Curs 10

2021-2022

Fundamentele limbajelor de programare

Cuprins

O implementare a limbajului IMP în Prolog

Sintaxa BNF a limbajului IMP

$E ::= n \mid x$
$\quad \mid$ `++` (E)
$\quad \mid E$ `+` $E \mid E$ `-` $E \mid E$ `*` E

$B ::=$ `true` $\mid$ `false`
$\quad \mid E$ `=<` $E \mid E$ `>=` $E \mid E$ `==` E
$\quad \mid$ `not(`B`)` $\mid$ `and(`B `,` B`)` $\mid$ `or(`B `,` B`)`

$C ::=$ `skip`
$\quad \mid x$ `=` E
$\quad \mid$ `if(` B `,` C `,` C `)`
$\quad \mid$ `while(` B `,` C `)`
$\quad \mid$ `{` C `}` $\mid C$ `;` C

$P ::=$ `{` C `},` E

Decizii de implementare

- `{}` si `;` sunt operatori

```
:- op(100, xf, {}).
:- op(1100, yf, ;).
```

- definim un predicat pentru fiecare categorie sintactică

```
stmt(while(BE,St)) :- bexp(BE),  stmt(St).
```

- `while`, `if`, `and`, etc sunt functori în Prolog
  `while(true,skip)` este un termen compus
- `,` are semnificatia obisnuită
- pentru valori numerice folosim întregii din Prolog

```
aexp(I) :- integer(I).
```

- pentru identificatori folosim atomii din Prolog

```
aexp(X) :- atom(X).
```

Expresiile aritmetice

$$
\begin{aligned}
E ::= {} & n \mid x \\
& \mid {+}{+}\,(E) \\
& \mid E + E \mid E - E \mid E * E
\end{aligned}
$$

Prolog

aexp(++(X)):- atom(X).

```
aexp(I) :- integer(I).
aexp(X) :- atom(X).
aexp(A1 + A2) :- aexp(A1), aexp(A2).
...
```

Expresiile aritmetice

Exemplu

```
?- aexp(1000).
true.
?- aexp(id).
true.
?- aexp(id + 1000).
true.
?- aexp(2 + 1000).
true.
?- aexp(x * y).
true.
?- aexp(- x).
false.
```

Expresiile booleene

$$
\begin{aligned}
B ::= &\ \texttt{true} \mid \texttt{false} \\
&\mid E =< E \mid E >= E \mid E == E \\
&\mid \texttt{not}(B) \mid \texttt{and}(B\ ,\ B) \mid \texttt{or}(B\ ,\ B)
\end{aligned}
$$

Prolog

```
bexp(true). bexp(false).
bexp(and(BE1,BE2)) :- bexp(BE1), bexp(BE2).

bexp(A1 =< A2) :- aexp(A1), aexp(A2).
...
```

Expresiile booleene

Exemplu

```
?- bexp(true).
true.
?- bexp(id).
false.
?- bexp(not(1 =< 2)).
true.
?- bexp(or(1 =< 2,true)).
true.
?- bexp(or(a =< b,true)).
true.
?- bexp(not(a)).
false.
?- bexp(!(a)).
false.
```

Instructiunile

$$
\begin{aligned}
C ::= &\ \texttt{skip} \\
\mid &\ x = E \\
\mid &\ \texttt{if(}\ B\ \texttt{,}\ C\ \texttt{,}\ C\ \texttt{)} \\
\mid &\ \texttt{while(}\ B\ \texttt{,}\ C\ \texttt{)} \\
\mid &\ \texttt{\{}\ C\ \texttt{\}} \mid C\ \texttt{;}\ C
\end{aligned}
$$

Prolog

```
stmt(skip).
stmt(X = AE) :- atom(X), aexp(AE).
stmt(St1;St2) :- stmt(St1), stmt(St2).
stmt(if(BE,St1,St2)) :-  bexp(BE), stmt(St1), stmt(St2).
...
```

Instructiunile

Exemplu

```
?- stmt(id = 5).
true.
?- stmt(id = a).
true.
?- stmt(3 = 6).
false.
?- stmt(if(true, x=2;y=3, x=1;y=0)).
true.
?- stmt(while(x =< 0,skip)).
true.
?- stmt(while(x =< 0,)).
false.
?- stmt(while(x =< 0,skip)).
true .
```

Programele

$P ::= \{\ C\ \},\ E$

Prolog

```
program(St,AE) :- stmt(St), aexp(AE).
```

Exemplu

```
test0 :- program( {x = 10 ; sum = 0;
                   while(0 =< x,
                         {sum = sum + x; x = x-1}
                        )}
                   , sum).
```

?- test0.
true.

