Lecţia 14:

Logica Floyd-Hoare; Extensia la programe interactive

v1.0 (06.05.07)

Gheorghe Stefanescu — Universitatea Bucureşti

Metode de Dezvoltare Software, Sem.2
Februarie 2007— Iunie 2007

Logica Floyd-Hoare

Cuprins:

- *Generalitati*
- Logica Floyd
- Logica Hoare
- Verificări de programe structurate
- Extensia la programe interactive
- Concluzii, diverse, etc.

Verificarea programelor

Verificarea programelor este:

- bazată pe demonstraţii, necesită interacţia om-calculator (nu este complet automatizată), verifică complet comportamentul, şi se foloseşte, în genere, pentru *programe care se termină şi produc un rezultat*

[Reamintim că, prin contrast, *model checking-ul* este o metodă de verificare care este:

- bazată pe modele, automată, verifică proprietăţi, şi se foloseşte, în genere, pentru programe concurente, reactive]

Sunt folosite două versiuni:

- *logica Floyd* (pentru programe schemă-logică arbitrare) şi
- *logica Hoare* (pentru programe “while” - structurate)

Modele Kripke peste logica de ordinul 1

Logică propoziţională vs. *logică predicativă* în stări:

- Model checking-ul (cu CTL) foloseşte modele Kripke peste *logica propoziţională* (i.e., ce avem într-o stare/lume este descris cu o formulă din logica propoziţională).

- O întrebare naturală şi foarte importantă este dacă putem extinde formalismul pentru a folosi *logica de ordinul întâi în stări*.

Variabile in lumi diferite

Problemă tehnică:

- Să zicem că avem: două stări s_0, s_1; o variabilă întreagă x; şi o tranziţie de la s_0 la s_1 care creşte pe x cu 1

$$s_0 \quad \bigcirc \xrightarrow{x := x+1} \bigcirc \quad s_1$$

- Dacă $x = 1$ în s_0, atunci asta implică $x = 2$ în s_1.

- Cum putem scrie? Este clar *greşit* să scriem

$$x = 1 \to x = 2$$

- Soluţie? Cele două părţi $x = 1$ şi $x = 2$ se referă la două lumi diferite, deci avem nevoie de un mecanism care să le conecteze.

..Variabile in lumi diferite

Două soluţii radical diferite:

Soluţia 1:

- Presupunem că avem nişte *variabile rigide (globale)* (ca variabilele statice din programele 00) care pot fi accesate din toate stările/lumile.
- Dacă u este o variabilă rigidă care conţine valoarea 1 şi $s_i(x)$ denotă valoarea lui x în starea s_i, atunci situaţia anterioară se poate descrie prin

$$s_0(x) = u \rightarrow s_1(x) = u + 1$$

Soluţia 2:

- Facem *transformări locale* astfel ca toate componentele formulei să se refere la *aceeaşi lume*.

..Variabile in lumi diferite

Tipuri de sisteme: Relativ la *soluţia 2*, avem 3 tipuri de sisteme:

- *Evoluţie unică:* Dat un sistem şi o stare iniţială, *viitorul şi trecutul sunt unic determinate*.
  (Matematica clasică s-a focalizat pe astfel de sisteme, descrise uzual prin sisteme de ecuaţii diferenţiale.)
- *Evoluţie deterministă:* Dat un sistem şi o stare iniţială, *se poate determina unic viitorul, dar nu trecutul.*
  (Aceste sisteme sunt fundamentale pentru Computer Science. Cartea lui Wolfram "A New Kind of Science" încearcă să le identifice în multe alte domenii.)
- *Evoluţie nedeterministă:* Dat un sistem şi o stare iniţială, *nici viitorul, nici trecutul nu pot fi unic determinate.*
  (Curent, aceste sisteme sunt prea complexe pentru a fi eficient folosite.)

Din viitor către trecut

Concluzie:

- *In sistemele deterministe se poate exprima viitorul în funcţie de prezent $V = f(P)$, deci putem translata relaţii între variabile din viitor $R(V,..)$ în termeni de variabile din prezent $R(f(P),..)$.*
- Pentru exemplul nostru banal, x în s_1 este $x+1$ din s_0, deci, translatând toată formula în s_0 abţinem

$$x = 1 \rightarrow x+1 = 2$$

banal satisfăcută.

Logica Floyd-Hoare

Cuprins:

- Generalitati
- *Logica Floyd*
- Logica Hoare
- Verificări de programe structurate
- Extensia la programe interactive
- Concluzii, diverse, etc.

Metoda Floyd - I: Corectitudine partiala

Input: Condiţia satisfăcută de valorile iniţiale din program.
Output: Condiţia de satisfăcut de valorile de ieşire din program.
Metodă:

1. Tăiem buclele prin *puncte de tăietură* (START şi HALT sunt automat puncte de tăietură).
2. Găsim *aserţiuni inductive* pentru punctele de tăietură
3. Construim *condiţiile de verificare*:
   Pentru fiecare drum între două puncte de tăietură X şi Y, fie:
   — C_1 aserţiunea din X;
   — C_2 aserţiunea din Y (scrisă cu variabilele din X); şi
   — C_3 condiţia de a urma drumul (scrisă cu variabilele din X)

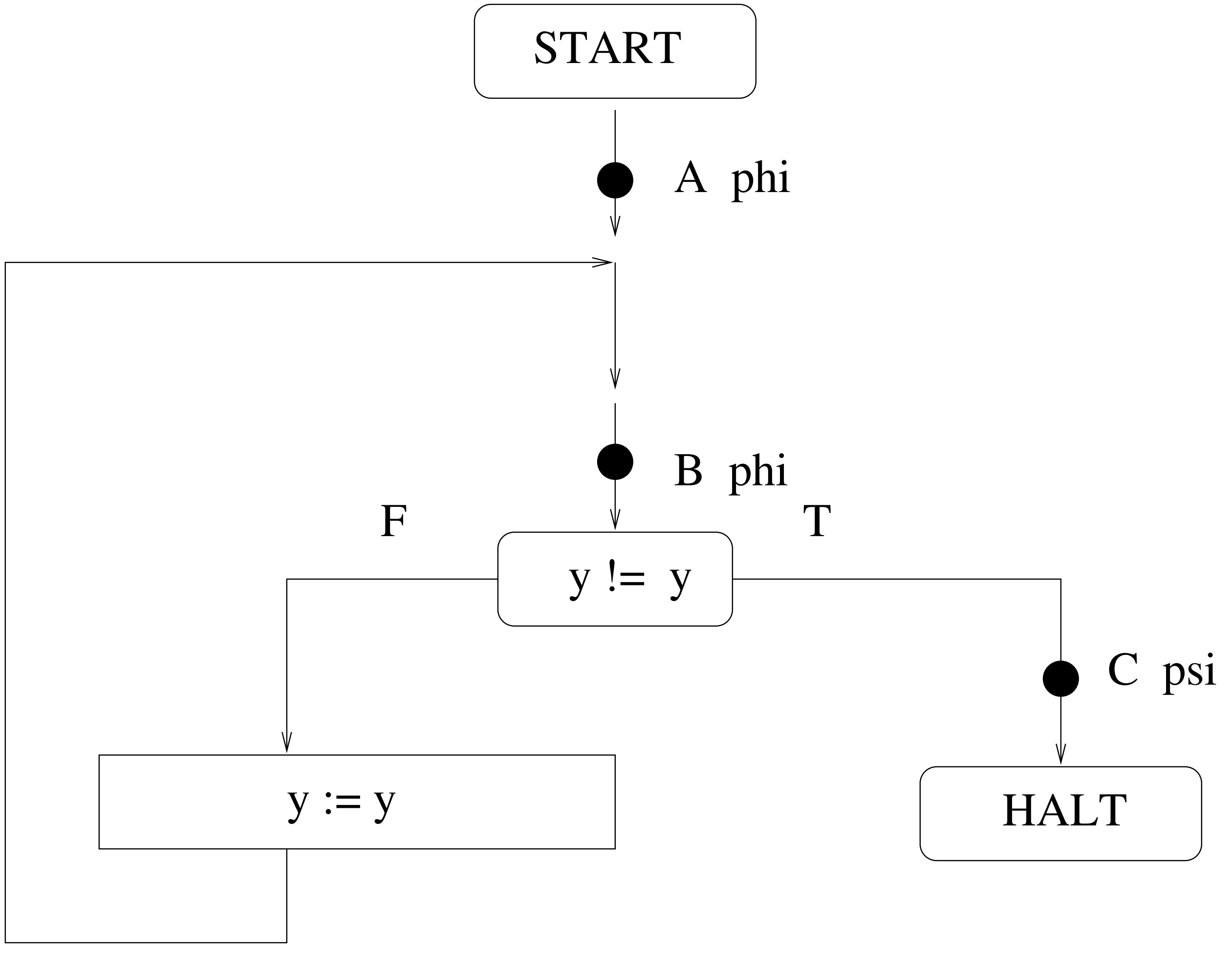
   Condiţia de verificare este:

$$C_1 \wedge C_3 \rightarrow C_2$$

..Floyd method - I: Corectitudine partiala

Teoremă: Dacă toate condiţiile de verificare sunt satisfăcute, atunci programul este parţial corect*, i.e., de fiecare dată când se termină produce rezultatul corect.*

Metoda este utilă doar combinată cu terminarea. Spre exemplu, programul următor este parţial corect pentru orice `phi` şi `psi`.

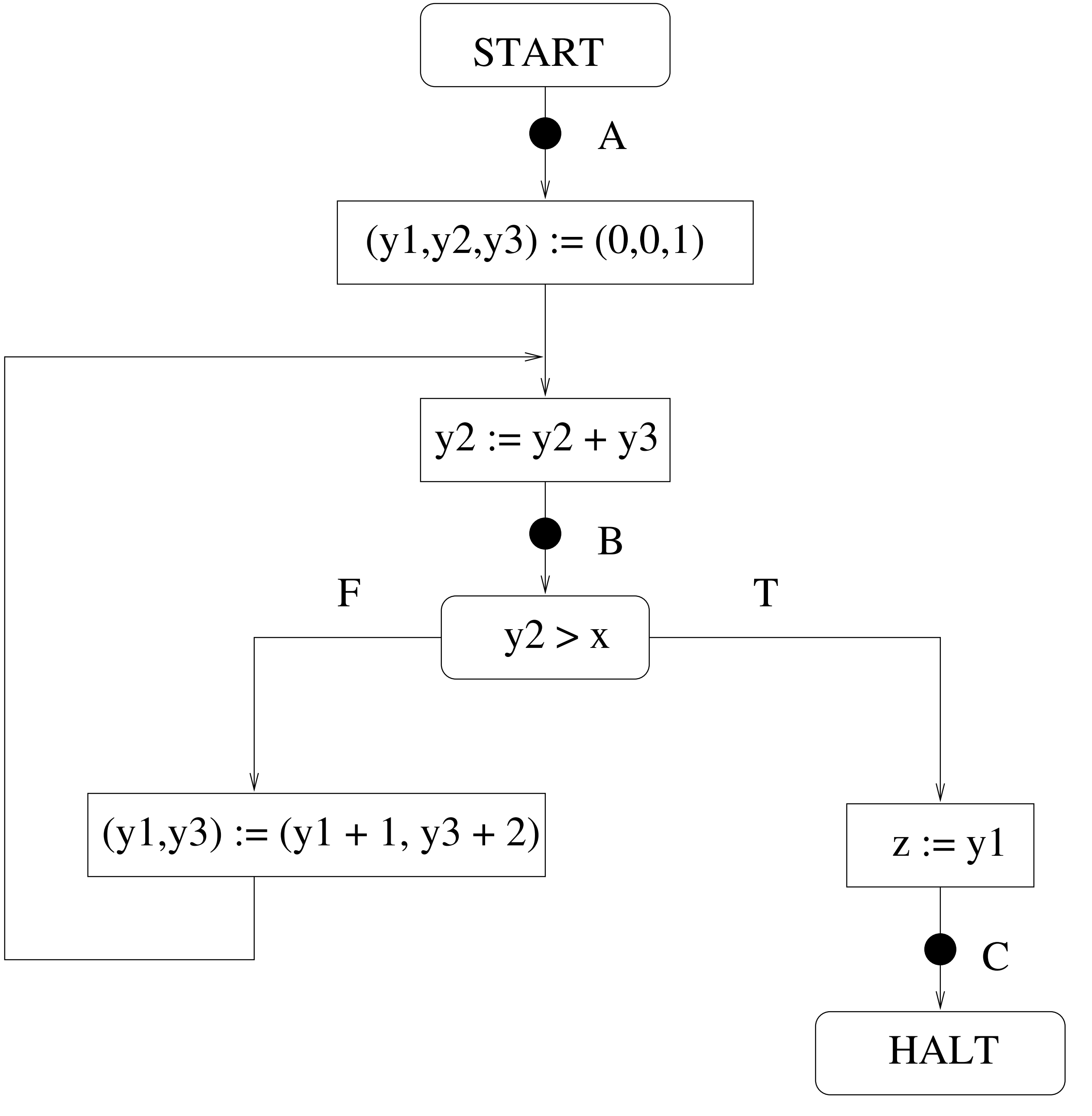


Exemplu

Exemplu:

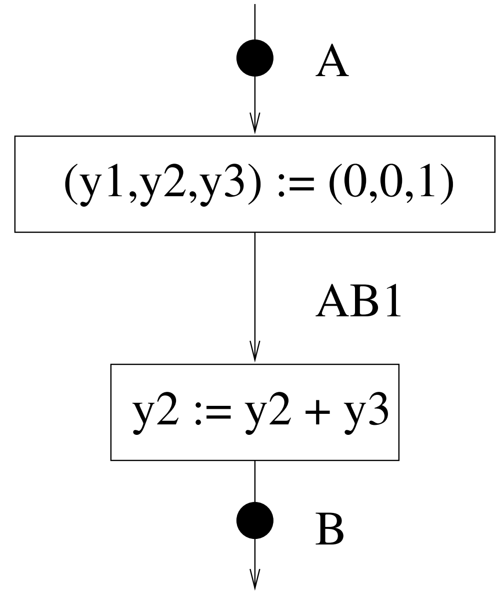
Un program RAD pentru a calcula $z = \lfloor \sqrt{x} \rfloor$

