Apstraktna interpretacija

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

Ozren Demonja, Stefan Maksimović, Marko Crnobrnja
{mi12319, mi12078, mi12024}@alas.matf.bg.ac.rs

1. april 2017.

Sažetak

U ovom tekstu predstavljena je metoda apstraktne interpretacije, objašnjeni uslovi njenog nastajanja i navedene njene primene u savremenom računarstvu za optimizaciju i verifikaciju softvera. Radi objašnjenja apstraktne interpretacije naveden je jedan neformalan neprogramski primer kao i celo poglavlje u kome je detaljno razmotren primer jednog C++ koda. Na kraju je data i matematička formalizacija koja ukazuje na valjanost upotrebe ove metode za pouzdanu verifikaciju softvera.

Sadržaj

1 Uvod

U protekle dve decenije se dosta toga promenilo u pogledu performansi računara. Današnji kućni računari su po snazi jači nego najmoćniji superračunari iz 70-ih. U međuvremenu, kroz paralelizovanje i inovacije u hijerarhiji memorije superračunari sada postižu 10 do 100 teraflopa(engl. floating point operations per second) [5].

Glavne krivce za ovakav porast brzine računara pronalazimo kroz dva aspekta. Prvi, osnovna tehnologija prema kojoj se računari konstruišu je doživela izuzetan napredak koji počiva na predviđanjima Murovog zakona (engl. Moore's law) [14]. Drugi aspekt je paralelizam u nekoj svojoj formi [5].

Ova poboljšanja u snazi nisu došla bez problema. Kako je arhitektura postajala kompleksnija nebi li mogla pratiti eksponencijalnu brzinu Murovog zakona, postajalo je sve teže i teže programirati. Većina vrhunskih programera je postala svesna potrebe da eksplicitno upravlja memorijom. U naporu da se poboljšaju performanse pojedinačnih procesa, programeri su učili kako da ručno transformišu njihov kod tako da se efikasnije izvrši planiranje instrukcija na višeprocesorskom sistemu [5].

U današnje vreme značajni deo koda u većini modernih kompajlera je posvećen optimizaciji generisanog koda. Često se dešava da je ponašanje pri izvršavanju optimizovanog koda nesaglasno sa pre-optimizovanim ponašanjem koda, drugim rečima optimizacija je uticala kako na semantiku programa tako i na pragmatiku. Ovaj problem se često dešava zbog nedovoljne strogosti koja je bila primenjena na ispravnost dokaza optimizacije. Za programske jezike sa definisanom matematičkom semantikom postoji rastući skup alata koji obezbeđuju osnovu za semantički korektnu transformaciju, jedan od tih alata je i apstraktna interpretacija [4].

2 Apstraktna interpretacija

Kao što se vidi iz prethodnog poglavlja apstraktna interpretacija je tehnika za automatsku statičku analizu programa. Sastoji se od zamene preciznih elemenata programa sa manje detaljnim apstrakcijama. Apstrakcija dovodi do gubitka sigurnih informacija, informacija koje su eksplicitno date, usred samog procesa apstrahovanja što dovodi do nemogućnosti dovođenja zaključaka za sve programe. Apstraktna interpretacija omogućava da otkrijemo greške nastale tokom izvršavanja programa, kao što su deljenje sa 0, prekoračenje, itd, a takođe otkriva korišćenje zajedničkih promenljivih i mrtvih petlji [4].

Glavna prednost alata koji koriste apstraktnu interpretaciju je da se test obavlja bez ikakve pripreme, baziran na kodu projekta. Ako se uporedi sa troškovima jediničnog testiranja, to predstavlja značajan argument [4].

2.1 Problem koji se rešava

Da bi se lakše shvatio problem prvo ćemo pokazati tri neprogramerska primera apstraktne interpretacije. Ovi primeri će služiti za uspostavljanje principa pristupa.

Pretpostavimo da želimo da putujemo negde. Jedna od odluka koju moramo napraviti je da li želimo da hodamo, vozimo se ili letimo. Umesto da ovu odluku sprovodimo metodom pokušaja i greške, mi ćemo koristiti osobinu putovanja, udaljenost (koju možemo izmeriti na mapi) da odlučimo koji je najbolji način transporta. Mapa je apstraktna reprezentacija putovanja i merenjem rastojanja mi apstrahujemo sam proces putovanja [4].

