Definiții accesibile

Adrian Manea, 510 SLA

15 decembrie 2019

Cuprins

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)` pînă la `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.

Considerăm exemplul din figura 2.1.

Să studiem definițiile din blocul `B2`. Toate definițiile din blocul `B1` ajung la începutul blocului `B2`. Definiția `d5: j = j - 1` din blocul `B2` ajunge și la începutul blocului `B2`, deoarece nu mai există altă definiție a lui `j` care să se găsească în bucla ce duce înapoi în `B2`. Această definiție, însă, distruge definiția `d2: j = n`, ceea ce o face să nu ajungă la `B3` sau `B4`.

Pe de altă parte, instrucțiunea `d4: i = i + 1` din blocul `B2` nu ajunge la începutul lui `B2`, deoarece variabila `i` este mereu redefinită de `d7: i = u3`.

În fine, definiția `d6: a = u2` ajunge și ea la începutul blocului `B2`.

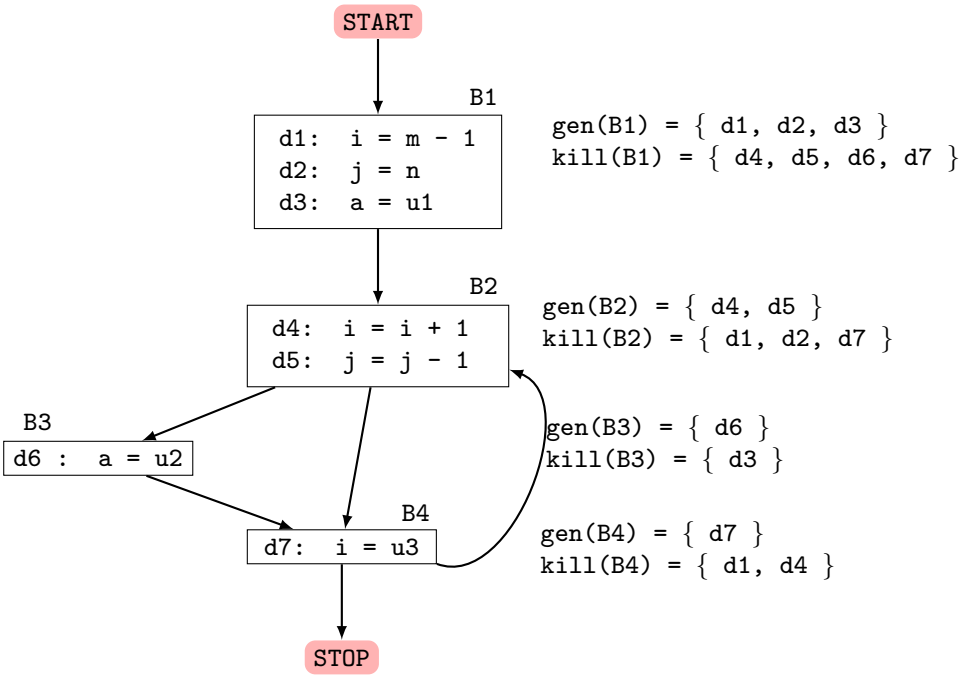

Figura 2.1: Exemplu pentru definiții accesibile

Observaţie 2.2: Prin definirea definițiilor accesibile așa cum am făcut-o, pot exista inadvertențe, însă toate sînt în direcția *conservativă*, adică mai curînd se consideră aspecte redundante, decît să se omită unele. De exemplu, în codul:

```
if (a == b) { instrucțiune1 }
else if (a == b) { instrucțiune2 }
```

`instrucțiune2` nu se va atinge niciodată, pentru nicio valoare a lui `a`.

Însă în general, problema dacă toate drumurile dintr-un graf de flux al datelor sînt parcurse este indecidabilă, astfel că alegem asumpția că ele sînt efectiv parcurse, chiar dacă acest lucru poate conduce la situații precum cea de mai sus. Eroarea în acest caz este de partea pozitivă, astfel că nu pierdem informație, ci cel mult avem o cantitate redundantă.