Input (în A):
$x \geq 0$
Output (în C):
$z^2 \leq x < (z+1)^2$



Substitutia inapoi

Backward-substitution: Substituţia înapoi pe drumul de la A la B:



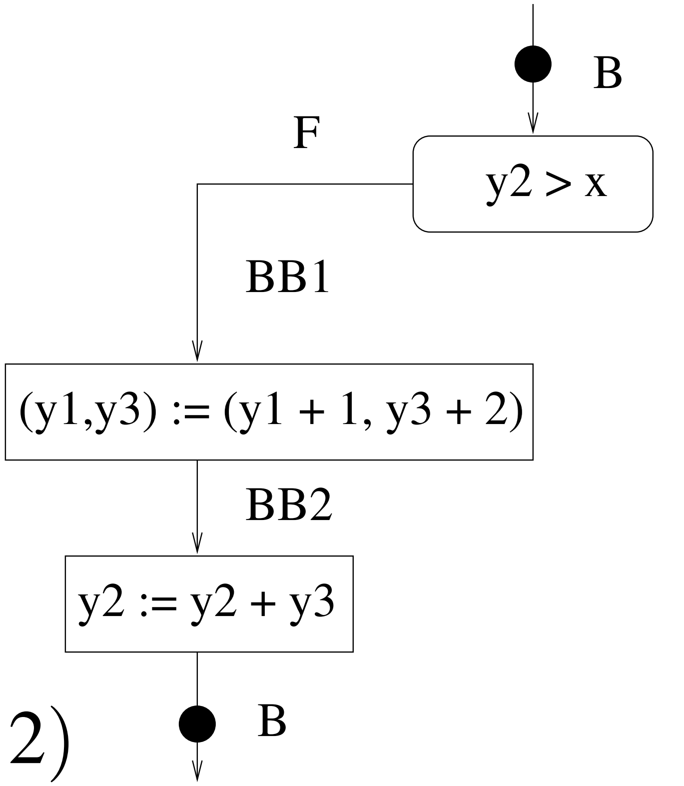
in B: substituţia: (y_1, y_2, y_3)
condiţia de drum: T

in AB1: substituţia: $(y_1, y_2, y_3)[y_2 + y_3/y_2]$
$= (y_1, y_2 + y_3, y_3)$
condiţia de drum: T

in A: substituţia: $(y_1, y_2 + y_3, y_3)[0/y_1, 0/y_2, 1/y_3]$
$= (0, 0 + 1, 1)$
condiţia de drum: T

..Substitutia inapoi

Substituţia înapoi pe drumul de la B la B:



in B: substituţia: (y_1, y_2, y_3)
condiţia de drum: T

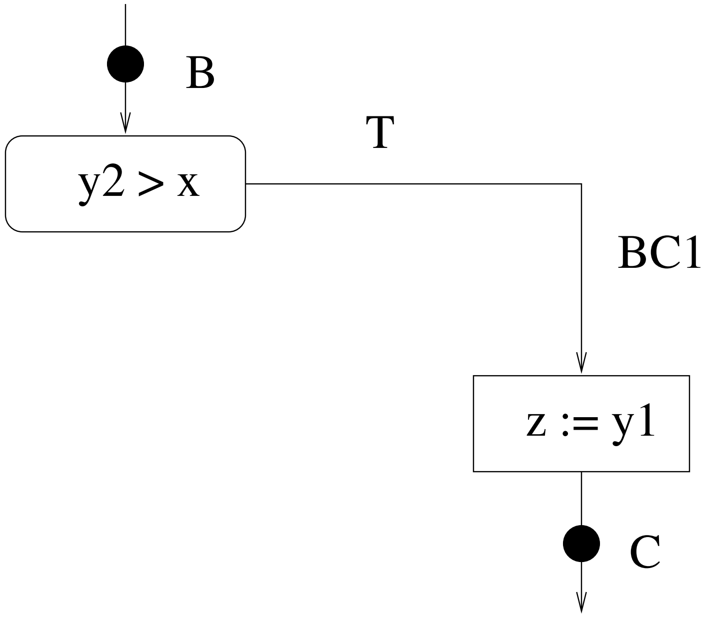
in BB2: substituţia: $(y_1, y_2 + y_3, y_3)$
condiţia de drum: T

in BB1: substituţia: $(y_1 + 1, y_2 + y_3 + 2, y_3 + 2)$
condiţia de drum: T

in B: substituţia: $(y_1 + 1, y_2 + y_3 + 2, y_3 + 2)$
condiţia de drum: $\neg(y_2 > x)$

..Substitutia inapoi

Substituţia înapoi pe drumul de la B la C:



in C: substituţia: z
condiţia de drum: T

in BC1: substituţia: y_1
condiţia de drum: T

in B: substituţia: y_1
condiţia de drum: $y_2 > x$

Conditiile de corectitudine

Conditiile de corectitudine:

- Aserţiunile în A şi C se dau.
- Incercăm (ghicim) aserţiunea în B:

$$y_1^2 \leq x \wedge y_2 = (y_1 + 1)^2 \wedge y_3 = 2y_1 + 1$$

- Acum, putem verifica corectitudinea parţială a acestui program, anume:

aserţiunea într-un punct X
$\wedge$ *condiţia de a urma un drum XY*
$\Rightarrow$ *aserţiunea din Y*

(ultimile două scrise în funcţie de variabilele din X)

..Conditiile de corectitudine

Aserţiunea în B: $y_1^2 \leq x \wedge y_2 = (y_1+1)^2 \wedge y_3 = 2y_1+1$

- Drumul A-AB1-B (cu $(0,1,1)$ şi T):

$$x \geq 0 \to [0^2 \leq x \wedge 1 = (0+1)^2 \wedge 1 = 2\cdot 0+1]$$

- Drumul B-BB1-BB2-B (cu $(y_1+1, y_2+y_3+2, y_3+2)$ şi $\neg(y_2 > x)$):

$$[y_1^2 \leq x \wedge y_2 = (y_1+1)^2 \wedge y_3 = 2y_1+1 \wedge \neg(y_2 > x)]$$
$$\to [(y_1+1)^2 \leq x \wedge y_2+y_3+2 = ((y_1+1)+1)^2$$
$$\wedge\ y_3+2 = 2(y_1+1)+1]$$

- Drumul B-BC1-C (cu y_1 şi $y_2 > x$):

$$[y_1^2 \leq x \wedge y_2 = (y_1+1)^2 \wedge y_3 = 2y_1+1 \wedge y_2 > x]$$
$$\to y_1^2 \leq x < (y_1+1)^2$$

Aceste condiţii sunt formule din logica cu predicate şi se verifică uşor că sunt adevărate, deci *programul RAD este parţial corect*.