Slično ovom, može nam biti zadato da odredimo za neki broj da li je paran ili neparan. Sve što trebamo učiniti u tom slučaju jest videti da li je najmanje značajna cifra broja parna ili neparna - zadatak koji zahteva manje računarskog napora nego deliti celi broj za dva (osim ako je jednocifren broj).

Treći primer, formalniji, gradi se korišćenjem pravila znaka. Određujemo znak rezultata množenja. Ako se pitamo koji je znak
336 * (-398)
mi odmah znamo da je rezultat negativan. Bez izvođenja operacije množenja mi određujemo znak na osnovu pravila znaka. Znamo da će množenje pozitivnog i negativnog broja uvek proizvesti za rezultat negativan broj. Ovaj treći primer je malo bliži apstraktnoj interpretaciji kod programiranja tako da ćemo malo dublje zaći u njega [4].

Da bismo razumeli apstraktnu interpretaciju moramo da prebacimo zadatak u sledeću formu:

$$a_+ \times a_- \tag{1}$$

gde $\times$ predstavlja pravilo znaka pri množenju, a_+ pozitivan, a a_- negativan broj. Zatim u ovakvom zapisu izvodimo sledeće jednostavnije izraze:

$$\begin{gathered} 0 \times a_+ = 0 \times a_- = a_+ \times 0 = a_- \times 0 = 0 \\ a_+ \times a_+ = a_- \times a_- = a_+ \\ a_+ \times a_- = a_- \times a_+ = a_- \end{gathered} \tag{2}$$

Do sada nismo razmatrali korektnost interpretacije, ali treba da bude jasno da možemo dobiti potpuno tačne odgovore u oba primera. Ova situacija postaje mnogo nejasnija ako umesto množenja stavimo sabiranje. Prvih nekoliko redova ne predstavljaju neki problem

$$\begin{gathered} 0 \pm a_+ = a_+ \pm 0 = a_{+,0} \\ 0 \pm a_- = a_- \pm 0 = a_{-,0} \\ a_+ \pm a_+ = a_+ \\ a_- \pm a_- = a_- \end{gathered} \tag{3}$$

Ali ostatak je problematičan:

$$a_+ \pm a_- = ??$$
$$a_- \pm a_+ = ??$$
(4)

Ako bi stavili znak (0, +, -), a da ne znamo vrednosti u nekim slučajevima bi pogrešili jer odgovor zavisi od vrednosti na koje se primenjuje. Kako možemo da okarakterišemo pravi izbor za ?? ? Da bi mogli to da uradimo moramo da znamo koji znak u apstraktnom izračunavanju predstavlja:

$$a_0 = \{0\}$$
$$a_+ = \{n \mid n > 0\}$$
$$a_- = \{n \mid n < 0\}$$
(5)

Iz ovoga sledi da je apstraktni račun tačan ukoliko je stvarna vrednost član skupa koji apstraktna vrednost predstavlja. Ako je ovo slučaj kaže se da je apstraktna interpretacija sigurna. Ako koristimo a da predstavimo cele brojeve, dobijamo sigurnu verziju sabiranja dodavanjem pravila:

$$s \pm a = a \pm s = a \pm a = a \quad \text{gde je s} \in \{0, -, +\} \quad (6)$$

[4]

2.2 Korišćenje u računarstvu

Zbog čega je apstraktna interpretacija korisna u računarstvu? Mnogi tradicionalni optimizatori koji su zasnovani na toku upravljanja (engl. Control flow) i na analizi toka podataka (engl. Data-flow analysis) se uklapaju u okvir apstraktne interpretacije. Neke posebne analize koje su značajne u deklarativnim jezicima su:

- *Analiza strogosti* (engl. Strictness analysis[4]): Analiza koja omogućava optimizaciju lenjih funkcionalnih programa identifikujući parametre koji mogu biti prosleđeni po vrednosti. Na ovaj način se izbegava potreba za pravljenjem zatvorenja (engl. closure) i otvara se mogućnost paralelne evaluacije.
- *Analiza menjanja u mestu* (engl. In-place update analysis) [7]: Ova analiza nam omogućava da odredimo tačke u programu na kojima je sigurno da se uništi objekat. Objekat se uništava kada nema ni jedan pokazivač koji pokazuje na njega. Rezultate u ovoj oblasti je doneo Hudak. Značajan rezultat je, po prvi put, funkcionalna verzija kviksort (engl. quicksort) algoritma koja može da se pokrene u linearnom prostoru. [10]
- *Analiza moda* (engl. Mode analysis) [4]: Značajno povećanje performansi može se postići u Prologu ako se zna kako se logičke promenljive koriste u relaciji (kao ulazne, izlazne ili oba). Kada je deklarativna zajednica postala svesna apstraktne interpretacije, nove aplikacije su otkrivene.

