Sinteza programa

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

Anja Ivanišević, Ivan Ristović, Milana Kovačević, Vesna Katanić

april 2018.

Sažetak

Sinteza programa je oblast koja se bavi automatskim generisanjem programa. Korisnik na neki način opiše željeni program, a zadatak sintezera je da ga generiše tako da on zadovoljava zadata ograničenja. Sintezer može svom zadatku da pristupi na različite načine, a neki od tih načina su detaljnije opisani u radu. Primene sinteze programa su brojne, od priprema podataka, prepravljanja i dobijanja sugestija prilikom kodiranja do predloga mogućih optimizacija. Kao jedan od značajnijih pristupa sintezi, detaljno je opisana tehnika CEGIS. To je pristup sintezi koji u svojoj osnovi sadrži iterativno generisanje kandidata za traženi program i proveru da li taj kandidat zadovoljava date uslove. S obzirom na obimnu terminologiju koja prati oblast sinteze programa, nećemo se previše baviti sitnim detaljima već ćemo čitaocu kroz reference pružiti osnove za dalje istraživanje.

Sadržaj

1 Uvod

U današnje vreme, pristup modernim tehnologijama je svima omogućen. Potražnja za softverom je sve veća, a samo mali procenat ljudi ima adekvatna znanja za njegovo programiranje. Standardni način programiranja se sastoji od dizajniranja algoritma koji rešava problem i njegove implementacije. Automatska sinteza programa ima potencijal da promeni generalni pristup implementaciji programa. Ovakav način kreiranja softvera bi mogao da omogući i manje stručnim licima da programiraju bez dubokog poznavanja algoritama, struktura podataka i optimizacija. Ideja je da korisnici daju opis željenih funkcionalnosti programu za automatsko generisanje koda (u daljem tekstu *sintezeru*), a on će im automatski generisati neophodnu implementaciju.

Poslednjih nekoliko decenija došlo je do značajnog pomaka u razvoju automatskog rezonovanja, naročito u unapređivanju SAT (eng. *Propositional Satisfiability Problem*) i SMT (eng. *Satisfiability Modulo Theories*) rešavača [**?**], koji su sad u mogućnosti da reše i neke industrijske probleme [**?**]. Ovaj napredak u automatskom rezonovanju dao je vetar u leđa razvoju programske sinteze koja svoja rešenja zasniva na logici prvog i drugog reda i automatskom dokazivanju teorema.

U ovom radu osvrnućemo se na neke primene programske sinteze, najveće izazove koji se u njoj javljaju, i neke tehnike zasnovane na CEGIS-u (eng. *Counterexample-Guided Inductive Synthesis*). Navedene oblasti su podstakle razvoj novih alata i doprinele razvoju sinteze programa kao posebnog pravca savremenog računarstva.

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 nekoliko primera izlaza koda, dok se ceo kod generiše automatski. Ovaj proces se naziva *Programiranje vođeno primerima* (eng. *Programming Based on Examples)* [**?**] (u daljem tekstu *PBE*).

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

- Priprema podataka
- Popravka koda
- Sugestije prilikom kodiranja
- Grafika
- Superoptimizacija
- Konkurentno programiranje

2.1 Priprema podataka

Priprema podataka predstavlja proces čišćenja i transformacije podataka iz polustruktuiranog formata u format pogodan za analizu i prezentovanje. Problem je visoka cena 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 [**?**].

Proces pripreme često obuhvata sledeće operacije nad podacima:

- izvlačenje
- transformacija
- formatiranje

Transformacija podataka se često svodi na manipulisanje niskama ili izmene 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 [**?**, **?**].

2.2 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 [**?**, **?**]. Primer koda generisanog od strane programa *SemFix* [**?**] se može videti na slici 1.

Ulaz			Izlaz	
inb	usep	dsep	expected	actual
1	0	100	0	0
1	11	110	1	0
0	100	50	1	1
1	-20	60	1	0
0	0	10	0	0

```
int buggy(int inb, int usep, int dsep)
{
    int bias;
    if (inb)
        bias = dsep; //fix: bias = usep+100
    else
        bias = usep;
    if (bias > dsep)
        return 1;
    else
        return 0;
}
```

Slika 1: Primer koda sinteziranog od strane programa *SemFix* [**?**] koristeći skup ulaznih i izlaznih test primera.

2.3 Sugestije prilikom kodiranja

Većina današnjih okruženja za rad omogućava neku vrstu automatske dopune koda (npr. *IntelliSense* za *MS Visual Studio* ili *Content Assist* za *Eclipse*). Programski sintezeri mogu potencijalno generisati ne samo tokene, već i čitave jedinice koda. Dva najbitnija pristupa ovom problemu su *statistički modeli* [**?**] i *dopuna usmerena tipovima* (eng. *type-directed*

completion) [**?**], ali je važno napomenuti da postoje i alati koji su uspešni iako ne koriste ove tehnike (npr. *InSynth* [**?**] i *Bing Developer Assistant* [**?**]).

