiše program na osnovu skupa ulaza i odgovarajućih izlaza. Sintezer koji koristi induktivnu pretragu pokušava da nađe opšti program koji zadovoljava dati skup ulaza i izlaza. Ovaj vid pretrage predstavlja jedan vid učenja na osnovu primera, takođe poznato kao *Mašinsko učenje* (eng. *Machine learning*). Dobra strana ovog pristupa jeste što je dovoljno fleksibilan da može da radi i sa nepotpunim specifikacijama problema. Međutim, može se desiti da rezultat ne zadovolji očekivanja korisnika ukoliko neki bitni slučajevi koji nisu pokriveni specifikacijom.

Postoji više različitih pristupa koji mogu da se koriste u induktivnoj pretrazi. Jedan od tih pristupa je sinteza sa aktivnim učenjem. *Aktivno učenje (eng.* Active learning*)* je poseban vid mašinskog učenja gde je algoritmu za učenje dopušteno da sam vrši selekciju podataka na osnovu kojih će da uči. Ovaj sintezer koristi induktivni algoritam za učenje, koji u sebi često poziva i deduktivne procedure. Iako kombinuje oba pristupa, sinteza sa aktivnim učenjem se u literaturi predstavlja kao primer induktivne sinteze.

Još jedan primer induktivne pretrage je CEGIS. Ova tehnika će biti detaljno opisana u poglavlju 4.

3.2.4 Tehnike sa ograničenjima

Mnoge uspešne tehnike sinteze programa u svojoj osnovi sadrže tehnike prilagođavanja datim ograničenjima (eng. *constraint solving*). One se

sastoje od dva velika koraka:

- *Generisanje ograničenja* - U opštem slučaju, kada se kaže prilagođavanje ograničenjima misli se na na pronalaženje modela za formulu koja opisuje željeni program. Osnovna ideja je da se specifikacija programa kao i njegova dodatna ograničenja zapišu u jednoj logičkoj formuli. Uglavnom se tom prilikom u formulu dodaju i pretpostavke o rešenju.
- *Razrešavanje ograničenja* - Formula u kojoj su zapisana ograničenja često sadrži kvatifikatore i nepoznate drugog reda. Ona se prvo transformiše u oblik pogodan za nekog od rešavača, na primer za SAT ili SMT rešavač. Na ovaj način se problem pretrage svodi na problem ispitivanja zadovoljivosti prosleđene formule. Svaki nađeni model za tu formulu predstavlja jedno moguće rešenje koje zadovoljava data ograničenja.

3.2.5 Statistička pretraga

Postoji veliki broj metoda koje se koriste za pretragu prostora programa a koriste neku vrstu statistike kako bi došle do rešenja. Mogu se koristiti tehnike mašinskog učenja, genetsko programiranje, probabilističko zaključivanje i mnoge druge.

Mašinsko učenje - Tehnike mašinskog učenja mogu doprineti ostalim pretragama uvodeći verovatnoću u čvorove granjanja prilikom pretrage. Vrednosti verovatnoće se uglavnom generišu pre sinteze programa: tokom treninga ili na primer na osnovu datih primera ulaza i izlaza.

Genetsko programiranje - Genetsko programiranje je metod inspirisan biološtkom evolucijom. Sastoji se od održavanja populacije programa, njihovog ukrštanja i mogućih mutacija. Svaka jedinka populacije se ispituje u kojoj meri zadovoljava specifikacije žejlenog programa. One jedinke koje bolje odgovaraju rešenju nastavljaju da evoluiraju. Uspeh genetskih algoritama zavisi od funkcije zadovoljivosti, čiji dobar izbor predstavlja najteži problem ove tehnike.

Probabilističko zaključivanje (eng. *Probabilistic inference*) - Ova tehnika dolazi do rešenja nadograđivanjem početnog programa. Dodaju se sitne izmene, jedna po jedna, i proverava da li takav promenjen program zadovoljava specifikacije.