Optimizacije zasnovane na apstraktnoj interpretaciji su verovatno tačne. Ako ovo prebacimo u gornje primere to bi bilo:

- Stroga analiza: Ako stroga analiza utvrdi da je funkcija stroga u argumentima onda to ona definitivno i jeste, ali analiza neće uspeti da detektuje neke parametre koji mogu biti prosleđeni po vrednosti.
- Analiza menjanja u mestu: Ako analiza menjanja u mestu ukaže da možemo destruktivno da ažuriramo podatke onda i možemo ali ćemo kopirati neke objekte koji su mogli biti uništeni. [10]
- Analiza moda: Analiza moda nekad neće uspeti da detektuje logičke promenljive koje se isključivo koriste kao ulazno-izlazne promenljive.

2.3 Aktuelna primena

Apstraktna interpretacija je široko primenjena u domenu analize i verifikacije koda gde postoji veliki broj aplikacija zasnovanih na njoj koja se već uveliko koriste u industriji[13]. Među poznatijim alatima tog tipa mogu se navesti Parasoft C/C++test, Frama-C i Polyspace [1].

U oblasti kompajlera, naprotiv, implementacija čiste apstraktne interpretacije znatno je ređa, uprkos ranom prepoznavanju potencijala za optimizaciju koda[8]. Posebno zanimljiv primer je CIAO[2], multiparadigmatski kompajler za jezike programiranja ograničenja namenjenih konkurentnom izvršavanju na više procesora. Njegov pretprocesor izvršava analize nad više različitih apstraktnih domena pomoću generičkog interpretatora koji radi s vrha naniže. [11].

Mnogi kompajleri poput GCC, Rosylyn, Clang i Stalin koriste analizu toka podataka ili neke njene aspekte za optimizaciju, ali treba imati na umu da iako se ova tehnika može posmatrati kao podvrsta apstraktne interpretacije, ona je znatno starija od nje i nije informisana teorijskim uvidima koje su otkrili bračni par Kuso i kasniji istraživači, te se oslanja na konkretne domene i ne koristi operatore proširenja.

3 Formalizacija

Označimo izvršno stanje programa u datom trenutku, pod čime se podrazumeva vrednost promenljivih kao i mesto u kodu do koga se došlo, odnosno na koje pokazuje programski brojač, sa $v \in V$ gde je V skup svih takvih stanja. Tada možemo primetiti binarnu relaciju prelaska stanja $v_0 \rightsquigarrow v_1$ koja predstavlja da stanje v_1 može uslediti za stanjem v_0. [3]

Bitno je napomenuti da se ova relacija ne može zameniti funkcijom koja bi slikala jedno stanje u iduće, jer prelazak može zavisiti od okolnosti koje nisu definisane unutar programa, poput učitavanja podataka ili redosleda izvršavanja instrukcija u slučajevima kada program ima više niti, tako da može postojati više različitih stanja u koje jedno stanje prelazi. Posebno su nam zanimljiva stanja

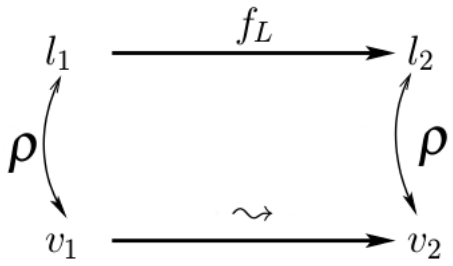
$$\ldots \rightsquigarrow v_n \rightsquigarrow v_n$$

koja odgovaraju zaustavljanjima programa.

[1] Parasoft: https://www.parasoft.com/product/cpptest/, Frama-C: http://frama-c.com/, Polyspace: https://www.mathworks.com/products/polyspace.html?s_cid=wiki_polyspace_2

[2] http://ciao-lang.org/

[3] Ovo poglavlje se primarno oslanja na [2], gde se mogu naći dokazi tvrdnji koji su ovde izostavljeni radi sažetosti.



