

Analiză statică  
Analiza fluxului de date

23 octombrie 2014

## Analiză statică: definiție

O analiză a *codului sursă* (fără a executa programul), cu scopul de a determina *proprietăți* ale programului sursă.

(în principal corectitudinea, dar și performanță, etc.)

Complementare analizelor *dinamice* (prin rularea codului)

Exemple de proprietăți:

variabile neinițializate (particular: pointeri nuli)

atribuiri nefolosite

vulnerabilități de cod (excepții, depășiri de indici), etc.

De obicei, analizele statice sunt legate de *semantica* programului  
uneori: și analize limitate la *structura* (sintactică) a codului

Istoric:

(sub)domeniu legat de *compilatoare*: în special pentru optimizare  
mai recent: în *proiectarea limbajelor*; pentru *detectarea de erori*

# Analiza fluxului de date

Tehnici cu originea în domeniul compilatoarelor folosite pentru *generarea* de cod (alocarea de regiștri) și *optimizarea* de cod (propagarea constantelor, factorizarea expresiilor comune, detectarea variabilelor nefolosite, etc.)

Aceleași tehnici pot fi aplicate și la probleme de *analiză* de cod (într-un cadru foarte general)

Ideea de bază:

construirea grafului de flux de control al programului  
urmărirea modului în care proprietățile de interes se modifică pe parcursul programului (la traversarea nodurilor / muchiilor grafului)

# Graful de flux de control al programului

engl. control flow graph, CFG

Reprezentare în care:

- nodurile sunt instrucțiuni

- muchiile indică secvențierea instrucțiunilor (inclusiv salturi)

⇒ putem avea: noduri cu:

- un singur succesori (ex. atribuiri),

- mai mulți succesori (instrucțiuni de ramificație)

- mai mulți predecesori (reunirea după ramificație)

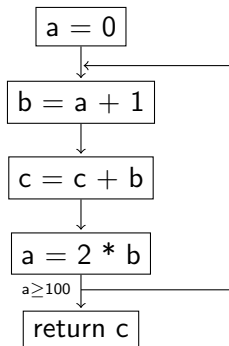
Obs.: Alternativ, dar mai puțin folosit:

- nodurile sunt puncte din program (valori pentru PC)

- muchiile sunt instrucțiuni cu efectele lor

## Exemplu: program și CFG

```
int a = 0, b, c = 0;  
do {  
    b = a + 1;  
    c = c + b;  
    a = 2 * b;  
} while (a < 100);  
return c;
```



## Notății

$G = (N, E)$  : graful de flux de control ( $N$  : noduri;  $E$  : muchii)

$s$  : o instrucțiune de program (nod în graful de flux de control)

$entry, exit$  : punctele de intrare și de ieșire din program

$in(s)$  : mulțimea muchiilor care au  $s$  ca destinație

$out(s)$  : mulțimea muchiilor care au  $s$  ca sursă

$src(e), dest(e)$  : instrucțiunea sursă și destinație a muchiei  $e$

$pred(s)$  : mulțimea predecesorilor instrucțiunii  $s$

$succ(s)$  : mulțimea predecesorilor instrucțiunii  $s$

Scriem *ecuații de flux de date*:

descriu cum se modifică valorile analizate (dataflow facts)  
de la o instrucțiune la alta.

Avem nevoie de valoarea analizată

la intrarea instrucțiunii  $s$  (indice  $in$ )

și la ieșirea instrucțiunii (indice  $out$ )

## Exemplu: Reaching definitions

Care sunt toate *atribuirile* (definițiile) *care pot atinge punctul curent* (înainte ca valorile atribuite să fie suprascrise) ?

Elementele de interes sunt perechi: (variabilă, linie de definiție).

Pentru fiecare instrucțiune (identificată cu eticheta ei  $l$ ) ne interesează valoarea dinainte  $RD_{in}(s)$  și de după  $RD_{out}(s)$

Nodul inițial din graf nu e atins de nici o definiție:

$$RD_{out}(entry) = \{(v, ?) \mid v \in V\}$$

O *atribuire*  $l : v \leftarrow e$

șterge toate definițiile anterioare (pt. variabila  $v$ , nu alte variabile)  
și introduce ca definiție instrucțiunea curentă

$$RD_{out}(l : v \leftarrow e) = (RD_{in}(s) \setminus \{(v, s')\}) \cup \{(v, l)\}$$

Definițiile de la *intrarea* unei instrucțiuni

sunt *reuniunea* definițiilor de la *ieșirea* instrucțiunilor precedente:

$$RD_{in}(s) = \bigcup_{s' \in pred(s)} RD_{out}(s')$$

## Exemplu: Live variables analysis

În fiecare punct de program, ce variabile vor avea valoarea folosită pe cel puțin una din căile posibile din acel punct ?