Metoda Floyd - II: Terminare

Metoda Floyd - Terminare:

1. Tăiem buclele (ca mai sus) pentru a găsi aserţiuni inductive bune.
2. Alegem o mulţime *bine-formată*, anume o mulţime parţial ordonată care nu are lanţuri descrescătoare infinite.
3. Alegem o funcţie bună (i.e., bine definită, daca folosim aserţiunile de mai sus) ce foloseşte variabilele din program.
4. Verificăm condiţia de terminare, anume în orice buclă *funcţia descreşte strict*.

Teoremă: Dacă condiţiie de terminare sunt satisfăcute, programul se termină.

Exemplu - terminare)

In exemplul nostru,

1. Luăm punctele de tăietură A şi B; noile aserţiuni sunt:

   in A: $x \geq 0$;

   in B: $y_2 \leq x \wedge y_3 > 0$

2. Luăm numerele naturale cu ordinea "$<$" şi funcţia:

   in B: $x - y_2$

3. Condiţia de terminare este: pe drumul de la B la B

   "aserţiunea din B şi condiţia de drum implică faptul că funcţia în B este mai mare decât funcţia în B calculată pentru noile valori ale variabilelor" anume

$$[y_2 \leq x \wedge y_3 > 0 \wedge \neg(y_2 > x)] \\ \rightarrow [x - y_2 > x - (y_2 + y_3 + 2)]$$

Deci *programul RAD se termină*.

Logica Floyd-Hoare

Cuprins:

- Generalitati
- Logica Floyd
- *Logica Hoare*
- Verificări de programe structurate
- Extensia la programe interactive
- Concluzii, diverse, etc.

Logica Hoare

Logica Hoare:

- *Logica Hoare* este cazul particular luat de logica Floyd aplicată *programelor strucutrate - "while"*.

- Avantajul original al logicii Hoare a fost că putem face verificarea *compoziţional*, modular. (Curent, există un formalism composiţional şi pentru logica Floyd.)

- Totuşi, majoritatea programelor uzuale sunt programe "while", aşa că logica Hoare este acum mult *mai populară* decât logica Floyd.

Un nucleu de limbaj de programare

Programe while (un nucleu de limbaj de programare ca în Pascal, C, C++, ori Java): Definite prin sintaxa:

- **Expresii întregi:** (n - număr înterg)

$$E ::= \mathtt{n} \mid x \mid (-E) \mid (E+E) \mid (E-E) \mid (E*E)$$

- **Expresii booleane:**

$$B ::= \mathtt{true} \mid \mathtt{false} \mid (!B) \mid (B\&B) \mid (B\|B) \mid (E<E)$$

- **Instrucţiuni:**

$$C ::= \mathit{null} \mid x := E \mid C;C \mid \mathtt{if}\ B\ \{C\}\ \mathtt{else}\ \{C\} \mid \mathtt{while}\ B\ \{C\}$$

Programul factorial

Fac1

Programul `Fac1` calculează funcţia factorial

```
y := 1;
z := 0;
while (z != x) {
  z := z + 1;
  y := y * z;
}
```

anume, ieşirea este $\mathtt{x}! = 1 \cdot 2 \cdot \ldots \cdot \mathtt{x}$.

Triplete Hoare

- Un *triplet Hoare* este o notaţie

$$(\!|\, \phi \,|\!)\ P\ (\!|\, \psi \,|\!)$$

- Fromula ϕ se numeşte *precondiţie*, iar ψ *postcondiţie* a lui P.
- Intuitiv, acestă notaţie are semnificaţia:
  - *Dacă programul P se rulează într-o stare ce satisface ϕ, atunci după execuţie se ajunge într-o stare ce satisface ψ.*
- De fapt, există două semantici diferite $\models_{par}$ şi $\models_{tot}$ una pentru corectitudinea parţială, cealaltă pentru corectitudinea totală: la ultima, se garantează că programul se termină, pe când la prima, nu.

Semantica tripletelor Hoare

Semantica tripletelor Hoare:

- Tripletul Hoare $(\!|\,\phi\,|\!)\ P\ (\!|\,\psi\,|\!)$ este satisfăcut relativ la *corectitudinea parţială*

$$\models_{par} (\!|\,\phi\,|\!)\ P\ (\!|\,\psi\,|\!)$$

  dacă pentru orice stare care satisface ϕ, starea rezultată după execuţia lui P satisface ψ, cu condiţia ca programul să se termine.

- Tripletul Hoare $(\!|\,\phi\,|\!)\ P\ (\!|\,\psi\,|\!)$ este satisfăcut relativ la *corectitudinea totală*

$$\models_{tot} (\!|\,\phi\,|\!)\ P\ (\!|\,\psi\,|\!)$$

  dacă pentru orice stare care satisface ϕ, programul P garantat se termină şi starea rezultată după execuţia lui P satisface ψ.

Reguli de corectitudine partiala

Reguli:

$$\frac{}{(\!|\, \psi[E/x] \,|\!)\ x := E\ (\!|\, \psi \,|\!)}\,\texttt{Assignment}$$

$$\frac{(\!|\, \phi \,|\!)\ C_1\ (\!|\, \eta \,|\!) \quad (\!|\, \eta \,|\!)\ C_2\ (\!|\, \psi \,|\!)}{(\!|\, \phi \,|\!)\ C_1;C_2\ (\!|\, \psi \,|\!)}\,\texttt{Composition}$$

$$\frac{(\!|\, \phi \wedge B \,|\!)\ C_1\ (\!|\, \psi \,|\!) \quad (\!|\, \phi \wedge \neg B \,|\!)\ C_2\ (\!|\, \psi \,|\!)}{(\!|\, \phi \,|\!)\ \texttt{if}\ B\ \{C_1\}\ \texttt{else}\ \{C_2\}\ (\!|\, \psi \,|\!)}\,\texttt{If}-\texttt{statement}$$

$$\frac{(\!|\, \eta \wedge B \,|\!)\ C\ (\!|\, \eta \,|\!)}{(\!|\, \eta \,|\!)\ \texttt{while}\ B\ \{C\}\ (\!|\, \eta \wedge \neg B \,|\!)}\,\texttt{Partial}-\texttt{while}$$

$$\frac{\vdash \phi' \to \phi \quad (\!|\, \phi \,|\!)\ C\ (\!|\, \psi \,|\!) \quad \vdash \psi \to \psi'}{(\!|\, \phi' \,|\!)\ C\ (\!|\, \psi' \,|\!)}\,\texttt{Implied}$$

Regula `Assignment`

Assignment:

$$\frac{}{(\!|\ \psi[E/x]\ |\!)\ x := E\ (\!|\ \psi\ |\!)}\texttt{Assignment}$$

- Este axiomă, deci poate fi folosită fără premize.
- Scopul este de a demonstra ψ în starea de după asignarea $x := E$.
- Translatată în variabilele de dinainte de asignare, condiţia devine $\psi[E/x]$.
- Deci, luăm *cea mai slabă precondiţie* care garantează că după instrucţiune condiţia este satisfăcută.