Slika 1: Odnos između apstraktnih i konkretnih stanja

Budući da je u opštem slučaju jedini način da odredimo v da izvršimo sam program, uopštićemo problem uvođenjem pojma prostora svojstava L. Njegovi elementi $l \in L$, koje nazivamo apstraktnim stanjima, obuhvataće svojstva koja stanja u koja program dospeva u datom trenutku imaju. Potrebno je naglasiti da dato apstraktno stanje ne predstavlja svojstva jednog konkretnog stanja, već više konkretnih stanja te da pojedinačne promenljive apstraktnih stanja uzimaju vrednosti iz partitivnog domena odnosno skupa podskupova domena promenljivih u konkretnim stanjima.[9]

Nad prostorom svojstava već možemo definisati funkciju $f_L : L \to L$ koja slika apstraktno stanje u ono koje mu sledi.

Pokazaće se korisno definisati dodatnu strukturu nad L:

Definicija 1. *Neka je nad L definisana relacija poretka, odnosno relacija $\sqsubseteq$ takva da za sve $a,b,c \in L$ važi*

1. *$a \sqsubseteq a$ (Refleksivnost)*
2. *ako su $a \sqsubseteq b$ i $b \sqsubseteq a$ tada $a = b$ (Antisimetričnost)*
3. *ako su $a \sqsubseteq b$ i $b \sqsubseteq c$ tada $a \sqsubseteq c$ (Tranzitivnost)*

Ukoliko za svaki podskup $L\prime \subseteq L$ postoji najmanja gornja granica $\bigsqcup L'$ i najveća donja granica $\bigsqcap L'$ tada se L naziva potpunom mrežom. *[12]*

Relacija poretka koju uvodimo nad prostorom svojstava je takva da su veći elementi opštiji od manjih, odnosno da predstavljaju slabije tvrđenje o stanju programa. Tada $\bigsqcup L'$ predstavlja disjunkciju apstraktnih stanja u L' odnosno stanje u kome važe bilo koja od datih svojstava, dok $\bigsqcap L'$ označava stanje u kome sva svojstva važe. Bitne vrednosti su takođe i $\bigsqcup L = \top$ i $\bigsqcap L = \bot$.

Sada želimo dovesti u vezu konkretna stanja programa sa apstraktnim stanjima koja ih modeliraju putem relacije $\rho \subseteq V \times L$ kao što je prikazano na slici 1 [4]. Zahtevamo sledeće od ove relacije:

1. $\forall v, l_1, l_2, (v \,\rho\, l_1) \vee (l_1 \sqsubseteq l_2) \Rightarrow (v \,\rho\, l_2)$
2. $\forall v, L' \subseteq L, (\forall l \in L', (v \,\rho\, l)) \Rightarrow v \,\rho\, (\bigsqcap L')$

Ovakvu relaciju nazivamo *relacijom ispravnosti*. Da bi dokazali njenu valjanost u konkretnom slučaju, dovoljno je dokazati je za početno stanje izvršavanja i pokazati da se valjanost čuva pri svakom prelasku u iduće stanje.

[4] slika je preuzeta sa izmenama iz [2]

3.1 Fiksne tačke

Ukoliko bismo želeli saznati svojstva programa l u nekoj tački izvršavanja, najdirektniji i najprecizniji način bi bio da izračunamo sva apstraktna stanja $l_i \in W(l)$ dobijena duž putanja izvršavanja koja vode do te tačke od početnog stanja l_0 i zatim nađemo $\bigsqcup W(l) = l$.

Nažalost, u praksi je takav račun nemoguć ili makar veoma zahtevan. Umesto toga, računaju se fiksne tačke funkcije $x = f_L(x)$ koje takođe čine potpunu mrežu pod uslovom da je f_L *monotona* [3], odnosno da važi

$$\forall l_1, l_2, l_1 \sqsubseteq l_2 \Rightarrow f_L(l_1) \sqsubseteq f_L(l_2)$$

Ako je uz to funkcija i *neprekidna*, $\bigsqcup f_L[L'] = f_L[\bigsqcup L']$, tada se najmanja fiksna tačka može izračunati kao

$$\bigsqcup \{ f_L^n(\bot) \}_{n\in\mathbb{N}} \quad \text{gde je} \quad f_L^0(\bot) = \bot \quad \text{i} \quad f_L^n(\bot) = f_L(f_L^{n-1}(\bot)) \quad [1]$$

Ipak, i ovom slučaju niz $f_L^n(\bot)$ može previše sporo konvergirati, zbog čega uvodimo još jedan, grublji, prostor svojstava M koji će služiti kao apstrakcija za L. Da bi objasnili odnos između ova dva prostora, moramo uvesti koncept galoaove veze:

Definicija 2. *Neka su $(A, \geqslant)$ i $(B, \geqslant)$ parcijalno uređeni skupovi a $F : A \to B$ i $G : B \to A$ monotone funkcije. Tada je $\langle A, F, G, B\rangle$* galoaova veza *ukoliko važi*

1. $\forall a \in A, a \leqslant G(F(a))$
2. $\forall b \in B, b \geqslant F(G(b))$

Teorema 1. *Ako između L i M postoji galoaova veza $\langle L, \alpha, \gamma, B\rangle$ tada je $\rho' \subseteq M \times V$, takva da*

$$m \,\rho'\, v \iff \gamma(m)\, \rho\, v$$

takođe relacija ispravnosti.

Druga tehnika je korišćenje operatora proširenja $\nabla : L \times L \to L$ takvog da je $x, y \sqsubseteq x \nabla y$ za sve x, y i pomoću koga se za bilo koji rastući niz $(y_n)_n$ može napraviti niz

$$(x'_n)_n \quad \text{gde je} \quad x'_0 = y_0 \quad \text{i} \quad x'_n = x'_{n-1} \nabla y_n$$

takav da konvergira u konačnom broju koraka.

Najčešće za funkciju prelaska f_L pravimo niz $(f_\nabla^n)_n$ takav da:

$$f_\nabla^n = \begin{cases} \bot, & \text{za} \quad n = 0 \\ f_\nabla^{n-1} & \text{za} \quad n > 0 \quad \text{i} \quad f_L(f_\nabla^{n-1}) \sqsubseteq f_\nabla^{n-1} \\ f_\nabla^{n-1} \nabla f_L(f_\nabla^{n-1}) & \text{inače} \end{cases}$$

Ovime efektivno ubrzavamo nizove koji rastu a zaustavljamo ih u suprotnom, time se sprečava zaglavljivanje prilikom analizi petlji i drugih cikličnih tokova upravljanja. Za limes ovog niza ispostavlja se da je veći od najmanje fiksne tačke, te da ga dobro aproksimira.

3.2 Pro upotrebe operatora proširenja

Razmotrimo sada

Listing 1: Primer koda 2

```
1 x = 1;
2 while (x<10000) {
3         x = x + 2;
4 }
```

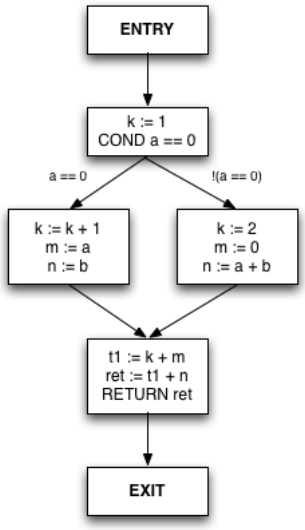
4 Primeri

Kako bismo bolje približili pojam apstraktne interpretacije, koristićemo se optimizacijskom tehnikom koju prevodioci koriste zvanu *propagacija konstanti*. Sam proces propagacije konstanti se svodi na menjanje promenljivih u izrazima konstantnim vrednostima, u slučaju kada njihove vrednosti ne zavise niti od ulaznih parametara funkcije, niti od dela koda koji se izvršavao u toj funkciji.

Listing 2: Kod sa grananjem

```
1  int foo(int a, int b) {
2    int k = 1;              // k je konstantno: 1
3    int m, n;
4    if (a == 0) {
5      ++k;                  // k je konstantno: 2
6      m = a;
7      n = b;
8    } else {
9      k = 2;                // k je konstantno: 2
10     m = 0;
11     n = a + b;
12   }
13   return k + m + n;       // k je konstantno: 2
14 }
```

4.1 Grafovi kontrole toka



Slika 2: Primer grafa kontole toka

Svaki deo koda 2 se može predstaviti *grafom kontrole toka (eng. control flow graph, CFG)*, i upravo takva reprezentacija će se koristiti za opis apstraktne interpretacije. Slika 2 prikazuje funkciju predstavljenu na ovaj način, koju ćemo

koristiti za opis ovog metoda statičke analize.
Neke napomene:

- Veze između čvorova grafa su predstavljene kao grane, pojavljuju se u slučaju kada imamo naredbe kontrole toka(petlje, uslovni/bezuslovni skokovi itd.).
- Naredbe imaju najviše jednu dodelu i tačno jednu operaciju. U slučaju naredbe sa više operacija, vršimo razdvajanje na naredbe sa po jednom operacijom, čije ćemo rezultate čuvati u pomoćnim promenljivama.

