Apstraktna interpretacija

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

Ozren Demonja, Stefan Maksimović, Marko Crnobrnja
mi12319@alas.matf.bg.ac.rs, mi12078@alas.matf.bg.ac.rs, 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 preformansi računara. Današnji kućni računari su jači nego najmoćniji superračunari iz 70-ih. U međuvremenu, kroz paralelizovanje i inovacije u hijerarhiji memorije superračunari sada postižu 10 do 100 teraflopa(eng. floating point operations per second). [5]

Glavni krivci za ovakvo poboljšanje u brzini računara su dva aspekta. Prvi, osnovna tehnologija prema kojoj se računari konstruišu je doživela izuzetan napredak koji počiva na predviđanjima Murovog zakona (eng. Moore's law) [9]. Drugi aspekt je paralelizam u nekoj svojoj formi [5].

Ova poboljšanja u snazi nisu došla bez problema. Kako je arhitektura postajala sve više i više kompleksna da bi 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 preformanse 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 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 optimitacije. 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. Sastoji se od zamene preciznih elemenata programa sa manje detaljnim apstrakcijama. Apstrakcija dovodi do gubitka sigurnih informacija, što dovodi do nemogućnosti dovođenja zaključaka za sve programe. Apstraktna interpretacija omogućava da otkrijemo runtime greške, kao što su delenje 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 iakve 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 dva neprogramerska primera apstraktne interpretacije koja 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. Ume-

sto 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.

Drugi primer, malo više formalan, se gradi korišćenjem pravila zanak. Određujemo znak rezultata množenja. Ako se pitamo koji je znak

336 * (-398)

mi odmah znamo da je rezultat negativan. Bez da izvodimo množenje pa određujemo znak mi na osnovu pravila znaka znamo da će množenje pozitivnog i negativnog broja uvek proizvesti za rezultat negativan broj. Ovaj drugi primer je malo bliži apstraktnoj interpretaciji kod programiranja tako da ćemo malo dublje zaći u njega.

Da bi smo razumeli apstraktnu interpolaciju moramo da prebacimo zadatak u sledeću formu:

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

gde $\times$ predstavlja pravilo znaka pri množenju, a_+ pozitivan i a_- negativan broj

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

i onda izvodimo ove jednostavije izraze. Do sada nismo razmatrali korektnost interpretacije ali treba da bude jasno da mozemo 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:

$$\begin{gathered} a_+ \pm a_- = ?? \\ a_- \pm a_+ = ?? \end{gathered} \tag{4}$$

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 mozemo da okarakterišemo pravi izbor za ??. Da bi mogli to da uradimo moramo da znamo koji znak u apstraktnom izracunavanju predstavlja:

$$\begin{aligned} a_0 &= \{0\} \\ a_+ &= \{n \mid n > 0\} \\ a_- &= \{n \mid n < 0\} \end{aligned} \tag{5}$$

Onda je apstraktna kalkulacija tačna ako je pravi odgovor član skupa koji apstraktni odgovor 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:

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

2.2 Korišćenje u računarstvu

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

- Stroga analiza: Analiza koja omogucava optimizaciju lenjih funkcionalnih programa identifikujući parameter koji mogu biti prosleđeni po vrednosti tako da se izbegne potreba za pravljenjem zatvorenja (eng. closure) i otvara se mogućnost paralelne evaluacije.
- Analiza menjanja u mestu: Ova analiza nam omogućava da odredimo tacke u programu na kojima je sigurno da se uništi objekat jer ni jedan pokazivač ne pokazuje na njega. Rezultate u ovoj oblasti je doneo Hudak. Značajan rezultat je, po prvi put, funkcionalna verzija kviksort algoritma koja može da se pokrene u linearnom prostoru. [7]
- Analiza relevantnih klauza: U mnogim prototipovima 5. generacije arhitekture programi mogu da naprave nelokalni pristup definicijama funkcija. Ovo povlači da postoji komunikacija povezana sa izvrsavanjem programa. Korišćenje analize delova postaje moguce identifikovati delove definicije funkcije koji su relevantni za naš program i tako smanjiti troškove.
- Analiza moda: Znacajno povećanje performansi može se postići u Prologu ako se zna kako se logicke varijable 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. [7]
- Analiza relevantnih klauza: Analiza relevantnih klauza će nas terati da komuniciramo sa supersetom koda koji je u stvari samo potreban za neke posebne aplikacije.
- Analiza moda: Analiza moda nekad neće uspeti da detektuje logičke promenjljive koje se isključivo koriste kao ulazno-izlazne promenjljive.

3 Primeri

Objasnićemo apstraktnu interpretaciju na primeru propagacije konstanti. Cilj nam je da otkrijemo u svakoj tački funkcije da li bilo koja od promenljivih koja se koristi u toj tački ima konstantnu vrednost, tj. da li ima istu vrednost nezavisno od ulaznih parametara funkcije i nezavisno od toga koji deo koda je izvršen u toj funkciji. Prevodioci koriste ovaj tip analize za optimizaciju propagacije konstanti, što znači menjanje konstantnih promenljivih kontantama. Ovo je primer C++ koda koji ćemo analizirati, sa komentarima koji ukazuju na konstantne promenljive.

