Definiții accesibile

Adrian Manea, 510 SLA

8 decembrie 2019

Cuprins

1 Analiza fluxului de date **2**
1.1 Abstractizări, puncte și căi de execuție 2
1.2 Schema de analiză 3

2 Definițiile accesibile **5**

Index **6**

Bibliografie **6**

1 Analiza fluxului de date

1.1 Abstractizări, puncte și căi de execuție

În general, analiza statică se ocupă cu studiul comportamentului programelor, fără a le rula. Subiectul pe care îl prezentăm, acela al *definițiilor accesibile* (eng. *reaching definitions*) se încadrează în studiul fluxului de date, în lungul unor căi de execuție. Analiza acestor căi de execuție ne permite să aplicăm diverse tehnici de optimizare, precum eliminarea bucăților de cod „mort", care nu este folosit niciodată sau eliminarea necesității definirii unor constante, atunci cînd acestea se propagă prin cod, înlocuindu-le cu exact valoarea lor.

Pentru a realiza o analiză a fluxului de date, trebuie să stabilim cîteva elemente fundamentale de *abstractizare*. Astfel, vom presupune că execuția programului înseamnă traversarea unor căi de execuție, care schimbă *starea programului*. Această stare este definită de valorile tuturor variabilelor din program, inclusiv ale celor din stiva de rulare care nu se află la suprafață. Execuția unei instrucțiuni intermediare transformă o *stare de intrare* într-o *stare de ieșire*, corespunzătoare stărilor dinaintea, respectiv de după punctul în care se află instrucțiunea intermediară studiată.

Atunci cînd analizăm comportamentul unui program, trebuie să luăm în calcul toate secvențele posibile de puncte, alcătuind drumuri, căi de execuție, printr-un graf al fluxului pe care îl parcurge programul în timpul execuției. Desigur, nu vom putea memora efectiv toate căile posibile și toate informațiile asociate acestor căi, mai ales din cauza faptului că numărul căilor posibile poate să fie infinit. În schimb, ne concentrăm pe informațiile relevante analizei pe care o facem, în funcție de scop.

De asemenea, căile posibile de execuție sînt studiate ținînd cont de următoarele fapte:

- În cadrul unui bloc simplu de cod, punctul în care se află programul după o anume instrucțiune coincide cu punctul programului de dinaintea instrucțiunii următoare;
- Dacă există un drum de la un bloc B_1 la un bloc B_2, atunci punctul programului de după ultima instrucțiune din B_1 poate fi urmat de primul punct de dinaintea primei instrucțiuni a blocului B_2.

De aceea, putem defini în general *o cale (de execuție)* de la un punct p_1 la un punct p_n ca fiind un șir $p_1, \dots, p_n$ astfel încît pentru orice $i = 1, 2, \dots n-1$ are loc exact una dintre afirmațiile:

(1) p_i este punctul care precede imediat o instrucțiune, iar p_{i+1} este punctul care urmează imediat aceeași instrucțiune;

(2) p_i este finalul unui anume bloc, iar p_{i+1} este începutul blocului care-i urmează imediat.

Așa cum am spus, însă, căile posibile de execuție pot fi în număr infinit, dar putem alege să păstrăm informațiile relevante și scopul este să sumarizăm starea programului într-un anume punct folosind un număr finit (și, preferabil, cît mai mic) de stări și informații. Analize diferite pot alege să folosească abstracțiuni diferite și, în general, nu există o analiză perfectă pentru a reprezenta o anume stare.

Două exemple preluate din [Aho et al., 2006] ilustrează cum aceleași stări pot fi interpretate diferit în funcție de scopul analizei:

(1) Să presupunem că vrem să facem *debugging* al unui program și vrem să vedem valorile unei variabile la un anumit punct al programului, precum și unde au fost definite aceste valori. De exemplu, într-o stare 5 am aflat că valoarea variabilei a este una dintre $\{1, 243\}$, care a fost obținută în urma uneia dintre definițiile $\{d_1, d_3\}$. Definițiile care *ar putea* să se propage pînă la un anumit punct al programului în lungul unei anume căi se numesc *definiții accesibile* (eng. *reaching definitions*).

(2) Presupunem, în schimb, că sîntem interesați să implementăm eliminarea constantelor (eng. *constant folding*). Dacă folosim o variabilă x accesibilă printr-o singură definiție, iar acea definiție îi atribuie o valoare constantă, atunci putem pur și simplu să înlocuim variabila x direct cu acea constantă. Pe de altă parte, dacă există mai multe definiții accesibile ale lui x dintr-un anume punct al programului, atunci nu putem să înlocuim valoarea lui x.

    Așadar, pentru scopul de a elimina constantele, vrem să vedem care definiții ale lui x fixat sînt *unice* și se propagă pînă la un anumit punct. De aceea, pur și simplu putem împărți analiza variabilelor în *constante* și *neconstante*, folosind o abstractizare binară.