2.1 Ecuații de transfer

Arătăm acum modul în care se pot formula constrîngerile pentru problema definițiilor accesibile. Începem prin studiul unei singure instrucțiuni:

```
d:  u = v + w.
```

Instrucțiunea generează o definiție `d` a unei variabile `u` și distruge toate celelalte definiții din program care se adresează variabilei `u`, dar nu afectează nicio altă definiție. Așadar, funcția

de transfer pentru definiția `d` poate fi scrisă:

$$f_d(x) = \texttt{gen(d)} \cup (x - \texttt{kill(d)}), \tag{2.1}$$

unde `gen(d)` = {`d`}, este mulțimea definițiilor generate de instrucțiune, iar `kill(d)` este mulțimea celorlalte definiții ale lui `u` din program.

În general funcția de transfer a unui bloc poate fi aflată prin compunerea tuturor funcțiilor de transfer ale instrucțiunilor din interiorul blocului. Se poate vedea ușor că prin compunerea a două funcții precum cea de mai sus (ecuația (2.1), pe care o vom numi ecuație `gen-kill`) se obține tot o funcție cu aceeași formă.

În general, avem:

$$f_b(x) = \texttt{gen(B)} \cup (x - \texttt{kill(B)}), \tag{2.2}$$

unde am notat:

$$\texttt{kill(B)} = \texttt{kill(1)} \cup \texttt{kill(2)} \cup \cdots \cup \texttt{kill(n)},$$

iar pentru generare, avem:

$$\begin{aligned}\texttt{gen(B)} = \texttt{gen(n)} \cup (\texttt{gen(n-1)} - \texttt{kill(n)}) \cup (\texttt{gen(n-2)} - \texttt{kill(n-1)} - \texttt{kill(n)} \cup \\ \cdots \cup (\texttt{gen(1)} - \texttt{kill(2)} - \texttt{kill(3)} - \cdots - \texttt{kill(n)})\end{aligned}$$

Mulțimea `gen` care rezultă dintr-un bloc se va numi mulțimea definițiilor *expuse mai jos*, deoarece această mulțime va conține toate definițiile din bloc care sînt vizibile și imediat după bloc. Rezultă că o definiție dintr-un bloc este expusă mai jos dacă și numai dacă nu este distrusă de o definiție a aceleiași variabile în interiorul aceluiași bloc.

De cealaltă parte, mulțimea `kill` a unui bloc este pur și simplu reuniunea tuturor definițiilor distruse de instrucțiuni individuale.

Remarcăm că, în general, o definiție poate apărea atît în mulțimea `gen`, cît și în `kill` ale unui bloc. Dacă acesta este cazul, atunci considerăm că ea este generată, deoarece presupunem că mulțimea `kill` acționează după mulțimea `gen`.

Exemplu 2.1: Considerăm blocul simplu `B`:

```
d1:  a = 3
d2:  a = 4
```

În acest caz, `gen(B)` = {`d2`}, deoarece `d1` nu este expusă mai jos. De asemenea, tot pentru acest bloc `kill(B)` = {`d1, d2`}, deoarece `d1` distruge `d2` și reciproc. Totuși, din cauza presupunerii de precedență, rezultatul funcției de transfer pentru acest bloc va include mereu `d2`.

2.2 Ecuațiile fluxului de control

Considerăm acum constrîngerile care rezultă din fluxul de control dintre blocurile de bază. Cum o definiție atinge un punct al programului dacă și numai dacă există cel puțin un drum prin care definiția să se propage, avem:

$$\texttt{OUT[P]} \subseteq \texttt{IN[B]},$$

unde P este un punct astfel încît să existe un drum de flux al controlului pînă la blocul B. Însă, cum o definiție nu poate ajunge la un punct decît dacă o face pe un drum, rezultă că IN[B] nu depășește reuniunea definițiilor accesibile din toate blocurile ce-l precedă pe B. Deci putem presupune că, în general, avem:

$$\texttt{IN[B]} = \bigcup_{\texttt{P}\downarrow\texttt{B}} \texttt{OUT[P]},$$

unde am notat cu $\texttt{P} \downarrow \texttt{B}$ faptul că punctul P precede blocul B.