Terminologije:

- Svaki čvor se zove **osnovni blok** *(eng. basic block,* ***BB****)*. Osnovni blok se definiše tako što ima samo jednu tačku ulaza, i jednu tačku izlaza, što će reći da nema grananja unutar osnovnih blokova.
- Naredbe ćemo zvati *instrukcije*, iako one uopšteno mogu imati različite nazive u zavisnosti od toga koliko operanada primaju.
- **Tačka u programu** je zamišljena tačka pre ili posle svake instrukcije. Funkcija ima dobro definisano stanje u svakoj tački, tako da će se naša analiza programa uvek referisati na ove tačke.

4.2 Konkretna interpretacija

Pre nego što pređemo kroz primer koji opisuje proces apstraktne interpretacije, napravićemo jedan prolaz kroz funkciju koju ispitujemo koristeći konkretne vrednosti kao ulaz. Pritom ćemo pratiti promenljive unutar funkcije, obraćajući pažnju na one koje su zadržale istu vrednost tokom izvršavanja. Uzmimo sledeće vrenosti za ulaz `a = 0, b = 7`:

```
(instrukcija)        (stanje interpretatora posle instrukcije)
ENTRY                a = 0, b = 7
k := 1               a = 0, b = 7, k = 1
COND a == 0          (TRUE)
k := k + 1           a = 0, b = 7, k = 2
m := a               a = 0, b = 7, k = 2, m = 0
n := b               a = 0, b = 7, k = 2, m = 0, n = 7
t1 := k + m          a = 0, b = 7, k = 2, m = 0, n = 7, t1 = 2
ret := t1 + n        a = 0, b = 7, k = 2, m = 0, n = 7, t1 = 2, ret = 9
RETURN ret
EXIT
```

Dakle, textttk = 2 pre nego što se koristi u naredbi `t1 := k + m`. Možemo pokrenuti funkciju za ostale ulaze i dobili bismo isti rezultat. Međutim, ovakav način testiranja nam ne može potvrditi da je `k = 2` za sve moguće ulaze. (Doduše može, ali samo ako bismo proverili sve moguće vrednosti koje promenljiva može da uzme.)

4.3 Približavanje apstraktnoj interpretaciji

Obratimo pažnju da za `k`, u okviru funkcije imamo dve bitne provere: `a == 0` i `a != 0`, dok nam vrednost za `b` u ovom slučaju nije bitna. Sada ćemo ispitati vrednost promenljivih u okviru funkcije za dva ulaza: jedan sa ulazom `a = 0, b = ?`, a drugi sa ulazom `a = NN, b = ?`, gde `NN` označava ne-nula vrednost, dok `?` označava bilo koju vrednost. Počnimo sa `a = 0, b = ?`:

```
(instrukcija)        (stanje interpretatora posle instrukcije)
ENTRY                a = 0, b = ?
k := 1               a = 0, b = ?, k = 1
```

```
COND a == 0          (TRUE)
k := k + 1           a = 0, b = ?, k = 2
m := a               a = 0, b = ?, k = 2, m = 0
n := b               a = 0, b = ?, k = 2, m = 0, n = ?
t1 := k + m          a = 0, b = ?, k = 2, m = 0, n = ?, t1 = 2
ret := t1 + n        a = 0, b = ?, k = 2, m = 0, n = ?, t1 = 2, ret = ?
RETURN ret
EXIT
```

Primetimo da nam se sada pojavljuju `NN` i `?`, apstraktne vrednosti koje predstavljaju skupove konkretnih vrednosti. Ostaje još da utvrdimo kako se ponašaju operatori nad apstraktnim vrednostima. Na primer, u poslednjem koraku, `ret := t1 + n` postaje `ret := 2 + ?`. Kako bismo saznali šta ovo znači, posmatramo skupove konkretnih vrednosti: Ako `?` može biti bilo koja vrednost, onda i `2 + ?` takođe može uzeti bilo koju vrednost, tako da `2 + ?` $\longrightarrow$ `?`. Preostali slučaj testira `a = NN, b = ?`:

```
(instrukcija)        (stanje interpretatora posle instrukcije)
ENTRY                a = NN, b = ?
k := 1               a = NN, b = ?, k = 1
COND a == 0          (FALSE)
k := 2               a = NN, b = ?, k = 2
m := 0               a = NN, b = ?, k = 2, m = 0
n := a + b           a = NN, b = ?, k = 2, m = 0, n = ?
t1 := k + m          a = NN, b = ?, k = 2, m = 0, n = ?, t1 = 2
ret := t1 + n        a = NN, b = ?, k = 2, m = 0, n = ?, t1 = 2, ret = ?
RETURN ret
EXIT
```