Listing 1: Primer koda

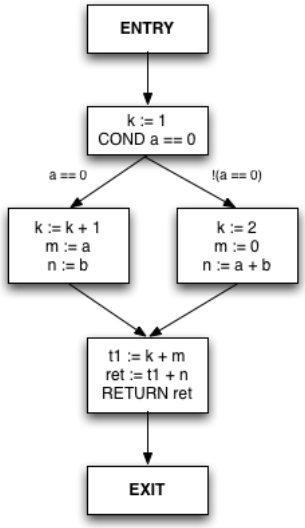
```
1  int foo(int a, int b) {
2    int k = 1;          // k je konstantno: 1
3    int m, n;
4    if (a == 0) {
5      ++k;              // k je konstantno: 2
6      m = a;
7      n = b;
8    } else {
9      k = 2;            // k je konstantno: 2
10     m = 0;
11     n = a + b;
12   }
13   return k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

3.1 Grafovi kontrole toka

Apstraktna interpretacija se obavlja nad dijagramom koji predstavlja funkciju koju ispitujemo, i zove se graf kontrole toka (eng. control flow graph, CFG). Na slici 1 je prikazan graf funkcije koju ćemo ispitivati.
Neke napomene:

- Svi mogući prelazi su prikazani kao ivice, tj. veze između čvorova koji sadrže kod
- Naredbe imaju tačno jednu operaciju i najviše jednu dodelu. Privermene promenljive se dodaju po potrebi.

Terminologija:



Slika 1: Primer grafa kontole toka

- 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.

3.2 Konkretna interpretacija

Kako bismo objasnili apstraktnu interpretaciju, počećemo prvo sa primerom konkretne interpretacije. Kasnije ćemo se nadograditi na ovaj primer kako bismo objasnili apstraktnu interpretaciju.
Mogli bismo početi tako što bismo zvali funkciju za različite ulaze, i potom gledali koje su sve promenljive konstantne kroz sve te pozive. Počnimo tako što ćemo pokrenuti program za ulaze `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, `k = 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 za svaki od 2^{64} ulaza.)

3.3 Približavanje apstraktnoj interpretaciji

Ako pogledamo prethodnu funkciju, možemo primetiti da postoje samo dva bitna slučaja: `a == 0`, `a != 0`, dok `b` nije bitno. Pokrenimo dva testa: jedan sa ulazom `a = 0, b = ?`, a drugi sa ulazom `a = NN, b = ?`, gde NN odznačava 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
```

Ovo izgleda poprilično isto kao i konkretan primer, samo što su sada neke vrednosti apstrahovane, `NN` i `?`, koje predstavljaju skupove konkretnih vrednosti.
Takođe moramo da znamo šta opeatori rade 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 + ?  -> ?`. 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
```

Sada smo testirali za svaki mogući ulaz, kao i svaku granu koda funkcije. Ovo je dokaz da `k = 2` je uvek tačno pre nego dođemo do `t1 := k + m`.
Procedura koju smo upravo ispratili daje određen uvid kako bismo bismo krenuli u proces apstraktne interpretacije, ali nismo generalizovali samu proceduru. Tačno smo zali 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 funckije, 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.

3.4 Apstraktna interpretacija kroz primer

Jedan od problema sa gornjim pristupom apstraktnoj interpretaciji je bio što nismo znali kako da odaberemo skupove apstraktnih vrednosti koje ćemo koristiti kao ulaz za test primere. Pokušajmo da sprovedemo jedan test gde nećemo birati takve skupove, dakle pokušajmo sa sledećim ulazom: `a = ?, b = ?`:

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

Šta sada? Nemamo informaciju o tome šta je `a`, tako da ne znamo kojom granom treba da idemo. Odabraćemo obe. 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

[10]

4 Formalizacija

Označimo, za početak, izvršno stanje programa u jednom momentu, 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. [1]

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.

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.[6]

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)

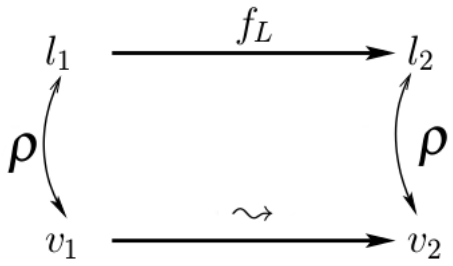
Ukoliko za svaki podskup $L\prime \subseteq L$ postoji najmanja gornja granica $\bigsqcup L'$ i najveća donja granica $\bigsqcap L'$ tada se L naziva potpunom mrežom. [8]

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 2 [2]. Zahtevamo sledeće od ove relacije:

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

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



Slika 2: Odnos između apstraktnih i konkretnih stanja

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 pokazat da se valjanost očuvava pri svakom prelasku u iduće stanje.

4.1 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 $\nabla : L \to L$ 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

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

takav da konvergira u konačnoj 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.

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.

[2] Sălcianu A. Notes on Abstract Interpretation.

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

[4] S. Abramsky and C. Hankin. An introduction to abstract interpretation, 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] Stoy J. E. Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. MIT Press, 1981.

[7] 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.

[8] Burris S. N. and Sankappanavar H.P. A Course in Universal Algebra. Springer-Verlag.

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

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