Putem privi această reuniune ca pe un operator *meet* pentru definițiile accesibile, ca în cazul laticelor.

2.3 Algoritm iterativ pentru definiții accesibile

Vom presupune, ca o convenție de organizare, că orice graf de flux al controlului are două blocuri goale: unul care reprezintă nodul de intrare în graf (ENTRY), iar celălalt, de ieșire (EXIT), care reprezintă punctul prin care trec toate căile de ieșire din graf.

Nicio definiție nu poate ajunge la nodul de intrare, deci funcția de transfer pentru acest nod este funcția vidă:

$$\texttt{OUT[ENTRY]} = \emptyset.$$

Pentru toate celelalte blocuri B, în afară de ENTRY, ecuațiile pe care vrem să le rezolvăm sînt:

$$\begin{aligned} \texttt{OUT[B]} &= \texttt{gen(B)} \cup (\texttt{IN[B]} - \texttt{kill(B)}) \\ \texttt{IN[B]} &= \bigcup_{\texttt{P}\downarrow\texttt{B}} \texttt{OUT[P]}. \end{aligned}$$

Algoritmul pe care îl prezentăm rezolvă exact acest sistem de ecuații. Soluția sistemului va fi *cel mai mic punct fix al ecuațiilor*, adică acea soluție care asociază IN și OUT valorile ce se vor regăsi în orice altă soluție. Mai mult, rezultatul algoritmului este acceptabil, deoarece orice definiție pentru IN și OUT va ajunge la punctul descris. Este, totodată, și o soluție dorită, deoarece nu include nicio definiție care sigur nu este accesibilă.

Planul algoritmului este următorul:

- *Date de intrare:* Un graf de flux pentru care s-au calculat `kill(B)` și `gen(B)`, asociate tuturor blocurilor `B`;
- *Date de ieșire:* `IN[B]` și `OUT[B]`, adică mulțimile de definiții care ajung la punctele de intrare, respectiv de ieșire ale fiecărui bloc `B` din graful fluxului;
- *Metoda:* Vom folosi o abordare iterativă, în care începem cu valoarea inițială `OUT[B]` = ∅, pentru orice bloc `B` și vom ajunge prin convergență la valorile căutate pentru `IN` și `OUT`. În principiu, putem folosi o variabilă booleană care să ne arate dacă valorile pentru `IN` și `OUT` s-au schimbat la iterația curentă și să ne oprim atunci cînd nu se mai fac schimbări (i.e. avem convergență), dar vom presupune că mecanismele de modificare sînt subînțelese, deci putem afla și direct dacă s-au făcut sau nu modificări.

  Vom face referire la Algoritmul 1, a cărui desfășurare urmează întocmai elementele de teorie prezentate pînă acum.

Algoritm 1 Algoritm iterativ pentru definiții accesibile

```
1:  procedură RDEF(B)
2:      OUT[ENTRY] = ∅
3:      for (B ≠ ENTRY) do
4:          OUT[B] = ∅
5:      final for
6:      while (OUT se schimbă) do
7:          for (B ≠ ENTRY) do
8:              IN[B] = ⋃_{P↓B} OUT[P]
9:              OUT[B] = gen(B) ∪ (IN[B] − kill(B))
10:         final for
11:     final while
12: final procedură
```

Intuitiv, algoritmul propagă definițiile cît de departe pot ajunge fără a fi distruse și prin aceasta, simulează toate execuțiile posibile ale programului. Remarcăm că algoritmul se va opri, întrucît pentru orice bloc `B`, mulțimea `OUT[B]` nu se poate micșora — atunci cînd este adăugată o definiție, ea rămîne acolo. Deci în final, această mulțime va conține cel mult toate definițiile din program, iar cînd nu se mai adaugă niciuna la `OUT[B]`, programul se oprește. Totodată, această condiție de oprire este una bună, deoarece dacă `OUT[B]` nu s-a schimbat, atunci nu se va schimba nici `IN[B]` la iterația următoare, ceea ce va face să nu se schimbe `OUT[B]` nici la iterația care va urma, deci nu se mai schimbă nimic.

