Analiza fluxului de date

7 decembrie 2004

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 proprietăți despre comportamentul programelor (in principal corectitudinea, dar și performanță, etc.)

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

Analiză și verificare

Analiza programelor e legată tot mai mult de verificarea formală.

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

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

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):

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

$\Rightarrow$ 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):

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

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

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

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

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*

Mulțimi parțial ordonate

Concret:
– am asociat cu punctele de program *mulțimi* de valori pentru proprietatea analizată
– am recalculat iterativ mulțimile respective, prin operații de *reuniune* sau *intersecție*; obținând tot mai multe (sau mai puține) valori

Care sunt premisele esențiale pentru a efectua calculele în acest fel ?

Abstract: O *mulțime parțial ordonată* $(L, \sqsubseteq)$ e o mulțime echipată cu o *relație de ordonare parțială* $\sqsubseteq \subseteq L \times L$, adică o relație:
– reflexivă, $x \sqsubseteq x$ pentru orice $x \in L$
– tranzitivă, $x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$, pentru orice $x, y, z \in L$
– antisimetrică: $x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$, pentru orice $x, y \in L$

Exemplu: mulțimea părților $(\mathcal{P}(D), \subseteq)$ sau $(\mathcal{P}(D), \supseteq)$

Latici

Latice (completă) = o multime parțial ordonată în care orice submulțime finită are un cel mai mic majorant (least upper bound) și cel mai mare minorant (greatest upper bound).

l_0 e majorant al lui $Y \subseteq L$ dacă $\forall l \in Y$ avem $l \sqsubseteq l_0$
l_0 e minorant al lui $Y \subseteq L$ dacă $\forall l \in Y$ avem $l_0 \sqsubseteq l$

Notăm: $\bigsqcup Y$: cel mai mic majorant al mulțimii $Y \subseteq L$
$\bigsqcap Y$: cel mai mare minorant al mulțimii $Y \subseteq L$
și $\bot = \bigsqcup \emptyset = \bigsqcap L \quad \top = \bigsqcap \emptyset = \bigsqcup L$

Definim atunci operațiile
meet : $x \sqcap y = \bigsqcap\{x, y\}$
join : $x \sqcup y = \bigsqcup\{x, y\}$
(în cazul mulțimii părților: intersecție, reuniune)

Latici (cont.)

Operațiile $\sqcap$ (*meet*) și $\sqcup$ (*join*) au proprietățile:
– sunt comutative
– sunt asociative
– $x \sqcap \bot = \bot$ și $x \sqcup \top = \top$, pentru orice x.

Latice *distributivă*: în care operatorii $\sqcap$ și $\sqcup$ sunt reciproc distributivi:
$x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$
$x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$

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 $F(s) : L \to L$ 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)

Ecuații de flux de date

Exemplu: pentru analize înainte:

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

unde prin $\bigsqcap$ 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 $s \in W$
  $W = W \setminus \{s\}$
  $Prop_{in}(s) = \bigsqcap_{s' \in pred(s)} Prop_{out}(s')$
  $Prop_{out}(s) = F(s)(Prop_{in}(s))$
  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 $p = s_1 s_2 \ldots s_n$ șir de instrucțiuni definim

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

și dorim să calculăm:

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

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

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

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 ce conțin proceduri: e specializată analiza fiecărei proceduri în funcție de locul de apel, sau se poate face un sumar (o analiză comună) ?

Analize interprocedurale

Modelarea programelor cu proceduri: în graful de flux de control, un
– muchie de la locul de apel la începutul procedurii
– muchie de la sfârșitul procedurii la instrucțiunea de după apel

În felul acesta, se obțin toate căile, dar și căi nefezabile
⇒ se restrânge analiza asupra căilor *realizabile*, în care apelurile și reîntoarcerile din funcții sunt împerecheate corect