Analiza programelor. Introducere

27 aprilie 2004

- Ce este analiza programelor
- Aplicaţii practice recente
- Analiza fluxului de date: Introducere

Analiza programelor: ce, de ce, cum ?

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

Scopul:
– pentru a deduce proprietăţi despre comportamentul programelor (in principal corectitudinea, dar şi performanţă, etc.)

Metode:
– prin analiza *statică* a codului sursă (NU executabilul; NU rularea lui)
⇒ metodă diferită de simulare sau testare

Analiză şi verificare

Analiza programelor e legată tot mai mult de verificarea formală.

Verificarea formală: stabileşte că un sistem e corect prin analiza riguroasă a unui model matematic al sistemului
– în general, proprietăţi specifice, detaliate despre comportament (ex. după evenimentul A apare evenimentul B etc.)
– necesită în principiu analiza (simbolică) a secvenţelor de execuţie a modelului (explorarea spaţiului stărilor)

Analiza statică: bazată tot pe tehnici matematice, riguroase
– de regulă pentru proprietăţi mai generale
– folosind aproximaţii sigure (conservatoare)
– de regulă nu explorează spaţiul stărilor programului

Caracteristici şi principii de bază

Analiza programelor: tehnici pentru prezicerea statică, la compilare a mulţimii comportamentelor dinamice (la rulare) ale programului
[Nielson & Nielson]

În general, o analiză precisă e nedecidabilă (v. Church, Gödel, Turing)
⇒ analiza trebuie să facă *aproximaţii*
⇒ dar trebuie să fie *sigură* (să corespundă semanticii programului, şi să nu *omită* situaţii posibile / erori.

Din punct de vedere practic:
– suficient de precisă (cu minim de avertismente false)
– eficientă (spaţiu/timp) pentru a trata programe de dimensiuni realiste

Exemplu: Absenţa erorilor la rulare în Airbus A340

[Patrick Cousot et. al 2003 (École Normale Supérieure)]

Contextul: programe C fără alocare dinamică şi recursivitate
ex. pentru software de sisteme integrate (embedded)
caz specific: software-ul de control de zbor din Airbus A340

Date: 132 000 linii sursă, 2 ani de cercetare şi analiză, 1h20' execuţie

Tipuri de erori la rulare tratate:
– comportament nedefinit conform standardului (împărţire la zero, depăşire de indici de tablou);
– comportament dependent de implementare (ex. depăşire aritmetică),
– nerespectarea aserţiunilor programatorului: `assert()` şi similare

Exemplu: Corelarea valorilor cu căile de execuţie

```
typedef enum {FALSE = 0, TRUE = 1} BOOLEAN;
BOOLEAN B;
void main () {
  unsigned int X, Y;
  while (1) {
    /* ... */
    B = (X == 0);
    /* ... */
    if (!B) {
      Y = 1 / X;
    };
    /* ... */
  };
}     /* sursa: Cousot et al. */
```

E uşor de detectat vizual corelarea între `B` şi `X`
dar într-un program mare, complexitatea analizei creşte exponenţial

Exemplu: calcule de reglare (filtre digitale)

```
BOOLEAN init; float P, X;
void filter () {
  static float E[2], S[2];
  if (INIT) S[0] = P = E[0] = X;
  else P = ((((0.5 * X - E[0] * 0.7)
         + E[1] * 0.4) + S[0] * 1.5) - S[1] * 0.7);
  E[1] = E[0]; E[0] = X; S[1] = S[0]; S[0] = P;
}
void main () {
  X = 0.2 * X + 5;
  INIT = TRUE;
  while (1) {
    X = 0.9 * X + 35;
    filter ();
    INIT = FALSE;
  }
}/* sursa: după Cousot et al. */
```

Problema: demonstrarea absenței depășirilor, și rămânerea lui P într-un interval dat (în acest caz, [-1327.05, 1327.05]
⇒tehnici de analiză pe intervale de valori, cu specific de teoria reglării

Exemplu: Calcule cu numere reale în standardul IEEE

```
float x = 1.0e30, y = x + 1.0e20;
printf("%f\n", y - x);
/* tipăreşte 0.000000 */

double x; float y, r;
/* x = ldexp(1.,50)+ldexp(1.,26); */
x = 1125899973951488.0;
y = x + 1; r = y - x;
printf("%f\n", r);
/* tipăreşte 67108864.000000 */
```

La reprezentarea a numerelor reale pot apărea erori de rotunjire
Analiza trebuie să țină cont de ele, chiar cumulate după ore de rulare!

Exemplu: detecția statică a erorilor

Clase de erori frecvente în programe:
– folosirea variabilelor neinițializate
– dereferențierea de pointeri nuli
– depășirea limitelor de indici în tablou

Aceste erori pot fi detectate prin *analiză statică* a codului sursă
ex. Splint (U. Virginia) sau UNO (Bell Labs) pt. C sau ESC/Java (Compaq SRC)

```
int *p = malloc(100 * sizeof(int));
if (p != NULL) printf("%d", p[100]);
--------------
splint +bounds pointer.c
pointer.c:7:18: Array element p[100] used before definition
pointer.c:7:18: Possible out-of-bounds read: p[100]
```

Problema: analize în același timp *precise* (fără multe alarme false) și *scalabile* la programe de dimensiuni mari

Exemplu: Analiza fluxului de valori/alias analysis

Problema: ce valori poate să ia o anumită variabilă într-un program ?
Mai precis: ce expresii din program pot fi atribuite la o variabilă ?
Se poate calcula un *graf al fluxului de valori* (value flow graph)
și folosi pentru a detecta eventuale atribuiri de valori eronate.