O implementare a semanticii small-step

Semantica small-step

- Defineste cel mai mic pas de executie ca o relatie de tranzitie intre configuratii:
  $\langle cod\ ,\ \sigma\rangle \to \langle cod'\ ,\ \sigma'\rangle$ `smallstep(Cod,S1,Cod',S2)`
- Executia se obtine ca o succesiune de astfel de tranzitii.
- Starea executiei unui program IMP la un moment dat este o functie partială: $\sigma = n \mapsto 10, sum \mapsto 0$, etc.

Semantica small-step

- Defineste cel mai mic pas de executie ca o relatie de tranzitie intre configuratii:
  $\langle cod\ ,\ \sigma \rangle \to \langle cod'\ ,\ \sigma' \rangle$ `smallstep(Cod,S1,Cod',S2)`
- Executia se obtine ca o succesiune de astfel de tranzitii.
- Starea executiei unui program IMP la un moment dat este o functie partială: $\sigma = n \mapsto 10, sum \mapsto 0$, etc.

Reprezentarea stărilor în Prolog

```
get(S,X,I) :- member(vi(X,I),S).
get(_,_,0).
set(S,X,I,[vi(X,I)|S1]) :- del(S,X,S1).

del([vi(X,_)|S],X,S).
del([H|S],X,[H|S1]) :- del(S,X,S1) .
del([],_,[]).
```

Semantica expresiilor aritmetice

- Semantica unei variabile

  $\langle x\ ,\ \sigma\rangle \to \langle i\ ,\ \sigma\rangle \quad$ *dacă* $i = \sigma(x)$

Prolog

```
smallstepA(X,S,I,S) :-
  atom(X),
  get(S,X,I).
```

Semantica expresiilor aritmetice

- Semantica adunării a două expresii aritmetice

$\langle i_1 + i_2 , \sigma \rangle \to \langle i , \sigma \rangle \quad \textit{dacă } i = i_1 + i_2$

$$\frac{\langle a_1 , \sigma \rangle \to \langle a_1' , \sigma \rangle}{\langle a_1 + a_2 , \sigma \rangle \to \langle a_1' + a_2 , \sigma \rangle} \qquad \frac{\langle a_2 , \sigma \rangle \to \langle a_2' , \sigma \rangle}{\langle a_1 + a_2 , \sigma \rangle \to \langle a_1 + a_2' , \sigma \rangle}$$

- Ordine nespecificată de evaluare a argumentelor

Prolog

```
smallstepA(I1 + I2,S,I,S):- integer(I1),integer(I2),
                            I is I1 + I2.

smallstepA(I + AE1,S,I + AE2,S):- integer(I),
                                  smallstepA(AE1,S,AE2,S).

smallstepA(AE1 + AE,S,AE2 + AE,S):-  ...
```

Semantica expresiilor aritmetice

Exemplu

```
?- smallstepA(a + b, [vi(a,1),vi(b,2)],AE, S).
AE = 1+b,
S = [vi(a, 1), vi(b, 2)] .

?- smallstepA(1 + b, [vi(a,1),vi(b,2)],AE, S).
AE = 1+2,
S = [vi(a, 1), vi(b, 2)] .

?- smallstepA(1 + 2, [vi(a,1),vi(b,2)],AE, S).
AE = 3,
S = [vi(a, 1), vi(b, 2)]
```

Semantica expresiilor aritmetice

Exemplu

```
?- smallstepA(a + b, [vi(a,1),vi(b,2)],AE, S).
AE = 1+b,
S = [vi(a, 1), vi(b, 2)] .

?- smallstepA(1 + b, [vi(a,1),vi(b,2)],AE, S).
AE = 1+2,
S = [vi(a, 1), vi(b, 2)] .

?- smallstepA(1 + 2, [vi(a,1),vi(b,2)],AE, S).
AE = 3,
S = [vi(a, 1), vi(b, 2)]
```

- Semantica * si – se definesc similar.

Semantica expresiilor booleene

- Semantica operatorului de comparatie

$\langle i_1 =< i_2 \ , \ \sigma \rangle \rightarrow \langle \texttt{false} \ , \ \sigma \rangle \quad \textit{dacă } i_1 > i_2$

$\langle i_1 =< i_2 \ , \ \sigma \rangle \rightarrow \langle \texttt{true} \ , \ \sigma \rangle \quad \textit{dacă } i_1 \leq i_2$

$$\frac{\langle a_1 \ , \ \sigma \rangle \rightarrow \langle a_1' \ , \ \sigma \rangle}{\langle a_1 =< a_2 \ , \ \sigma \rangle \rightarrow \langle a_1' =< a_2 \ , \ \sigma \rangle} \qquad \frac{\langle a_2 \ , \ \sigma \rangle \rightarrow \langle a_2' \ , \ \sigma \rangle}{\langle a_1 =< a_2 \ , \ \sigma \rangle \rightarrow \langle a_1 =< a_2' \ , \ \sigma \rangle}$$