2.4 Grafika

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 veoma naporno i podložno greškama. Korišćenjem PBE, omogućava se korisniku da prikaže par primera i ostavi posao sintezeru da predvidi naredne objekte u nizu [**?**]. Štaviše, korišćenjem grafičkog interfejsa za domen vektorske grafike, moguće je interaktivno postavljati grafike isključivo pomoću grafičkih alata ili grafičkog interfejsa, a generisanje programskog koda ostaviti sintezeru.

Programski opis grafika 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 srednjoškolske težine [**?**].

2.5 Superoptimizacija

Superoptimizacija predstavlja proces kreiranja optimalnog poretka instrukcija mašinskog koda tako da je dobijeni fragment koda ekvivalentan polaznom uz dobijanje na performansama [**?**]. 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 automatski optimizuje je korišćenje *enumerativne pretrage* (videti sekciju 3.2.1 na strani 6), sa *LENS* algoritmom kao predstavnikom [**?**].

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 u programski kod uspešno postavljaju minimalne konstrukte za sinhronizaciju. Predstavnik u ovom polju je *Sinteza vođena apstrakcijom* (eng. *Abstraction-Guided Synthesis)* (u daljem tekstu *AGS*) [**?**].

AGS je tehnika kojom se program i njegova apstrakcija zajedno menjaju sve dok apstrakcija programa ne postane dovoljno precizna da ga verifikuje. AGS algoritam pravi apstraktnu reprezentaciju programa u apstraktnom domenu i proverava da li postoji kršenje 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. sužavanjem domena) ili da menja sam program dodajući 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 je oblast koja je tek na početku svog razvoja. Danas, napisati čak i jednostavan sintezer nije lako. Naime, posmatrano sa visokog nivoa, problem sinteze se može razložiti na dva potproblema čije je efikasno rešavanje njen najveći izazov. Ti potproblemi su:

- 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 na koji to korisnik definiše. Međutim, precizno definisanje zahteva je zapravo mnogo teže nego što izleda na prvi pogled. Postoje različiti načini za definisanje zahteva, npr. korišćenjem formalnih logičkih izraza, zadavanjem primera ulaza i izlaza programa ili definisanjem neformalnim metodama.

Formalno definisanje zahteva je ponekad i komplikovanije od pisanja 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* [**?**] programa, potrebno je pretražiti prostor programa 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 kad bi se to uspelo, opis programa bi mogao da bude toliko obiman koliko i sama njegova implementacija. Kako bi sinteza ovakvih, realnih programa bila moguća, potrebno je omogućiti korisniku da on na početku definiše željeni program do neke tačke, a da kasnije tokom sinteze, interaktivno sa računarom, postepeno dolazi do rešenja.

Upravo ovakvu naprednu pretragu koristi gorepomenuti 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 Pretraživanje prostora programa

Svaka uspešna sinteza programa vrši neki vid pretrage prostora mogućih programa (eng. *search space*). Prostor mogućih programa predstavlja skup koji sadrži sve moguće programe koji se mogu napisati, a pretraga ovog skupa znači nalaženje programa koji zadovoljava specifikacije. 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, induktivnim i statističkim metodama. U praksi se uglavnom koristi neka od njihovih kombinacija, te ih je teško razgraničiti kao posebne metode pretrage.

3.2.1 Enumerativna pretraga

Tehnike enumerativne pretrage za sintezu programa 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 željeni program. To može da se postigne korišćenjem metapodataka 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 se na osnovu pretpostavki vrše velika odsecanja, moguće je doći do gubitka nekog od mogućih rešenja usled pogrešnog numerisanja na početku. Zbog ovoga je enumerativna tehnika poluodlučiva, tj. ne garantuje pronalazak rešenja ukoliko ono postoji. Međutim, 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 nadole. Koristi tehniku podeli-pa-vladaj (eng. *divide-and-conquer*). Program se 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 ovom 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 provela mnogo vremena pokušavajući da pogodi njihove prave vrednosti.

3.2.3 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 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 kvantifikatore i nepoznate drugog reda. Ona se prvo transformiše u oblik pogodan za neki 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.

3.2.4 Induktivna pretraga

Induktivna pretraga je veoma opšti pojam. Ona se, na primer, može smatrati kao nadogradnja tehnike pretrage sa ograničenjima (videti 3.2.3) [**?**]. Prilikom svake iteracije se generišu ograničenja, rešavačem se dođe do mogućeg rešenja a zatim se ispita da li je ono zadovoljavajuće kao opšte rešenje.