1.2 Schema de analiză

Atunci cînd realizăm o analiză a fluxului de date, asociem fiecărui punct al programului o *valoare de flux de date* care reprezintă o abstractizare a mulțimilor tuturor stărilor posibile ale programului care pot fi observate în acel punct. Această mulțime de valori posibile se numește *domeniul* aplicației pe care o studiem. De exemplu, în cazul definițiilor accesibile, domeniul este mulțimea submulțimilor care conțin definiții din program. O anume valoare de flux de date este o mulțime de definiții și vrem să asociem fiecărui punct al programului mulțimea precisă de definiții care se pot propaga pînă la punctul respectiv.

Cum am precizat deja, alegerea abstracțiunilor depinde de scopul analizei și vom încerca să păstrăm doar informațiile relevante.

Preluînd din nou din [Aho et al., 2006], dacă `s` este o instrucțiune a programului vom nota cu `IN[s]` valorile de flux de date dinaintea instrucțiunii și cu `OUT[s]` valorile de după instrucțiune, pentru orice `s`.

Problema fluxului de date este să găsim o soluție care să satisfacă constrîngerile ce pot exista asupra `IN[s]` și `OUT[s]`, pentru orice `s`.

Constrîngerile pot fi de două tipuri:

- bazate pe semantica instrucțiunilor, numite *funcții de transfer*;
- bazate pe fluxul controlului.

În ce privește funcțiile de transfer, adăugăm că acestea pot ține cont de propagarea înainte sau înapoi a informației. Astfel, dacă notăm f_s funcția de transfer asociată instrucțiunii `s`, putem avea:

$$\texttt{OUT[s]} = f_s(\texttt{IN[s]}) \text{ sau } \texttt{IN[s]} = f_s(\texttt{OUT[s]}).$$

Constrîngerile bazate pe fluxul controlului se extrag din modul în care se propagă informația. De exemplu, în cadrul unui bloc simplu B, care conține instrucțiunile `s(1), s(2),..,s(n)` în această ordine, trebuie să avem:

$$\texttt{IN[s(i+1)]} = \texttt{OUT[s(i)]}, \quad \forall i = 1, 2, \ldots, n-1.$$

Dar între blocuri, fluxul controlului poate fi mai complex. De exemplu, dacă studiem propagarea definițiilor, putem fi interesați de *reuniunea* tuturor blocurilor în care s-au făcut definiții, anterioare punctului curent.

Detaliem în continuare studiul propagării definițiilor, folosind contextul teoretic stabilit mai sus.

2 Definițiile accesibile

Vom fi interesați de a afla în ce punct al programului *este posibil* ca variabila x să fi fost definită atunci cînd parcurgem fiecare punct p al programului. O astfel de informație aparent simplă poate fi foarte utilă, de exemplu, pentru a determina dacă o anume variabilă este, de fapt, constantă și să o înlocuim cu valoarea ei sau dacă o anume variabilă este folosită fără a fi definită. De exemplu, putem introduce o definiție-fantomă (eng. *dummy*) a unei variabile în punctul de la începutul programului și să studiem dacă această definiție se propagă pînă la un punct în care variabila este utilizată. Dacă da, atunci ea este utilizată fără a fi definită, situație care trebuie semnalată.

Ca terminologie, vom spune că o definiție d *se propagă* pînă la un punct p dacă există un drum de execuție de la punctul ce urmează imediat lui d pînă la p, astfel încît d să nu fie *distrusă* pe acest drum. Spunem că o definiție d a unei variabile x este *distrusă* dacă există o altă definiție d' a lui x în drumul pe care îl studiem.

Observaţie 2.1: În analiza drumurilor de propagare a definițiilor, vom ține cont de bucle, astfel că în cadrul unei bucle considerăm că definiția nu este distrusă, iar punctul ce succede definiția va fi cel de ieșire din buclă.

Mai adăugăm că variabilele pot apărea ca parametri în proceduri, tablouri sau referințe indirecte, care sînt, în general, *alias-uri*, astfel că nu putem spune precis dacă o variabilă este sau nu afectată în mod direct de o asemenea instrucțiune. Pentru consecvență și pentru a face o analiză conservatoare, dacă nu este clar dacă o instrucțiune atribuie sau nu o valoare variabilei x, vom presupune că *o poate face*. Dar pentru simplitate, vom elimina din studiu cazurile cu alias-uri și vom folosi doar variabile locale, scalare.

Bibliografie

[Aho et al., 2006] Aho, A. et al. (2006). *Compilers: Principles, Techniques, and Tools*. Pearson.

[Appel şi Ginsburg, 2004] Appel, A. şi Ginsburg, M. (2004). *Modern Compiler Implementation in C*. Cambridge University Press.

[Cooper şi Torczon, 2012] Cooper, K. şi Torczon, L. (2012). *Engineering a Compiler*. Morgan Kaufmann.

[Møller şi Schwartzbach, 2018] Møller, A. şi Schwartzbach, M. (2018). *Static Program Analysis*. online.

[Muchnick, 1997] Muchnick, S. (1997). *Advanced Compiler Design Implementation*. Morgan Kaufmann.

[Nielson et al., 2005] Nielson, F. et al. (2005). *Principles of Program Analysis*. Springer.

[Schinz şi Liu, 2019] Schinz, M. şi Liu, F. (2019). Advanced compiler construction. Notițe de curs online, accesibile la adresa https://cs420.epfl.ch/index.html.