(analiză utilă în compilatoare pentru alocarea regiștrilor)

Funcția de transfer:  $LV_{in}(s) = (LV_{out}(s) \setminus write(s)) \cup read(s)$

O variabilă e *live* înainte de  $s$

dacă e citită de  $s$

sau *elive* după  $s$  fără a fi scrisă de  $s$   $\Rightarrow$  sensul analizei e *înapoi*

Operația de combinare (meet):

$$LV_{out}(s) = \begin{cases} \emptyset & \text{dacă } succ(s) = \emptyset \\ \bigcup_{s' \in succ(s)} LV_{in}(s') & \text{altfel} \end{cases}$$

$\Rightarrow$  combinarea făcută prin uniune (*may*, pe cel puțin o cale)

Calculul: algoritm de tip *worklist* care face modificări pornind de la valorile inițiale până nu mai apar schimbări  $\Rightarrow$  se atinge un *punct fix*



## Exemplu: Available expressions

În fiecare punct de program, care sunt expresiile a căror valoare a fost calculată anterior, fără să se modificat, pe toate căile spre acel punct? (dacă valoarea se ține minte într-un registru, nu trebuie recalculată)

Funcția de transfer:  $AE_{out}(s) = (AE_{in}(s) \setminus \{e \mid V(e) \cap write(s) \neq \emptyset\}) \cup \{e \in Subexp(s) \mid V(e) \cap write(s) = \emptyset\}$

(expresiile de la intrarea în  $s$  care nu au variabile modificate de  $s$ , și orice expresii calculate în  $s$  fără a li se modifica variabilele)

Operația de combinare (meet):

$$AE_{in}(s) = \begin{cases} \emptyset & \text{dacă } pred(s) = \emptyset \\ \bigcap_{s' \in pred(s)} AE_{out}(s') & \text{altfel} \end{cases}$$

$\Rightarrow$  combinarea e făcută prin intersecție (*must*, pe toate căile);  
analiza e *înainte*

## Exemplu: Very busy expressions

Care sunt expresiile care trebuie evaluate pe orice cale din punctul curent înainte ca valoarea vreunei variabile din ele să se modifice ?

⇒ evaluarea se poate muta în punctul curent, înainte de ramificații  
– o analiză înapoi, și de tip universal (*must*)

$$VBE_{in}(s) = (VBE_{out}(s) \setminus \{e \mid V(e) \cap write(s) \neq \emptyset\}) \cup Subexp(s)$$

$$VBE_{out}(s) = \begin{cases} \emptyset & \text{dacă } succ(s) = \emptyset \\ \bigcap_{s' \in succ(s)} VBE_{in}(s') & \text{altfel} \end{cases}$$

## Proprietăți analizate (dataflow facts)

*Concret*: analizăm diverse proprietăți, de ex.

- valoarea unei variabile într-un punct de program
- sau *intervalul* de valori pentru o variabilă
- sau mulțimi de variabile (live), expresii (available, very busy), definiții posibile pentru o valoare (reaching definitions), etc.

*Abstract*: o mulțime  $D$  de valori pentru o proprietate (*dataflow facts*)

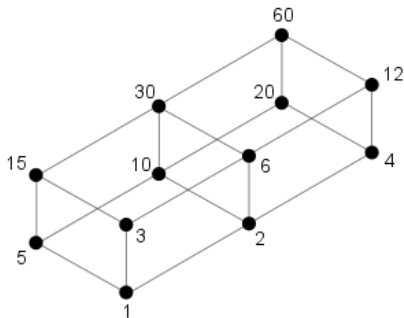
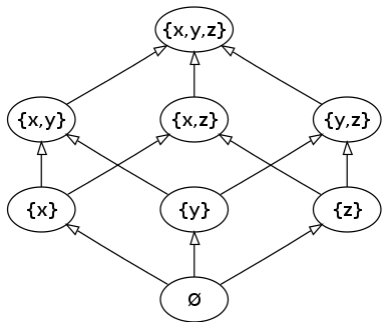
Restricție:  $D$  e o mulțime *finită*

# Lattice

O *latice* e o mulțime *parțial ordonată*, în care orice două elemente au un *minorant* și un *majorant*.

(elemente mai mici, respectiv mai mari în ordine decât cele două). Ex: mulțimea părților unei mulțimi (intersecție, reuniune)

Ex: mulțimea divizorilor unui număr (c.m.m.d.c, c.m.m.m.c)



Imagine: [http://en.wikipedia.org/wiki/File:Hasse\\_diagram\\_of\\_powerset\\_of\\_3.svg](http://en.wikipedia.org/wiki/File:Hasse_diagram_of_powerset_of_3.svg)

[http://en.wikipedia.org/wiki/File:Lattice\\_of\\_the\\_divisibility\\_of\\_60.svg](http://en.wikipedia.org/wiki/File:Lattice_of_the_divisibility_of_60.svg)

## Funcții de transfer

**Concret:** instrucțiunile determină modificări ale stării programului. Valoarea unei variabile după o instrucțiune e o funcție a valorii de la începutul instrucțiunii.