Sa druge strane, induktivna pretraga može da koristi tehnike mašinskog učenja. To može da radi na razne načine, a jedan od takvih 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. Ovakav 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.

Dobra strana induktivnog 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 nisu pokriveni specifikacijom datom primerima ulaza i izlaza.

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

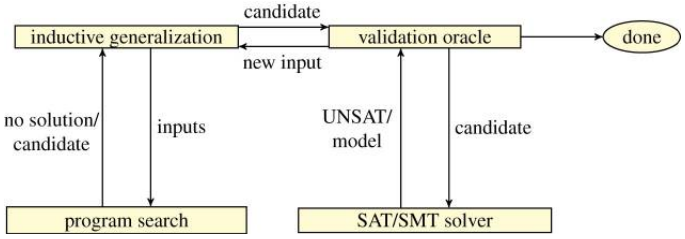
3.2.5 Statistička pretraga

Postoji veliki broj metoda koje se koriste za pretragu prostora programa, a koriste neku vrstu statistike kako bi došle do rešenja. Neke od tehnika koje se mogu koristiti su tehnike mašinskog učenja, genetsko programiranje i probabilističko zaključivanje.

Mašinsko učenje - Tehnike mašinskog učenja mogu doprineti ostalim pretragama uvodeći verovatnoću u čvorove grananja 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 biološkom evolucijom. Sastoji se od održavanja populacije programa, njihovog ukrštanja i mogućih mutacija. Ispituje se u kojoj meri svaka jedinka zadovoljva. 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 se da li takav promenjen program zadovoljava specifikacije.



Slika 2: CEGIS petlja [**?**]

4 CEGIS

Program se može sintetisati tako što se definišu njegove specifikacije i zapišu u vidu formule koja se prosledi SMT rešavaču (npr. Z3 [**?**]). SMT rešavač nađe valuaciju koja zadovoljava formulu i to predstavlja rešenje. Problem nastaje u tome što formula koja se prosleđuje rešavaču sadrži univerzalne kvantifikatore koji usporavaju pretragu. Naime, rešavač bi u svakom slučaju pronašao rešenje za datu formulu, ali da bi se to desilo u realnom vremenu, CEGIS u sebi sadrži posebne tehnike za optimizaciju. Jedna od mogućih implementacija se može naći na [**?**].

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 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 kontraprimer koji pokazuje da poslednji sintetisani program nije rešenje. Ukoliko kontraprimer ne postoji, poslednji sintetisani program je rešenje. Ukoliko se prođe kroz sve iteracije i ne pronađe se rešenje, specifikacija programa nije smislena.

4.1 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 kontraprimer znači da smo pronašli traženi program. U suprotnom, verifikator prosleđuje sintezeru informacije o kontraprimeru, koje će mu pomoći prilikom daljeg traženja novog kandidata. Na slici 2 predstavljena je opšta arhitektura CEGIS-a.

Ovakav vid pretrage zove se pretraga vođena kontraprimerima (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?

4.2 Primene CEGIS-a

U nastavku ćemo opisati kako se neke različite vrste sinteze programa koriste zajedno sa CEGIS idejom. Fokusiraćemo se na *sintezu vodjenu uzorom*, *stohastičku superoptimizaciju* i *enumerativnu pretragu*.

4.2.1 Sinteza vođena uzorom

Sinteza vođena uzorom (eng. *Oracle-guided synthesis*, u daljem tekstu *OGS*) [**?**] pretpostavlja da imamo implementaciju programa koji želimo da sintetišemo, koja se naziva program uzor. Mi ćemo uzor 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 program uzor, možemo da kreiramo novi test primer jednostavno generišući proizvoljne ulaze i od uzora ćemo dobiti odgovarajući izlaz za svaki prosleđeni ulaz. Uzor će predstavljati specifikaciju za OGS.

Faza sinteze

OGS-u prosleđujemo biblioteku komponenti na osnovu koje sintezer 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 dobijen program će biti u SSA formi (eng. *Static Single Assignment form*, u daljem tekstu *SSA*) [**?**].

Na primer, razmotrimo 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 forma jednostavno povezuje komponente međusobno, tako da ako želimo da imamo dva sabiranja moramo da obezbedimo dve komponente za sabiranje. Takođe primetimo da se vrednost $o2$ ne koristi, takozvani *mrtvi kod*. Prednost ove forme je što ne moramo unapred tačno da odredimo koliko ćemo imati kojih operacija. Dovoljno je samo da zadamo gornju granicu. Ukoliko se desi da neku komponentu ne iskoristimo, sintezer će generisati mrtvi kod za nju.