Numărul nodurilor din graful de flux este o margine superioară pentru numărul de repetări ale buclei **while**. Aceasta se poate vedea prin faptul că, dacă o definiție ajunge la un punct, nu o poate face decît pe o cale care nu conține cicluri, iar numărul total de noduri din graf este o

margine superioară pentru nodurile din afara ciclurilor. Totodată, la fiecare repetare a buclei **while**, definițiile se propagă cu cel puțin un nod.

În ciuda aparențelor, algoritmul de mai sus poate fi făcut foarte eficient folosind, de exemplu, *vectori de biți* care să țină evidența definițiilor active. Astfel, se consideră un vector de biți cu lungimea egală cu numărul definițiilor din program, inițializat cu valoarea nulă pe toate pozițiile, iar poziția `i` din vector devine `1` atunci cînd definiția `di` este vizibilă la iterația curentă prin program. Avantajul acestei abordări este că, de îndată ce am construit vectorii de biți, manipularea lor se poate face foarte simplu, folosind operații logice la nivel de biți. Un exemplu analizat este [Aho et al., 2006, Example 9.12, p. 607], alte detalii la [Muchnick, 1997, §8.1], iar un algoritm implementat în C++ este aici.

2.4 Exemplu suplimentar

Preluăm acum un alt exemplu, din [Schinz şi Liu, 2019], pentru a ilustra și cazul variabilelor neinițializate.

Notăm cu $o_i = \{(v_j, d_k)\}$ mulțimea de ieșire corespunzătoare nodului i dintr-un graf de flux al controlului, care este o mulțime de perechi alcătuite din variabile v_j definite la nodul k al grafului. De exemplu, $o_5 = \{(x, 3)\}$ va însemna că la ieșirea din nodul al cincilea al grafului s-a propagat definiția variabilei x, definiție apărută la nodul al treilea. Alternativ, vom folosi și o scriere de forma `o5 = (x,3)`, iar prin abuz de notație, `o5 - x` va însemna, de exemplu, mulțimea definițiilor accesibile la ieșirea din nodul 5, mai puțin toate definițiile lui `x`. Notația `o1 + o2` va însemna reuniunea mulțimilor respective.

Așadar, avem figura 2.2.

Concluzia analizei este că o singură definiție a lui `y` ajunge la nodul final, anume definiția din nodul 2, `y = 3`. Rezultă că pentru optimizare, se poate înlocui în nodul 5 `z = z + 3`, valoarea accesibilă a lui `y`.

Cazul variabilelor neinițializate poate fi tratat cu o analiză similară. Să considerăm acum situația în care se omite definiția `y = 3` în nodul 2, ca în figura 2.3.

Conform analizei, am putea crede că `y` poate fi înlocuită cu valoarea din definiția 5 chiar și în nodul 4, ceea ce este evident fals!

Soluția problemei este să se înceapă analiza cu toate variabilele inițializate. Cele care sînt inițializate efectiv sînt înregistrate în nodul corespunzător, iar cele neinițializate încă sînt înregistrate ca și cum ar fi fost inițializate într-un punct oarecare. Astfel, am obține în exemplul din figura 2.3:

```
o1 = {(x,1), (y,?), (z,?)}
```

și constatăm că „definiția“ `(y,?)` se propagă pînă în `o4`, inclusiv. Așadar, nu se poate înlocui `y` cu `3` mai devreme.