4 CEGIS

4.1 Motivacija

Program se može sintetisati tako što se definišu njegove specifikacije i zapišu u vidu formule koja se prosledi SMT rešavaču (kao što je na primer Z3). SMT nađe valuaciju koja je zadovoljiva i to predstavlja rešenje. Problem nastaje u tome što formula koja se prosleđuje rešavaču sadrži univerzalne kvantifikatore koje usporavaju pretragu. Naime, rešavač bi u svakom slučaju pronašao rešenje za datu formulu, ali kako bi se to desilo u realnom vremenu, CEGIS (*Counter-Example Guided Inductive Synthesis*) u sebi sadrži posebne taktike za optimizaciju.

Za većinu realnih problema nije neophodno da se razmatraju svi ulazi i izlazi kako bi se došlo do programa koji radi tačno za svaki od njih. Ovako razmišljajući, problem se menja i postaje: *koji je najmanji podskup ulaza koji je potrebno razmatrati da bi se sintetisao program koji*

zadovoljava date specifikacije? CEGIS tehnika upravo traga za tim minimalnim skupom. U petlji, korišćenjem SMT rešavača, on postepeno dolazi do svih mogućih implementacija željenog programa koristeći sve ulaze koji su razmatrani do tog trenutka (počinje sa 0 ulaza). U sledećoj iteraciji on razmatra dalje. Paralelno sa tim, drugim SMT rešavačem pronalazi kontra-primer koji pokazuje da poslednji sintetisani program nije rešenje. Ukoliko kontra-primer ne postoji, poslednji sintetisani prigram je rešenje. Ukoliko se prođe kroz sve iteracije i ne pronađe se rešenje, specifikacija programa nije smislena.

Jedna od mogućih implementacija se može naći na [11].

4.2 Arhitektura

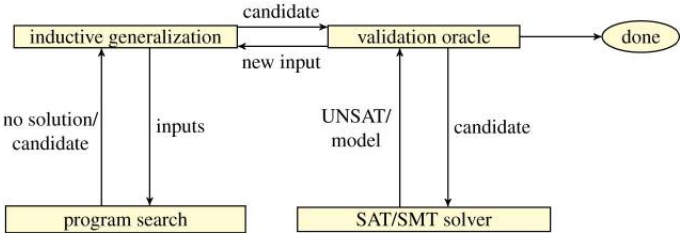
CEGIS se sastoji iz dve faze, induktivne sinteze i verifikacije. Na početku sintezeru dajemo specifikaciju željenog programa. U fazi sinteze pronalazi se program kandidat koji može da zadovolji specifikacije. Nakon toga se u fazi verifikacije proverava da li taj kandidat zaista zadovoljava specifikacije. Ako verifikator ne uspe da pronađe kontra primer znači da smo pronašli traženi program. U suprotnom, verifikator prosleđuje sintezeru informacije o kontra primeru, koje će mu pomoći prilikom daljeg traženja novog kandidata.

Ovakav vid pretrage zove se pretraga vođena kontra primerima (eng. *counterexample-guided*), zato što je povratna informacija sintezeru kontra primer koji se dodaje u specifikaciju programa.

Da bismo u potpunosti definisali CEGIS sintezu programa, potrebno je da odgovorimo na nekoliko važnih pitanja:

- Kako treba da izgleda specifikacija traženog programa?
- Kako ćemo vršiti sintezu programa kandidata?
- Kako da proverimo da li program kandidat zadovoljava specifikacije?
- Kako da prosledimo povratne informacije za buduće kandidate?

staviti ref na ovo zbog slike: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5597726/



Slika 1: CEGIS petlja

4.3 Primene CEGISa

U nastavku ćemo opisati kako se tri različite vrste sinteze programa koriste zajedno sa CEGIS idejom.