..Regula `Assignment`

Exemple: Fie *P* programul

```
x := 2
```

Instanţe ale regulii `Assignment`

- $(\!|\, 2 = 2 \,|\!)\ \mathtt{x := 2}\ (\!|\, x = 2 \,|\!)$
- $(\!|\, 2 = 4 \,|\!)\ \mathtt{x := 2}\ (\!|\, x = 4 \,|\!)$
- $(\!|\, 2 = y \,|\!)\ \mathtt{x := 2}\ (\!|\, x = y \,|\!)$
- $(\!|\, 2 > 0 \,|\!)\ \mathtt{x := 2}\ (\!|\, x > 0 \,|\!)$

Composition:

$$\frac{(\!|\,\phi\,|\!)\ C_1\ (\!|\,\eta\,|\!) \qquad (\!|\,\eta\,|\!)\ C_2\ (\!|\,\psi\,|\!)}{(\!|\,\phi\,|\!)\ C_1;C_2\ (\!|\,\psi\,|\!)}\texttt{Composition}$$

- Scopul este de a demonstra $(\!|\,\phi\,|\!)\ C_1;C_2\ (\!|\,\psi\,|\!)$
- In acest sens, *ghicim* o *proprietate interpolantă* η
- Verificăm corectitudinea parţială a ambelor părţi, i.e., $(\!|\,\phi\,|\!)\ C_1\ (\!|\,\eta\,|\!)$ şi $(\!|\,\eta\,|\!)\ C_2\ (\!|\,\psi\,|\!)$
- Cum vedem, regula de compunere `Composition` (şi mai târziu `Partial-while`) ilustrează *partea creativă*, ne-automată din metodă, asemănătoare cu ghicirea unor aserţiuni inductive bune în logica Floyd.

..Regula `Composition`

Exemplu: Fie P programul

```
x := 2; x := x + 1;
```

Presupunem că trebuie demonstrat

$$(\!|\, 2+1=3 \,|\!)\ \texttt{x := 2; x := x + 1}\ (\!|\, x=3 \,|\!)$$

Luăm aserţiunea interpolantă $x+1=3$. Atunci,

1. $(\!|\, x+1=3 \,|\!)\ \texttt{x := x + 1}\ (\!|\, x=3 \,|\!)$ (ass)
2. $(\!|\, 2+1=3 \,|\!)\ \texttt{x := 2}\ (\!|\, x+1=3 \,|\!)$ (ass)
3. $(\!|\, 2+1=3 \,|\!)\ \texttt{x := 2; x := x + 1}\ (\!|\, x=3 \,|\!)$ (comp. 1,2)

Notă:

- Aici este o demonstraţie ad-hoc. Convenţia de scriere a demonstraţiilor este destul de diferită (vezi slide-urile ce urmează).

Regula `If-statement`

If-statement:

$$\frac{(\!|\, \phi \wedge B \,|\!)\; C_1 \;(\!|\, \psi \,|\!) \qquad (\!|\, \phi \wedge \neg B \,|\!)\; C_2 \;(\!|\, \psi \,|\!)}{(\!|\, \phi \,|\!)\; \mathtt{if}\; B\; \{C_1\}\; \mathtt{else}\; \{C_2\}\; (\!|\, \psi \,|\!)}\,\mathtt{If-statement}$$

- Spre a ne atinge ţinta, dividem testul în două părţi, câte una pentru fiecare ramură a if-ului.
- Condiţia pentru a urma o ramură este inclusă în precondiţia utilizată pentru acea ramură.

Regula `Partial-while`

Partial-while:

$$\frac{(\!|\, \eta \wedge B \,|\!)\ C\ (\!|\, \eta \,|\!)}{(\!|\, \eta \,|\!)\ \mathtt{while}\ B\ \{C\}\ (\!|\, \eta \wedge \neg B \,|\!)}\ \mathtt{Partial-while}$$

- Ghicim o condiţie invariantă η.
- Demonstrăm că η este într-adevăr un invariant, anume η şi condiţia pentru a accesa corpul lui `while` implică η.
- Regula `Partial-while` ne asigură că η este adevărat după instrucţiunea. In plus, aserţiunea finală include şi informaţia că, după `while`, condiţia de acces în corpul lui `while` este falsă.

Regula `Implied`

Implied:

$$\frac{\vdash \phi' \to \phi \quad (\!|\, \phi \,|\!)\; C \;(\!|\, \psi \,|\!) \quad \vdash \psi \to \psi'}{(\!|\, \phi' \,|\!)\; C \;(\!|\, \psi' \,|\!)}\;\texttt{Implied}$$

- Regula este evidentă: Dacă se pleacă de la o triplet Hoare valid şi se *întăreşte precondiţia* şi se *slăbeşte postcondiţia*, tripletul rămâne valid.

Corectitudine & Completitudine

Dacă un triplet $(\!|\,\phi\,|\!)\; P\; (\!|\,\psi\,|\!)$ admite o demonstraţie în acest sistem logic scriem

$$\vdash_{par} (\!|\,\phi\,|\!)\; P\; (\!|\,\psi\,|\!)$$

- *Corectitudine* (uşor; se verifică fiecare regulă separat)

$$\vdash_{par} (\!|\,\phi\,|\!)\; P\; (\!|\,\psi\,|\!) \text{ implică } \models_{par} (\!|\,\phi\,|\!)\; P\; (\!|\,\psi\,|\!)$$

- *Completitudine* (dificil, nu o dăm aici)

$$\models_{par} (\!|\,\phi\,|\!)\; P\; (\!|\,\psi\,|\!) \text{ implic1 } \vdash_{par} (\!|\,\phi\,|\!)\; P\; (\!|\,\psi\,|\!)$$

Notă: Completitudinea nu este complet pură*, de regulă folosindu-se condiţii tehnice care asigură existenţa invarianţilor.*

Logica Floyd-Hoare

Cuprins:

- Generalitati
- Logica Floyd
- Logica Hoare
- *Verificări de programe structurate*
- Extensia la programe interactive
- Concluzii, diverse, etc.

O demonstratie

Demonstratii: Ca în orice logică, dată cu astfel de reguli, se pot specifica demonstraţii folosind *arbori de demonstraţii* (proof trees). Un arbore de demonstraţie pentru programul `Fac1` este:

$$
\dfrac{
\dfrac{
\dfrac{\llparenthesis 1=1 \rrparenthesis\ \texttt{y:=1}\ \llparenthesis y=1 \rrparenthesis}{\llparenthesis \top \rrparenthesis\ \texttt{y:=1}\ \llparenthesis y=1 \rrparenthesis}
\qquad
\dfrac{\llparenthesis y=1\wedge 0=0 \rrparenthesis\ \texttt{z:=0}\ \llparenthesis y=1\wedge z=0 \rrparenthesis}{\llparenthesis y=1 \rrparenthesis\ \texttt{z:=0}\ \llparenthesis y=1\wedge z=0 \rrparenthesis}
}{\llparenthesis \top \rrparenthesis\ \texttt{y:=1; z:=0}\ \llparenthesis y=1\wedge z=0 \rrparenthesis}
\qquad
\dfrac{
\dfrac{
\dfrac{
\dfrac{\llparenthesis y\cdot(z+1)=(z+1)! \rrparenthesis\ \texttt{z := z+1}\ \llparenthesis y\cdot z=z! \rrparenthesis}{\llparenthesis y=z!\wedge z\neq x \rrparenthesis\ \texttt{z := z+1}\ \llparenthesis y\cdot z=z! \rrparenthesis}
\qquad
\llparenthesis y\cdot z=z! \rrparenthesis\ \texttt{y := y*z}\ \llparenthesis y=z! \rrparenthesis
}{\llparenthesis y=z!\wedge z\neq x \rrparenthesis\ \texttt{z := z+1; y := y*z}\ \llparenthesis y=z! \rrparenthesis}
}{\llparenthesis y=z! \rrparenthesis\ \texttt{while(z != x) \{z := z+1; y := y*z\}}\ \llparenthesis y=z!\wedge z=x \rrparenthesis}
}{\llparenthesis y=1\wedge z=0 \rrparenthesis\ \texttt{while(z != x) \{z := z+1; y := y*z\}}\ \llparenthesis y=x! \rrparenthesis}
}{\llparenthesis \top \rrparenthesis\ \texttt{y := 1; z := 0; while(z != x) \{z := z+1; y := y*z\}}\ \llparenthesis y=x! \rrparenthesis}
$$