Posle ovog koraka se vidi da su pokriveni svi mogući ulazi, kao i svaki deo koda funkcije, stoga možemo zaključiti da je `k = 2` uvek tačno pre nego što dođemo do naredbe `t1 := k + m`.
Procedura koju smo upravo ispratili daje određen uvid kako bismo krenuli u proces apstraktne interpretacije, ali nismo generalizovali samu proceduru. Tačno smo znali koje apstraktne vrednosti da koristimo za test slučajeve, i to smo mogli samo zato što smo imali kao primer jednostavnu funkciju. Ova metoda neće biti primenjiva na komplikovane funkcije, i nije automatizovana.
Drugi problem je što smo posmatrali svaku granu funkcije posebno. Funckija sa `k` iskaza može imati i do 2^k grana, dok funkcija sa petljama ih može imati i beskonačno, i ovo nam onemogućava da imamo kompletnu pokrivenost.

4.4 Apstraktna interpretacija kroz primer

Pre nego što krenemo u sam proces apstraktne interpretacije, treba da odaberemo skupove apstraktnih vrednosti. Ponekad ne znamo kako da odaberemo skupove takvih vrednosti, tako da ćemo sada proći kroz funkciju bez odabira ikakvih vrednosti: Dakle, pokušajmo sa sledećim ulazom: `a = ?, b = ?`:

```
(instrukcija)        (stanje interpretatora posle instrukcije)
ENTRY                a = ?, b = ?
k := 1               a = ?, b = ?, k = 1
COND a == 0
```

Pošto nemamo informaciju o tome šta je `a`, krenućemo put obe grane. Prvo za potvrdnu granu:

```
(instrukcija)        (stanje interpretatora posle instrukcije)
                     a = ?, b = ?, k = 1
```

```
k := k + 1          a = ?, b = ?, k = 2
m := a              a = ?, b = ?, k = 2, m = ?
n := b              a = ?, b = ?, k = 2, m = ?, n = ?
```

Potom za negativni slučaj:

```
(instrukcija)       (stanje interpretatora posle instrukcije)
                    a = ?, b = ?, k = 1
k := 2              a = ?, b = ?, k = 2
m := 0              a = ?, b = ?, k = 2, m = 0
n := a + b          a = ?, b = ?, k = 2, m = 0, n = ?
```

U ovoj tački, dva izvršna toka se spajaju. Mogli bismo da nastavimo da ih testiramo ponaosob, ali znamo da će to dovesti do eksplozije u uopštenom slučaju, tako da ćemo izvršiti spajanje stanja. Potrebno nam je jedno stanje koje pokriva obe grane:

```
a = ?, b = ?, k = 2, m = ?, n = ?
a = ?, b = ?, k = 2, m = 0, n = ?
```

Ovo stanje možemo dobiti tako što ćemo spajati promenljivu po promenljivu. Na primer, `k` je 2 u jednom i u drugom stanju, tako da je `k = 2` u rezultujućem stanju. Za `m`, ono može biti bilo šta u prvom stanju, tako da iako je ono 0 u drugom stanju, može uzeti bilo koju vrednost u rezultujućem stanju. Kao rezultat dobijamo:

```
a = ?, b = ?, k = 2, m = ?, n = ?
```

Možemo nastaviti izvršavanje u jednom toku:

```
t1 := k + m         a = ?, b = ?, k = 2, m = ?, n = ?, t1 = ?
ret := t1 + n       a = ?, b = ?, k = 2, m = ?, n = ?, t1 = ?, ret = ?
RETURN ret
EXIT
```

Gde dobijamo odgovor koji smo želeli, `k = 2`. Osnovne ideje su bile:

- Proći kroz funkciju koristeći apstraktne vrednosti kao ulaz
- Apstraktna vrednost predstavlja skup konkretnih vrednosti
- Kod kontrole toka gde imamo grananje, krenimo put obe grane
- Gde imamo spajanje, spajamo izlaz iz obe grane

[15]

4.5 Primer upotrebe operatora proširenja

Razmotrimo sada primer 3. Jasno je da naša analiza ne može proći kroz petlju svih 10000 puta, u opštem slučaju mi ni ne možemo znati postoji li izlaz iz date petlje. Zato je potrebno upotrebiti gorepomenuti operator proširenja, u ovom slučaju takav da se desni kraj intervala proširi do beskonačnosti što nam daje fiksnu tačku operacije koja se izvršava unutar petlje.