4.3.1 Sinteza vođena oracle-om

Sinteza vođena oracle-om (eng. *Oracle-guided synthesis* u daljem tekstu OGS) pretpostavlja da imamo implementaciju programa koji želimo da sintetišemo, koja se naziva oracle program. Mi ćemo oracle tretirati kao crnu kutiju, možemo da joj damo ulazne vrednosti i dobijemo odgovarajući izlaz, ali nećemo razmatrati njenu implementaciju. Počinjemo sa skupom test primera, koji su parovi ulaz-izlaz. Pošto imamo oracle, možemo da kreiramo novi test primer jednostavno generišući random ulaze i od oracla ćemo dobiti odgovarajući izlaz za svaki prosleđeni ulaz. Oracle program će predstavljati specifikacija za OGS.

Faza sinteze

OGS-u prosleđujemo biblioteku komponenti na osnovu koje sanitaizer kreira program koji je kandidat za rešenje. Faza sinteze će uzeti sve komponente i odlučiti kako da poveže njihove ulaze i izlaze tako da formiraju program. Ovako dobijem program će biti u SSA formi (eng. *Static Single Assignment form* SSA).

Na primer neka imamo biblioteku od tri komponente - dva sabiranja i jedno korenovanje. Njih je moguće povezati na sledeći način:

```
program(x,y):
        o1 = add(x, y)
        o2 = add(o1, y)
        o3 = sqrt(o1)
        return o3
```

Primetimo da SSA foma jednostavno povezuje komponente međusobno, tako da ako želimo da imamo dva sabiranja moramo da obezbedimo dve komponente za sabiranje. Takodje primetimo da se vrednost o2 ne koristi, takozvani mrtvi kod. U ovom slučaju to je poželjna karakteristika, jer ne moramo unapred tačno da odredimo koliko ćemo imati kojih operacija, već možemo samo da zadamo gornju granicu, a sanitaizer će generistai mrtvi kod za komponente koje se ne koriste.

Sanitaizer koristi test primere da ograniči broj načina na koje se komponente mogu povezati. On koristi SMT rešavač da odredi koje komponente da poveže kako bi program bio tačan za sve test primere. Ako SMT rešavač ne uspe da nađe rešenje, onda nijedan program koji koristi samo date komponente ne može da zadovolji sve test primere - komponente nisu dovoljne. U suprotnom vraća program u SSA formi koji zadovoljava sve test primere.

Faza verifikacije

Ako postoji rešenje mi program koji je kandidat za rešenje prosleđujemo fazi verifikacije. Ovaj korak koristi SMT rešavač da odgovori na sledeće pitanje: Da li postoji program P', različit od kandidata za rešenje P, koji takođe zadovoljava sve test primere, ali se na nekom ulazu z ralikuje od P? Pojasnimo ovo deo po deo. Iz faze sinteze dobili smo program P koji je kandidat za rešenje i zadovoljava sve test primere. Ono što tražimo od verifikatora ja novi ulaz z i novi program P', takvi da za ulaz z programi P i P' daju različite izlaze. Program P' takođe zadovoljava sve početne test primere. Drugim rečima pitamo se da li postoji više od jednog programa koji mogu da zadovolje sve test primere? Ako postoji nismo završili. U

ovom trenutku ništa ne pitamo oracle program. Može da se desi i da program P i P' daju netačan izlaz za ulaz z, jedino što je u ovom trenutku važno za verifikator je da oni daju različit rezultat.

Na primer evo kako radi za dva test primera i komponente koje smo koristili u prethodnom primeru.

Oracle:	O(x, y) = sqrt(x+y)
Candidate:	P(x, y) = sqrt(x) + y
New program:	P'(x, y) = x+y

	Inputs	**O(x, y)**	**P(x, y)**	**P'(x, y)**
Existing test cases	0, 0	0	0	0
	1, 0	1	1	1
New test case	4, 5	3	7	9

Slika 2: Primer rada faze verifikacije

U ovom slučaju imamo samo dva test primera (0,0) i (1,0). Sanitaizer nam daje program P=sqrt(x)+y koji je kandidat i koji zadovoljava oba test primera. Od verifikatora tražimo da nam da dve stvari: novi program P' i novi ulaz z. U ovom slučaju on to i uspeva. Vraća novi program x+y i novi test primer (4,5). Na novom test primeru program P i P' daju različite izlaze: sqrt(4)+5=7 dok je 4+5=9. U ovom slučaju ispostavlja se da oba programa daju pogrešan rezultat sobzirom da je izlaz koji daje oracle sqrt(4+5)=3. Medjutim činjenica da se P i P' razlikuju nam je dovoljna da se u petlji vratimo nazad pomoću povratnog koraka.

