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). [2]

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) [4]. Drugi aspekt je paralelizam u nekoj svojoj formi [2].

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. [2]

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. [1]

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. [1]
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. [1]

2.1 Problem koji se rešava

Da bi se lakše shvatio problem prvo će mo pokazati dva ne programerska 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 će mo 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 krisćenjem 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 će mo malo dublje zaći u njega.

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

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

gde $\times$ predstavlja pravilo znaka pri mnozenju

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

i onda izvodimo ove jednostavije izraze. Do sada nismo razmatrali korektnost interpretacije ali treba da bude jasno da mozemo dobiti potpuno tacne odgovor u oba primera. Ova situacija postaje mnogo nejasnija ako umesto mnozenja stavimo sabiranje. Prvih nekoliko redova ne predstavljaju neki problem

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

Ali ostatak je problematican:

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

Ako bi stavili znak (0, +, -) a da ne znamo vrednosti u nekim slucajevima bi pogresili jer odgovor zavisi od vrednosti na koje se primenjuje. Kako mozemo da okarakterisemo pravi izbor za ??. Da bi mogli to da

uradimo moramo da znamo koji znak u abstraktnom izracunavanju predstavlja:

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

Onda je apstraktna kalkulacija tacna ako je pravi odgovor clan seta koji apstraktni odgovor predstavlja. Ako je ovo slucaj mi kazemo da je apstraktna interpretacija sigurna. Ako koristimo ?? da predstavimo cele brojeve, dobijamo sigurnu verziju sabiranja dodavanjem pravila:

$$s \pm ?? \quad = ?? \quad \pm \quad s \quad = ?? \pm ?? \quad = ?? gdes \quad \in \quad [0, -, +] \tag{6}$$

2.2 Koriscenje u racunarstvu

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

Stroga analiza: Analiza koja omogucava otimizaciju lenjih funkcionalnih programa identifikujuci parameter koji mogu biti prosledjeni po vrednosti tako da se izbegne potreba za pravljenjem klozura (eng. closure) I otvara se mogucnost paralelne evaluacije.

Analiza menjanja u mestu: Ova analiza nam omogucava da odredimo tacke u programu na kojima je sigurno da se unisit objekat jer ni jedna pokazivac ne pokazuje na njega. Rezultate u ovoj oblasti je doneo Hudak. Znacajan rezultat je, po prvi put, funkcionalna verzija kviksort algoritma moze da se pokrene u linearnom prostoru. [3]

Analiza relevantnih klauza: U mnogim prototipovima 5 generacije arhitekture programi mogu da naprave ne-lokalni pristup definicjama funkcija. Ovo povlaci da postoji komunikacija povezan sa izvrsavanjem programa. Koriscenje analize delova postaje moguce identifikovati delove definicje funkcije koji su relevantni za nas program i tak smanjiti troskove.

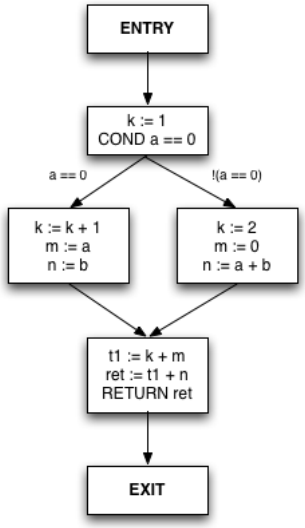
Analiza moda: Znacajno povecanje preformansi moze se postici u Prologu ako zna kako se logicke varijable koriste ue relaciji (kao ulazne, izlazne ili mesavina ovo dvoje). Kada su deklarativna zajednica postala svesna apstraktne interpretacije, nove aplikacije su otkrivene. Optimizacije bazirane na apstraktoj interpretaciji su verovatno tacne. Ako ovo prebacimo u gornje primere to bi bilo:

3 Formalizacija

4 Primeri

4.1 Uvod

Objasnićemo apstraktnu interpretaciju na primeru *propagacije konstanti*. Cilj nam je da otkrijemo u svakoj tački funkcije da li bilo koja od



Slika 1: Primer grafa kontole toka

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 }
```

4.2 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:

- 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.

4.3 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.)

4.4 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.

4.5 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

[5]

5 Zaključak

Ovde pišem zaključak.

Literatura

[1] S. Abramsky and C. Hankin. *An introduction to abstract interpretation*, pages 5–41. Ellis Horwood, 1990.

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

[3] 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.

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

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

A Dodatak

Ovde pišem dodatne stvari, ukoliko za time ima potrebe.