Dacă apar pointeri, valoarea unei variabile se poate modifica indirect
⇒ care sunt toți pointerii care pot indica o anumită variabilă ?
⇒ pot doi pointeri să indice spre același obiect (*alias*) ?

– algoritm precis (ține cont de sensul atribuirii, $x \leftarrow y$ [Andersen '94] cubic, impractic pentru programe mari
– algoritm cu unificare (la fel pt. $x \leftarrow y$ și $y \leftarrow x$) [Steensgaard '96] aproape liniar, relativ imprecis
– algoritm hibrid [Das'00], practic liniar, precizie apropiată de primul

Exemplu: Detecție de erori în sisteme de operare

[Engler et al., Stanford '00] - *meta-compilare*
– descoperit > 500 de erori in cod sistem (Linux, OpenBSD, etc.)
– prin analize statice de tipul celor din compilatoare, combinate cu enunțarea de proprietăți și reguli specifice domeniului de aplicație

Exemple:
– întreruperile sunt reactivate înainte de întoarcerea din funcție:
= fiecare `cli` are o pereche `sti` pe toate căile de ieșire
– verificarea în nucleu a pointerilor din spațiul utilizator:
= toate utilizările se fac doar în funcții care își testează pointerii
– `malloc/free`: test de pointer nul, neutilizare după eliberare
– cu întreruperile dezactivate nu apelează funcții care se pot bloca
– utilizarea semafoarelor se face corect, cu apeluri pereche

Exemplu: Typestate analysis / gcc

[Das, Lerner, Seigle '02 - Microsoft + U. Washington]
– o analiză statică (*property simulation*) scalabilă la $n \cdot 100$ kloc
– exemplu: absența de erori in > 600 apeluri de sistem pentru 15 pointeri de fișiere în codul `gcc` (140 kloc)
– prin analiză hibridă între o analiză standard de flux de data (imprecisă) și analiză dependentă de cale (path-sensitive, prea costisitoare)
– păstrând corelarea dintre starea proprietății analizate (ex. `uninit, open, closed` pentru fișier) și variabilele relevante din program)

```
if (dump)
  f = fopen(dumpFil, "w");
if (p) x = 0; else x = 1;
if (dump) fclose(f);
```

Exemplu: Securizarea prin verificari la rulare

CCured [Necula et. al., UC Berkeley]
Securizarea de cod C prin inserarea unui minim de verificări la rulare

Problema: folosirea variată și nesigură a pointerilor în limbajul C

Soluția: un sistem de tipuri care capturează modul de folosire a fiecărui pointer din program: `SAFE` (doar dereferențiat), `SEQ` (cu indice), `WILD` (cu typecast arbitrar)

Cum: instrumentare la nivel sursă, modificând reprezentarea pointerilor (adresă de bază + tag + lungime validă pt. verificări de indici)
– se deduce pentru fiecare pointer tipul cel mai restrictiv
– se instrumentează cu verificările necesare (nenul, depășire, etc.)

Rezultate: pe cod real (Apache, OpenSSL, OpenSSH, sendmail, bind), cu cca 10%-50% degradare de performanță (cu Purify: 10-100 ori!)

Tehnici de analiză a programelor

– Analiza fluxului de date
principalele tehnici originare din domeniul compilatoarelor
aspecte legate de dualitatea precizie / eficiență

– Analiza bazată pe constrângeri
cadru general pentru reprezentarea prin relații de constrângere între mulțimi, cu proceduri eficiente și generice de soluționare

– Interpretare abstractă
simplifică programul prin definirea unei semantici care consideră doar aspectele relevante pentru proprietatea dorită

– Sisteme de tipuri
definind sistem corespunzător de tipuri, multe proprietăți pot fi convertite la probleme de inferență / verificare a tipurilor

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.)

Ulterior, unificate într-un cadru general care permite aplicarea și la alte probleme de analiză de cod.

Abordarea 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

Reprezentare în care:
– nodurile sunt instrucțiuni
– muchiile indică secvențierea instrucțiunilor (inclusiv salturi)
⇒ putem avea: noduri cu:
  – un singur succesor (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

Notații

$G = (N, E)$: 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

Cu aceste noțiuni scriem *ecuații de flux de date* ce descriu cum se modifică valorile analizate (dataflow facts) de la o instrucțiune la alta.

Notăm cu indicii $_{in}$ și $_{out}$ valoarea analizată la intrarea și respectiv ieșirea din instrucțiunea s.

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 pentru variabila v (dar nu pt. alte variabile) și o introduce pe cea 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, care sunt variabilele ale căror valoare va fi 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 e *live* după s fără a fi scrisă de s) $\Rightarrow$ sensul analizei e *înapoi*

Operația de combinare (meet):

$$LV\_eout(s) = \begin{cases} \emptyset & \text{dacă } succ(s) = \emptyset \\ \cup_{s' \in succ(s)} LV\_ein(s') & \text{altfel} \end{cases}$$

$\Rightarrow$ 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):

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

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

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

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