Listing 3: Kod sa petljom

```
1 x = 1;
2 while (x<10000) {
3         x = x + 2;
4 }
5
```

Na kraju, dobijena stanja ograničavamo uslovom petlje koji zahteva se x nalazi unutar $(-\infty, 10000)$ na početku petlje a unutar $[10000, +\infty)$ na njenom izlazu. Na tabeli 1 prikazane su vrednosti dobijene u toku apstraktne interpretacije. [5]

Tabela 1: Apstraktne vrednosti promenljive x po linijama koda i koracima apstraktne interpretacije.

stanje	početak	prvi prolaz	drugi prolaz	proširenje	uslov
x_1	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$
x_2	$[]$	$[1,1]$	$[1,1]$	$[1,1]$	$[1,1]$
x_3	$[]$	$[1,1]$	$[1,3]$	$[1,+\infty)$	$[1,9999]$
x_4	$[]$	$[3,3]$	$[3,5]$	$[3,+\infty)$	$[3,10001]$
x_5	$[]$	$[]$	$[]$	$[]$	$[10000,10001]$

5 Zaključak

U ovom radu smo pokušali da predstavimo tehniku apstraktne interpretacije na način koji će biti razumljiv onima koji nisu imali pređašnjeg kontakta sa teorijom verifikacije programa ili semantičke analize. Prišli smo temi iz neformalnog, tehničkog i formalno-matematičkog ugla. Iako nam ovakva podela nije omogućila da zađemo dublje u materiju, nadamo se da je zahvaljujući njoj čitalac našao u skladu sa svojim sklonostima nešto što bi ga zainteresovalo za dalje proučavanje ove oblasti. Jer apstraktna interpretacija je nesumnjivo korisna i visoko prilagodljiva metoda koja pored sadašnjosti ima i svoju budućnost.

Literatura

[1] Baranga A. The contraction principle as a particular case of Kleene's fixed point theorem. *Discrete Mathematics*, 1991.

[2] Sălcianu A. Notes on Abstract Interpretation. Neobjavljeni rad, Novembar 2001.

[3] Tarski A. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 1955.

[4] S. Abramsky and C. Hankin. An introduction to abstract interpretation. In *Abstract Interpretation for Declarative Languages*, pages 5–41. Ellis Horwood, 1990.

[5] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann Publishers, 2001.

[6] J.-L. Boulanger. *Static analysis of software : the abstract interpretation*. Wiley, London, 2011.

[7] D. C. Cann. *Compilation techniques for high per- formance applicative computation*. PhD thesis, Colorado State University, jul 1989.

[8] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[5] Primer je preuzet iz [6]

[9] Stoy J. E. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1981.

[10] J. Y. Girard and Y. Lafont. Linear logic and lazy computation. In Hartmut Ehrig, Robert Kowalski, Giorgio Levi, and Ugo Montanari, editors, *TAPSOFT '87: Proceedings of the International Joint Conference on Theory and Practice of Software Development Pisa, Italy, March 23–27, 1987*, pages 52–66, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

[11] M. García de la Banda Manuel V. Hermenegildo, Francisco Bueno Carrillo and Alvaro Germán Puebla Sánchez. The CIAO multiparadigm compiler and system: A progress report. Technical report, Technical University of Madrid, 1995.

[12] Burris S. N. and Sankappanavar H.P. *A Course in Universal Algebra*. Springer-Verlag., 2012.

[13] Björn Wachter Reinhard Wilhelm. Abstract Interpretation with Applications to Timing Validation. In *Computer Aided Verification*. jul 2008.

[14] Robert R. Schaller. Moore's Law: Past, Present, and Future. *IEEE Spectr.*, 34(6):52–59, jun 1997.

[15] Mozilla wiki. Abstract Interpretation, 2009. dostupno na: https://wiki.mozilla.org/Abstract_Interpretation.