Povratni korak

Ovaj korak razmatra novodobijeni ulaz z. Ulaz z se daje oraclu i od oracla se dobija ispravan izlaz. Zatim se ulaz z i njegov ispravan izlaz dodaju u skup test primera i ponovo prolazi kroz petlju.

Da bi smo bili u potpunosti sigurni u naše rešenje moramo da imamo i neku vrstu validacije. Problem je da tokom prolaska kroz CEGIS petlju možemo da nađemo jedinstveni program koji zadovoljava sve test primere pre nego što nadjemo test primer koji taj program ne zadovoljava. Faza validacije treba da nakon završetka petlje potvrdi da program zadovoljava sve ulaze, a ne samo test primere.

4.3.2 Stohastička superoptimizacija

Stohastička superoptimizacija pretražuje prostor programa i traži novi koji se ponaša isto kao i originalni, ali je brži ili efikasniji. Ovde ćemo takodje prtpostaviti da imamo implementaciju programa kao specifikaciju. Ovo nam neće predstavljati problem, jer tražimo optimalan skup instrukcija za dati kod.

Faza sinteze

Faza sinteze na osnovu tekućeg programa P sintetiše novi program P' primenom MCMC (eng. *Markov-chain Monte Carlo sampling.* Definiše se i funkcija prilagođenosti (eng. *cost function*) koja odredjuje koliko je program P' blizu traženom programu i koliko je brz program P' kako bi odredili da li da prihvatimo program P' ili ne. Što je kandidat bliži željenom programu veće su šanse da bude prihvaćen, ali čak i progami koji su dalji ili sporiji od traženog mogu biti prihvaćeni sa nekom verovatnoćom. Ako je kandidat prihvaćen prelazimo na fazu verifikacije, a u suprotnom ponavljamo proces.

Faza verifikacije

U ovoj fazi se kandidat i ciljni program prosleđuju verifikatoru kako bi se utvrdilo da li su ekvivalentni. Pošto verifikator može da bude veoma spor pre prosleđivanja programa verifikatoru koriste se test primeri iz funkcije prilagpđenosti. Ako bilo koji od test primera ne uspe, sa sigurnošću možemo reći da kandidat ne može da bude ispravno rešenje pa se ni ne poziva verifikator.

Povratni korak

U ovoj fazi poredimo prethodno prihvaćeni program P i novog kandidata P'. Ako je P' bolji od P onda ćemo nadalje njega razmatrati. Ako P' nije bolji od P verovatnoća da ćemo razmatrati P' zavisi od toga koliko su programi P i P' slični.

4.3.3 Enumerativna pretraga

Ovde ćemo za specifikaciju koristiti konačan skup test primera. Takodje pretpostavićemo da imamo gramatiku koja opisuje naš ciljani jezik. Na primer možemo uzeti gramatiku sa dve operacije add i sub i dve promenljive x i y. add(x, sub(x,y)) je jedan primer programa u ovoj gramatici.

Faza sinteze

Ideja enumerativne pretrage je da pretraži sve moguće programe. Programe delimo prema dubini, npr. program koji vraca x ima dubinu 0, a onaj koji vraća (x+y) dubinu 1. Sintezu počinjemo od dubine 0 i pritom numerišemo sve programe na ovoj dubini. Na dubini k, ispitujemo sve programe koji imaju oblik operacija(a,b), gde su a i b bilo koji izraz dubine k-1. Broj programa koje treba ispitati ovakvim načinom pretrage raste eksponencijalno, pa se oslanjamo na povratni korak kako bi ubrzali pretragu.