..O demonstratie

..Cum se vede greu, repetăm figura cu fonturi mai mari - ambele "..." sunt pe aceeaşi linie):

$$\dfrac{\dfrac{\dfrac{\dfrac{(\!|\, y\cdot(z+1)=(z+1)!\,|\!)\ \texttt{z := z+1}\ (\!|\, y\cdot z=z!\,|\!)}{(\!|\, y=z!\wedge z\neq x\,|\!)\ \texttt{z := z+1}\ (\!|\, y\cdot z=z!\,|\!)}\qquad (\!|\, y\cdot z=z!\,|\!)\ \texttt{y := y*z}\ (\!|\, y=z!\,|\!)}{(\!|\, y=z!\wedge z\neq x\,|\!)\ \texttt{z := z+1; y := y*z}\ (\!|\, y=z!\,|\!)}}{(\!|\, y=z!\,|\!)\ \texttt{while(z != x) \{z := z+1; y := y*z\}}\ (\!|\, y=z!\wedge z=x\,|\!)}}{\ldots\quad (\!|\, y=1\wedge z=0\,|\!)\ \texttt{while(z != x) \{z := z+1; y := y*z\}}\ (\!|\, y=x!\,|\!)}$$

$$\dfrac{\dfrac{\dfrac{(\!|\, 1=1\,|\!)\ \texttt{y:=1}\ (\!|\, y=1\,|\!)}{(\!|\, \top\,|\!)\ \texttt{y:=1}\ (\!|\, y=1\,|\!)}\qquad \dfrac{(\!|\, y=1\wedge 0=0\,|\!)\ \texttt{z:=0}\ (\!|\, y=1\wedge z=0\,|\!)}{(\!|\, y=1\,|\!)\ \texttt{z:=0}\ (\!|\, y=1\wedge z=0\,|\!)}}{(\!|\, \top\,|\!)\ \texttt{y:=1; z:=0}\ (\!|\, y=1\wedge z=0\,|\!)}\quad \ldots}{(\!|\, \top\,|\!)\ \texttt{y := 1; z := 0; while(z != x) \{z := z+1; y := y*z\}}\ (\!|\, y=x!\,|\!)}$$

Tablouri de demonstratii

Tablouri de demonstratii:

- Putem folosi arbori de demonstraţii, ca mai sus, pentru logica Hoare..
- .. dar nu sunt potriviţi decât pentru programe minuscule (ca `Fac1` de mai sus).
- O soluţie este de a utiliza *tablouri de demonstraţii*, i.e., o *mixtură de cod de program şi aserţiuni logice*.

..Tablouri de demonstratii

Exemplu (tablou de demonstraţie):

$$
\begin{array}{ll}
 & (\!|\, \phi_0 \,|\!) \\
C_1; & \\
 & (\!|\, \phi_1 \,|\!) \quad \texttt{justification 1} \\
C_2; & \\
\vdots & \\
C_{n-1}; & \\
 & (\!|\, \phi_{n-1} \,|\!) \quad \texttt{justification n-1} \\
C_n; & \\
 & (\!|\, \phi_n \,|\!) \quad \texttt{justification n}
\end{array}
$$

unde în fiecare pas

$$
\begin{array}{ll}
 & (\!|\, \phi_{i-1} \,|\!) \\
C_i & \\
 & (\!|\, \phi_i \,|\!)
\end{array}
$$

se foloseşte o regulă din logica Hoare.

Repetitii

Intrebarea naturală este dacă alternanţa

aserţiune $\rightarrow$ instrucţiune $\rightarrow$ aserţiune $\rightarrow$ instrucţiune ...

este ori nu *strictă*. Răspunsul este:

- *se permite repetarea aserţiunilor* una după alta, cu condiţia ca ce urmează să fie *implicat* de ce era înainte (aici, putem folosi o logica de ordinul 1 extinsă cu reguli de aritmetică, etc.)
- dar nu *se permite repetarea instrucţiunilor* fără aserţiuni înter ele

Notă: Motivul asimetriei este că ne focalizăm pe verificarea programelor. O demonstraţie complet formală ar trebui să includă şi demonstraţii pentru implicaţiile dintre aserţiuni.

Assignment:

Cum am văzut în logica Floyd, programele se parsează dinspre viitor către trecut, mutând aserţiunile înapoi folosind tehnica de *cea mai slabă precondiţie*.

Exemplu. $\vdash (\!|\ \top\ |\!)$ `z := x; z := z+y; u := z` $(\!|\ u = x + y\ |\!)$

$(\!|\ \top\ |\!)$
$(\!|\ x + y = x + y\ |\!)$ `Implied (4)`
`z := x`
$(\!|\ z + y = x + y\ |\!)$ `Assignment (3)`
`z := z+y`
$(\!|\ z = x + y\ |\!)$ `Assignment (2)`
`u := z`
$(\!|\ u = x + y\ |\!)$ `Assignment (1)`

Instructiunea `if`

If: Metoda pentru `if` este similară, anume *împingem aserţiunile înapoi pe fiecare ramură şi colectăm rezultatele deasupra* `if`*-ului*.

Exemplu: pentru

$$(\!|\, \phi \,|\!)\ \texttt{if (B) \{C1\} else \{C2\}}\ (\!|\, \psi \,|\!)$$

avem

- împingem ψ în sus prin `C1` obţinând un rezultat ϕ_1
- împingem ψ în sus prin `C2` obţinând un rezultat ϕ_2
- luăm pentru ϕ formula $(B \rightarrow \phi_1) \wedge (\neg B \rightarrow \phi_2)$

Notă: Mai exact, avem aici o regulă echivalentă pentru `if` *anume:*

$$\frac{(\!|\, \phi_1 \,|\!)\ C_1\ (\!|\, \psi \,|\!) \quad (\!|\, \phi_2 \,|\!)\ C_2\ (\!|\, \psi \,|\!)}{(\!|\, (B \rightarrow \phi_1) \wedge (\neg B \rightarrow \phi_2) \,|\!)\ \texttt{if}\ (B)\ \{C_1\}\ \texttt{else}\ \{C_2\}\ (\!|\, \psi \,|\!)}$$

..Instructiunea `if`

Exemplu:

$\vdash (\!|\top|\!)$ `a := x+1; if (a-1 == 0)` $\{$`y := 1;`$\}$ `else` $\{$ `y := a;`$\}$ $(\!| y = x+1 |\!)$

Dem.