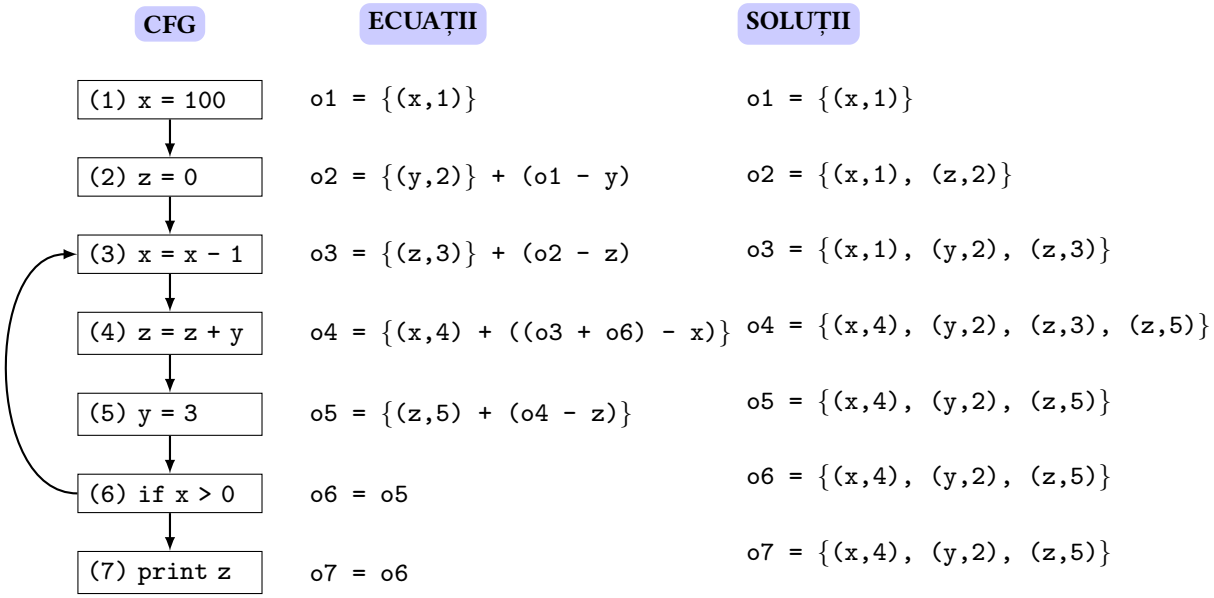

Figura 2.2: Exemplu pentru definiții accesibile

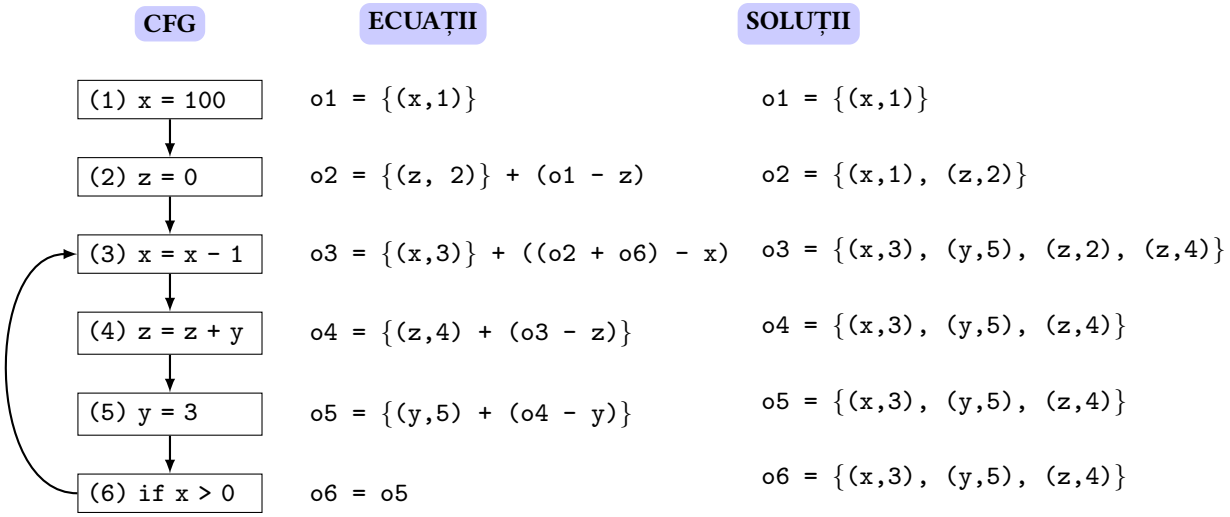

Figura 2.3: Exemplu pentru variabilă neinițializată y

Index

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.