**Abstract:** Fiecare instrucțiune  $s$  are asociată o funcție de transfer  $F(s) : L \rightarrow L$  care determină modul în care valoarea proprietății la începutul instrucțiunii e modificată de instrucțiune:

$Prop_{out}(s) = F(s)(Prop_{in}(s))$  (pentru analize înainte),  
sau invers (pentru analize înapoi)

Restricție: punem condiția ca funcțiile de transfer să fie *monotone*:

$$x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$$

(dacă știm mai multe despre argument, atunci și despre rezultat)

Caz particular: *bitvector frameworks*: laticea e o mulțime de părți  $\mathcal{P}(D)$ , funcții de transfer monotone și de forma:

$$F(s)(v) = (v \setminus kill(s)) \sqcup gen(s)$$

( $v$  = dataflow fact,  $gen/kill(s)$  = informația generată/eliminată în  $s$ )

## Ecuții de flux de date

Exemplu: pentru analize înainte:

$$\begin{aligned} Prop_{out}(s) &= F(s)(Prop_{in}(s)) \\ Prop_{in}(s) &= \prod_{s' \in pred(s)} Prop_{out}(s') \end{aligned}$$

unde prin  $\prod$  am reprezentat efectul combinării informațiilor (*meet*) pe mai multe căi (ar putea fi  $\cap$  sau  $\cup$ )

Inițial, e cunoscută valoarea  $Prop_{out}(entry)$ .

Pentru analize înapoi, se schimbă rolul între *in* și *out*, și e cunoscută valoarea lui  $Prop_{in}(exit)$ .

## Soluția: Algoritm de tip *worklist*

Pentru calculul soluției la sistemul de ecuații de mai sus:  
algoritmi iterativ care propagă modificările în sensul analizei

**foreach**  $s \in N$  **do**  $Prop_{in}(s) = \top$  /\* no info \*/

$Prop_{in}(entry) = init$  /\* in functie de analiza \*/

$W = \{entry\}$

**while**  $W \neq \emptyset$

**choose**  $s \in W$

$W = W \setminus \{s\}$

$Prop_{in}(s) = \prod_{s' \in pred(s)} Prop_{out}(s')$

$Prop_{out}(s) = F(s)(Prop_{in}(s))$

**if** change **then**

**forall**  $s' \in succ(s)$  **do**  $W = W \cup \{s'\}$

## Terminare: condiția de punct fix

Terminarea analizei e garantată dacă funcția de transfer e monotonă:  
 $x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$ , ceea ce implică faptul că proprietățile calculate se modifică în mod monoton.

Def: *Punct fix* pentru o funcție  $f$ : o valoare  $x$  pt. care  $f(x) = x$   
*Teorema lui Tarski* garantează că o funcție monotonă pe o latice completă are un punct fix minimal și un punct fix maximal.

Algoritmul worklist calculează punctul fix minimal dat fiind sistemul de funcții de transfer.



## Meet over all paths

Dorim să calculăm efectul combinat al instrucțiunilor programului:  
pentru  $p = s_1 s_2 \dots s_n$  șir de instrucțiuni definim

$$F(p) = F(s_n) \circ \dots \circ F(s_2) \circ F(s_1)$$

și dorim să calculăm:

$$\prod_{p \in \text{Path}(\text{Prog})} F_p(\text{entry})$$

Dar algoritmul iterativ combină efectele la fiecare punct de întâlnire înainte de a calcula mai departe. Funcțiile  $f$  fiind monotone, avem:

$$f(x \sqcup y) \supseteq f(x) \sqcup f(y)$$

deci analiza *pierde din precizie*

Pentru funcțiile de transfer *distributive* avem chiar:

$$f(x) \cup f(y) = f(x \cup y)$$

Se demonstrează că în acest caz algoritmul iterativ (care generează o soluție de punct fix) e echivalent cu calculul soluției prin combinarea valorilor pe toate căile posibile (*meet over all paths*).

⇒ combinarea diverselor căi de execuție nu pierde informație

Cele 4 exemple date (live variables, etc.) sunt distributive.

## Clasificare a analizelor

- înainte sau înapoi
- must sau may
- dependente sau independente de fluxul de control (flow (in)sensitive):
  - Trebuie luată în considerare ordinea instrucțiunilor în program
    - nu: ce variabile sunt folosite/modificate, funcții apelate, etc.
    - da: proprietăți legate de valorile calculate efective de program
- dependente sau independente de context
  - în cazul programelor cu proceduri: e specializată analiza procedurii în funcție de locul de apel, sau se poate face un sumar (o analiză comună) ?
- dependente sau nu de cale (*path*-(in)sensitive)
  - (ține cont de corelările pe căile individuale de execuție ?)