Sintezer 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, program koji je kandidat za rešenje se prosleđuje 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 razlikuje 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 su 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, sinteza nije završena. U ovom trenutku ništa ne pitamo uzor. Može da se desi i da program P i P' daju netačan izlaz za ulaz z, ali 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 3: Primer rada faze verifikacije

U ovom slučaju imamo samo dva test primera $(0, 0)$ i $(1, 0)$. Sintezer 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 programi 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 s obzirom na to da je izlaz koji daje program uzor $\sqrt{4+5} = 3$. Međutim 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 uzoru i od uzora se dobija ispravan izlaz z'. Zatim se ulaz z i z' dodaju u skup test primera i ponovo se prolazi kroz petlju.

Da bismo bili u potpunosti sigurni u naše rešenje moramo da imamo i neku vrstu validacije. Problem je to što tokom prolaska kroz CEGIS petlju možemo da nađemo jedinstveni program koji zadovoljava sve test primere pre nego što nađemo 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.

4.2.2 Stohastička superoptimizacija

Stohastička superoptimizacija pretražuje prostor programa i traži novi koji se ponaša isto kao i originalni, ali je brži ili efikasniji. Ovde ćemo takođe pretpostaviti 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 određuje 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 prilagođenosti. 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 u buduće njega razmatrati. Ako P' nije bolji od P verovatnoća da ćemo razmatrati P' zavisi od toga koliko su programi P i P' slični.

4.2.3 Enumerativna pretraga

Ovde ćemo za specifikaciju koristiti konačan skup test primera. Takođe 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 vraća 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. Postavlja 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 Zaključak

Razvojem tehnika automatske sinteze programa postavlja se pitanje da li će programeri moći da prestanu da govore računarima **kako** da rade, već da se fokusiraju na to da im kažu **šta** treba da urade. Ova oblast još uvek nije dovoljno razvijena da bi se koristila za razvijanje realnih, velikih aplikacija. Potrebno je još rada da bi se došlo do toga. Najveći potencijal ima induktivna sinteza programa. Iako teorijski deluje da ovaj pristup nije dovoljno efikasan, te da će se izvršavati predugo, CEGIS je, kao vodeći predstavnik ove grupe, u praksi pokazao neočekivano dobre rezultate. Uprkos tome, i uspešno sintetisanje manjih programa može značajno da olakša rad programerima. S obzirom na to da najveći deo vremena programeri provedu pišući manje delove koda, pa i automatska sinteza samo tih delova može značajno da ubrza njihov rad.

Literatura

[1] Bing Developer Assistant. on-line at https://marketplace.visualstudio.com/items?itemName=VisualStudioPlatformTeam.DeveloperAssistant.

[2] InSynth. on-line at https://www.insynth.co.uk.

[3] Syntax-Guided Synthesis. on-line at http://sygus.seas.upenn.edu/files/sygus_extended.pdf.

[4] Clark Barrett and Cesare Tinelli. Satisfiability Modulo Theories. on-line at http://homepage.divms.uiowa.edu/~tinelli/papers/BarTin-14.pdf.

[5] Alexandros Beskos and Andrew Stuart. MCMC Methods for Sampling Function Space. on-line at https://pdfs.semanticscholar.org/c591/557c399d21ad5c31ded0e44f47226b8e7253.pdf.

[6] Gianfranco Bilardi and Keshav Pingali. Algorithms for Computing the Static Single Assignment Form. on-line at http://iss.ices.utexas.edu/Publications/Papers/JACM2003.pdf.

[7] James Bornholt. Program Synthesis Explained, 2015. on-line at https://homes.cs.washington.edu/~bornholt/post/synthesis-explained.html.

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

[9] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. Qlose: Program Repair with Quantitative Objectives, 1995.

[10] Cristina David and Daniel Kroening. Program synthesis: challenges and opportunities. on-line at https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5597726.

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

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

[13] Sumit Gulwani, Vijay Anand Korthikanti, and Ashish Tiwari. Synthesizing geometry constructions, 2011.

[14] Barbara Jobstmann, Andreas Griesmayer, and Roderick Bloem. Program repair as a game, 2005.

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

[16] Aditya Krishna Menon, Omer Tamuz, Sumit Gulwani, Butler W. Lampson, and Adam Kalai. A machine learning framework for programming by example, 2013.

[17] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, , and Satish Chandra. *SemFix: program repair via semantic analysis.* ICSE, 2013.

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

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

[20] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models, 2014.

[21] Microsoft Research. The Z3 Theorem Prover. on-line at https://github.com/Z3Prover/z3.

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

[23] Rishabh Singh and Sumit Gulwani. Synthesizing number transformations from input-output examples, 2012.

[24] Martin T. Vechev, Eran Yahav, and Greta Yorsh. Abstraction-guided synthesis of synchronization, 2010.