$(\!|\top|\!)$

$(\!| (x+1-1=0 \to 1 = x+1) \wedge (\neg(x+1-1=0) \to x+1 = x+1) |\!)$ Impl.(5)

`a := x+1`

$(\!| (a-1=0 \to 1 = x+1) \wedge (\neg(a-1=0) \to a = x+1) |\!)$ Ass.(4)

`if (a-1 == 0)`$\{$

$(\!| 1 = x+1 |\!)$ If(3′)

`y := 1`

$(\!| y = x+1 |\!)$ Ass.(2′)

$\}$ `else` $\{$

$(\!| a = x+1 |\!)$ If(3)

`y := a`

$(\!| y = x+1 |\!)$ Ass.(2)

$\}$

$(\!| y = x+1 |\!)$ If(1)

Instructiunea `while`

Instructiunea `while`:

- Dacă aplicăm substituţia înapoi în `while` intrăm într-un ciclu infinit.
- Putem depăşi această circularitate printr-o metodă creativă: Spre a demonstra

$$(\!|\, \phi \,|\!)\ \mathtt{while}\ B\ \{C\}\ (\!|\, \psi \,|\!)$$

trebuie să *ghicim (descoperim)* un invariant portivit η astfel încât

  - $\phi \rightarrow \eta$
  - $\eta \wedge \neg B \rightarrow \psi$
  - $(\!|\, \eta \,|\!)\ \mathtt{while}\ (B)\ \{C\}\ (\!|\, \eta \wedge \neg B \,|\!)$

Cautarea invariantilor

Sugestii:

- Dacă suntem *autorul* programului:
  - descriem atent relaţiile pe care le ştim între diversele variabile din program în punctul curent
  - daca verificarea eşuează, dar formulele sunt bune, probabil programul este greşit ori incomplet şi trebuie amendat
- Dacă suntem doar *verificator* al unui program dat
  - încercăm să găsim formule care să exprime relaţiile dintre variabile în diverse puncte din program
  - dacă verificarea eşuează, schimbăm formulele şi reîncercăm verificarea

Exemplu (factorial)

Exemplu: Pentru $x = 6$ execuţia este:

iteration	z	y	B
0	0	1	true
1	1	1	true
2	2	2	true
3	3	6	true
4	4	24	true
5	5	120	true
6	6	720	false

```
y := 1;
z := 0;
while(z != x) {
    z := z+1;
    y := y*z
}
```

Nu este dificil de *ghicit* că $y = z!$ este un bun invariant: pare a fi (1) *invariant*; (2) *suficient de slab* spre a fi adevărat la intrarea în `while`; şi (3) *suficient de tare* spre a implica (împreună cu negaţia condiţiei lui `while`) concluzia după `while`.

..Exemplu (factorial)

Demonstraţie pentru:

$\vdash_{par}$:

$(\!|\ \top\ |\!)$

```
y := 1;
z := 0;
while (z != x) {
    z := z+1;
    y := y*z;
}
```

$(\!|\ y = x!\ |\!)$

..Exemplu (factorial)

$\vdash_{par}$

```
    (| ⊤ |)
    (| 1 = 0! |)                         Implied (8)
y := 1;
    (| y = 0! |)                         Assignment (7)
z := 0;
    (| y = z! |)                         Assignment (6)
while (z != x) {
        (| y = z! ∧ z ≠ x |)             Inv.Hyp.& Guard (2)
        (| y·(z+1) = (z+1)! |)           Implied (5)
    z := z+1;
        (| y·z = z! |)                   Assignment (4)
    y := y*z;
        (| y = z! |)                     Assignment (3)
}
    (| y = z! ∧ ¬(z ≠ x) |)              Partial-while (2)
    (| y = x! |)                         Implied (1)
```

Exemplu `Min_Sum`

Min_Sum: Fie $a[1], \ldots, a[n]$ un vector cu valori întregi

- o *secţiune* este o zonă continuă $a[i], \ldots, a[j]$ cu $1 \leq i \leq j \leq n$;
- o *secţiune cu suma minimă* este o secţiune $a[i], \ldots, a[j]$ astfel încât suma $a[i] + \ldots + a[j]$ este cea mai mică din sumele tututor secţiunilor din a.

Exemplu:

- Fie $a = [-1, 3, 15, -6, 4, -5]$; o secţiune minimă este $[-6, 4, -5]$ cu suma -7

Sarcini: De găsit

- un program care să calculeze suma minimă a secţiunilor
- o descriere formală a specificaţiei intrare-ieşire
- o demonstraţie a corectitudinii parţială

..Exemplu `Min_Sum`

Soluţii:

- O soluţie simplă, dar neeficientă, este:
  Listăm toate secţiunile, calculăm sumele, reţinând-o pe cea mai mare - complexitate $O(n^3)$.

- Un algorithm mai complicat, dar linear, este:

```
k := 2;
t := a[1];
s := a[1];
while (k != n+1) {
    t := min(t+a[k],a[k]);
    s := min(s,t);
    k := k+1;
}
```

– *s* este folosit pentru a reţine suma minimă obţinută până în momentul curent

– *t* este folosit pentru a reţine suma minimă a tuturor secţiunilor care au un capăt în punctul curent

..Exemplu `Min_Sum`

O execuţie:

- Input $n = 6; a = [-1, 3, 15, -6, 4, -5]$

$iteration$	k	t	s
0	2	-1	-1
1	3	2	-1
2	4	15	-1
3	5	-6	-6
4	6	-2	-6
5	7	-7	-7

..Exemplu `Min_Sum`

Specificaţie:

- S1. $(\!|\ \top\ |\!)$ `Min_Sum` $(\!|\ \forall i,j(i \leq j \leq n \rightarrow s \leq S_{i,j})\ |\!)$
- S2. $(\!|\ \top\ |\!)$ `Min_Sum` $(\!|\ \exists i,j(i \leq j \leq n \wedge s = S_{i,j}\ |\!)$

Aici demonstrăm doar S1. Un invariant portivit este din două părţi, una pentru s, cealaltă pentru t:

$$\forall i,j(i \leq j < k \rightarrow s \leq S_{i,j}) \wedge \forall i(i < k \rightarrow t \leq S_{i,k-1})$$

descriind formal proprietăţile din enunţ: (1) s este suma secţiunii minime din $[1,\ldots,k-1]$, iar (2) t este suma secţiunii minime din $[1,\ldots,k-1]$ printre cele ce se termină în $k-1$.