Faza verifikacije

Pošto nam se specifikacija programa zasniva na test primerima u ovoj fazi ćemo jednostavno pokrenuti sve test primere i uporediti rezultate. Ako program daje odgovarajuće izlaze za sve test primere završili smo.

Povratni korak

Jedan od načina da smanjimo prostor pretrage je da na dubini k ne razmatramo sve programe dužine k-1 već samo različite programe. Posta-

vlja se pitanje kako da odredimo da li su dva programa različita? Pošto smo ciljni program definisali pomoću test primera nije bitno da li su nam dva programa semantički jednaka, već nam je bitno da za iste test primere daju različite rezultate. Na ovaj način smo iz pretrage izbacili mnogo programa koji nisu semantički jednaki, ali se isto ponašaju u našem slučaju.

5 Slike i tabele

Slike i tabele treba da budu u svom okruženju, sa odgovarajućim naslovima, obeležene labelom da koje omogućava referenciranje.

Teorema 5.1 *Ovako se ubacuje slika. Obratiti pažnju da je dodato i* `\usepackage{graphicx}`

Slika 3: Pande

Na svaku sliku neophodno je referisati se negde u tekstu. Na primer, na slici 3 prikazane su pande.

Teorema 5.2 *I tabele treba da budu u svom okruženju, i na njih je neophodno referisati se u tekstu. Na primer, u tabeli 1 su prikazana različita poravnanja u tabelama.*

Tabela 1: Razlčita poravnanja u okviru iste tabele ne treba koristiti jer su nepregledna.

centralno poravnanje	levo poravnanje	desno poravnanje
a	b	c
d	e	f

6 Zaključak

Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak.

Literatura

[1] Andreas Griesmayer i Roderick Bloem Barbara Jobstmann. Program repair as a game. 2005.

[2] Thomas Ball i Dan Grossman Daniel Perelman, Sumit Gulwani. Type-directed completion of partial expressions. 2014.

[3] Sumit Gulvani. Programming Based on Examples, 2016. on-line at: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/12/pbe16.pdf.

[4] Sumit Gulwani. *Automating string processing in spreadsheets using input-output examples*. Symposium on Principles of Programming Languages, POPL, Austin, TX, USA, 2011.

[5] Yves Chauvin i David E. Rumelhart. *Backpropagation: theory, architectures, and applications*. Psychology Press, 1995.

[6] Rishabh Singh i Sumit Gulwani. Synthesizing number transformations from input-output examples. pages 634–651, 2012.

[7] Roopsha Samanta i Rishabh Singh Loris D'Antoni. Qlose: Program Repair with Quantitative Objectives. 1995.

[8] Eran Yahav i Greta Yorsh Martin T. Vechev. Abstraction-guided synthesis of synchronization. 2010.

[9] Henry Massalin. Superoptimizer - A look at the smallest program. 1987.

[10] Rastislav Bodik i Dinakar Dhurjati Phitchaya Mangpo Phothilimthana, Aditya Thakur. Scaling up superoptimization. 2016.

[11] Microsoft Research. Counter-example guided inductive synthesis (CEGIS) implementation for the SMT solver Z3, 2017. on-line at: https://github.com/marcelwa/CEGIS.

[12] Vijay Anand Korthikanti i Ashish Tiwari Sumit Gulwani. Synthesizing geometry constructions. pages 50–61, 2011.

[13] Martin Vechev i Eran Yahav Veselin Raychev. Code completion with statistical language models. 2014.

A Dodatak

Ovde pišem dodatne stvari, ukoliko za time ima potrebe. Ovde pišem dodatne stvari, ukoliko za time ima potrebe. Ovde pišem dodatne stvari, ukoliko za time ima potrebe. Ovde pišem dodatne stvari, ukoliko za time ima potrebe. Ovde pišem dodatne stvari, ukoliko za time ima potrebe.