..Exemplu `Min_Sum`

Cotectitutine parţială, proprietatea S1: $\vdash_{par}$

$(\!|\ \top\ |\!)$

$(\!|\ (a[1] \leq S_{1,1}) \wedge (a[1] \leq S_{1,1}\ |\!)$ `Implied (11)`

$(\!|\ \forall i(i < 2 \rightarrow a[1] \leq S_{i,1}) \wedge \forall i,j(i \leq j < 2 \rightarrow a[1] \leq S_{i,j})\ |\!)$ `Implied (10)`

```
k := 2;
```

$(\!|\ \forall i(i < k \rightarrow a[1] \leq S_{i,k-1}) \wedge \forall i,j(i \leq j < k \rightarrow a[1] \leq S_{i,j})\ |\!)$ `Assignment (9)`

```
t := a[1];
```

$(\!|\ \forall i(i < k \rightarrow t \leq S_{i,k-1}) \wedge \forall i,j(i \leq j < k \rightarrow a[1] \leq S_{i,j})\ |\!)$ `Assignment (8)`

```
s := a[1];
```

$(\!|\ \forall i(i < k \rightarrow t \leq S_{i,k-1}) \wedge \forall i,j(i \leq j < k \rightarrow s \leq S_{i,j})\ |\!)$ `Assignment (7)`

..Exemplu `Min_Sum`

`while (k != n+1) {`

$(\!|\, \forall i(i<k \rightarrow t \leq S_{i,k-1}) \wedge \forall i,j(i \leq j < k \rightarrow s \leq S_{i,j}) \wedge k \neq n+1 \,|\!)$ `Inv+Grd(2)`

$(\!|\, \forall i(i<k+1 \rightarrow min(t+a[k],a[k]) \leq S_{i,k}) \wedge \forall i,j(i \leq j < k+1 \rightarrow min(s,min(t+a[k],a[k])) \leq S_{i,j}) \,|\!)$ `Implied(lemma)(6)`

`t := min(t+a[k],a[k]);`

$(\!|\, \forall i(i<k+1 \rightarrow t \leq S_{i,k}) \wedge \forall i,j(i \leq j < k+1 \rightarrow min(s,t) \leq S_{i,j}) \,|\!)$ `Ass(5)`

`s := min(s,t);`

$(\!|\, \forall i(i<k+1 \rightarrow t \leq S_{i,k}) \wedge \forall i,j(i \leq j < k+1 \rightarrow s \leq S_{i,j}) \,|\!)$ `Ass(4)`

`k := k+1;`

$(\!|\, \forall i(i<k \rightarrow t \leq S_{i,k-1}) \wedge \forall i,j(i \leq j < k \rightarrow s \leq S_{i,j}) \,|\!)$ `Ass(3)`

`}`

$(\!|\, \forall i(i<k \rightarrow t \leq S_{i,k-1}) \wedge \forall i,j(i \leq j < k \rightarrow s \leq S_{i,j}) \wedge (k = n+1) \,|\!)$ `While(2)`

$(\!|\, \forall i,j(i \leq j \leq n \rightarrow s \leq S_{i,j}) \,|\!)$ `Implied(1)`

..Exemplu `Min_Sum`

Lemă: Dacă $S_{i,j}$ reprezintă suma numerelor dintre i şi j în vectorul a, i.e., $a[i]+\ldots+a[j]$, atunci:

1. *dacă* $\forall i(1 \leq i < k \rightarrow t \leq S_{i,k-1})$, *atunci* $\forall i(1 \leq i < k+1 \rightarrow min(t+a[k],a[k]) \leq S_{i,k})$

2. *dacă* $\forall i(1 \leq i < k \rightarrow t \leq S_{i,k-1}) \wedge \forall i,j(1 \leq i \leq j < k \rightarrow s \leq S_{i,j})$, *atunci* $\forall i,j(1 \leq i \leq j < k+1 \rightarrow min(s,t+a[k],a[k]) \leq S_{i,j})$

Dem. (1):

—dacă $i < k : t \leq S_{i,k-1} \rightarrow t+a[k] \leq S_{i,k-1}+a[k] = S_{i,k} \rightarrow min(t+a[k],a[k]) \leq S_{i,k}$

—dacă $i = k : min(t+a[k],a[k]) \leq a[k] = S_{k,k}$

Dem. (2):

—dacă $j < k : min(s,t+a[k],a[k]) \leq s \leq S_{i,j}$

—dacă $j = k : min(s,t+a[k],a[k]) \leq min(t+a[k],a[k]) \leq_{by\ (1)} S_{i,k}$

Corectitudinea totala

Corectitudinea totală: Ca şi în cazul logicii Floyd, *corectitudinea totală* constă din două părţi:

- *corectitudinea parţială* şi
- *terminarea*

Pentru programele while, regula nouă care se foloseşte este

`Total-while`:

$$\frac{(\!|\, \eta \wedge B \wedge 0 \leq E = E_0 \,|\!)\; C\; (\!|\, \eta \wedge 0 \leq E < E_0 \,|\!)}{(\!|\, \eta \wedge 0 \leq E \,|\!)\; \mathtt{while}\ (B)\ \{C\}\; (\!|\, \eta \wedge \neg B \,|\!)}\mathtt{Total-while}$$

Ce este nou aici, este o expresie E care descreşte cu fiecare buclă while; spre a o descrie se foloseşte notaţia E_0 pentru valoarea lui E înainte de execuţia buclei `while`.)

Exemplu (factorial)

Demonstraţie pentru:

$\vdash_{tot}$

$$(\!|\, x \geq 0 \,|\!)$$

```
y := 1;
z := 0;
while (z != x) {
    z := z+1;
    y := y*z;
}
```

$$(\!|\, y = x! \,|\!)$$

..Exemplu (factorial)

$(\!| x \geq 0 |\!)$

$(\!| 1 = 0! \wedge 0 \leq x - 0 |\!)$ `Implied (8)`

```
y := 1;
```

$(\!| y = 0! \wedge 0 \leq x - 0 |\!)$ `Assignment (7)`

```
z := 0;
```

$(\!| y = z! \wedge 0 \leq x - z |\!)$ `Assignment (6)`

```
while (z != x) {
```

$(\!| y = z! \wedge z \neq x \wedge 0 \leq x - z = E_0 |\!)$ `Inv.Hyp.& Guard (2)`

$(\!| y \cdot (z+1) = (z+1)! \wedge 0 \leq x - (z+1) < E_0 |\!)$ `Implied (5)`

```
  z := z+1;
```

$(\!| y \cdot z = z! \wedge 0 \leq x - z < E_0 |\!)$ `Assignment (4)`

```
  y := y*z;
```

$(\!| y = z! \wedge 0 \leq x - z < E_0 |\!)$ `Assignment (3)`

```
}
```

$(\!| y = z! \wedge \neg(z \neq x) |\!)$ `Total-while (2)`

$(\!| y = z! |\!)$ `Implied (1)`

Logica Floyd-Hoare

Cuprins:

- Generalitati
- Logica Floyd
- Logica Hoare
- Verificări de programe structurate
- *Extensia la programe interactive*
- Concluzii, diverse, etc.

Extensia la programe interactive

a se insera...

Logica Floyd-Hoare

Cuprins:

- Generalitati
- Logica Floyd
- Logica Hoare
- Verificări de programe structurate
- Extensia la programe interactive
- *Concluzii, diverse, etc.*

Concluzii, diverse, etc.

a se insera...