Prolog

```
smallstepB(I1 =< I2,S,true,S):- integer(I1),integer(I2),
                                (I1 =< I2).
smallstepB(I1 =< I2,S,false,S):- integer(I1),integer(I2),
                                 (I1 > I2).
smallstepB(I =< AE1,S,I =< AE2,S):- ...
smallstepB(AE1 =< AE,S,AE2 =< AE,S):- ...
```

Semantica expresiilor Booleene

- Semantica negatiei

$\langle \texttt{not(true)}\ ,\ \sigma\rangle \rightarrow \langle \texttt{false}\ ,\ \sigma\rangle$

$\langle \texttt{not(false)}\ ,\ \sigma\rangle \rightarrow \langle \texttt{true}\ ,\ \sigma\rangle$

$$\frac{\langle a\ ,\ \sigma\rangle \rightarrow \langle a'\ ,\ \sigma\rangle}{\langle \texttt{not}\ (a)\ ,\ \sigma\rangle \rightarrow \langle \texttt{not}\ (a')\ ,\ \sigma\rangle}$$

Prolog

```
smallstepB(not(true),S,false,S) .
smallstepB(not(false),S,true,S) .

smallstepB(not(BE1),S,not(BE2),S) :- ...
```

Semantica compunerii si a blocurilor

- Semantica blocurilor

  $\langle \{\ s\ \}\ ,\ \sigma\rangle \rightarrow \langle s\ ,\ \sigma\rangle$

- Semantica compunerii secventiale

  $\langle \{\};\ s_2\ ,\ \sigma\rangle \rightarrow \langle s_2\ ,\ \sigma\rangle \qquad \dfrac{\langle s_1\ ,\ \sigma\rangle \rightarrow \langle s_1'\ ,\ \sigma'\rangle}{\langle s_1 ; s_2\ ,\ \sigma\rangle \rightarrow \langle s_1' ; s_2\ ,\ \sigma'\rangle}$

Prolog

```
smallstepS({E},S,E,S).

smallstepS((skip;St2),S,St2,S).
smallstepS((St1;St),S1,(St2;St),S2) :- ...
```

Semantica atribuirii

- Semantica atribuirii

$$\langle x = i \ , \ \sigma \rangle \rightarrow \langle \{\} \ , \ \sigma' \rangle \quad \textit{dacă } \sigma' = \sigma[i/x]$$

$$\frac{\langle a \ , \ \sigma \rangle \rightarrow \langle a' \ , \ \sigma \rangle}{\langle x = a \ , \ \sigma \rangle \rightarrow \langle x = a' \ ; \ , \ \sigma \rangle}$$

Prolog

```
smallstepS(X = AE,S,skip,S1) :- integer(AE),set(S,X,AE,S1).

smallstepS(X = AE1,S,X = AE2,S) :- ...
```

Semantica lui `if`

- Semantica lui `if`

$$\langle \texttt{if}\,(\texttt{true}, bl_1, bl_2)\;,\;\sigma\rangle \to \langle bl_1\;,\;\sigma\rangle$$

$$\langle \texttt{if}\,(\texttt{false}, bl_1, bl_2)\;,\;\sigma\rangle \to \langle bl_2\;,\;\sigma\rangle$$

$$\frac{\langle b\;,\;\sigma\rangle \to \langle b'\;,\;\sigma\rangle}{\langle \texttt{if}\,(b, bl_1, bl_2)\;,\;\sigma\rangle \to \langle \texttt{if}\,(b', bl_1, bl_2)\;,\;\sigma\rangle}$$

Prolog

```
smallstepS(if(true,St1,_),S,St1,S).
smallstepS(if(false,_,St2),S,St2,S).

smallstepS(if(BE1,St1,St2),S,if(BE2,St1,St2),S) :- ...
```

Semantica lui `while`

- Semantica lui `while`

  $\langle \texttt{while}\ (b, bl)\ ,\ \sigma\rangle \rightarrow \langle \texttt{if}\ (b, bl\ ;\texttt{while}\ (b, bl), \texttt{skip})\ ,\ \sigma\rangle$

Prolog

```
smallstepS(while(BE,St),S,if(BE,(St;while(BE,St)),skip),S).
```

Semantica programelor

- Semantica programelor

$$\frac{\langle a_1 \ , \ \sigma_1 \rangle \rightarrow \langle a_2 \ , \ \sigma_2 \rangle}{\langle (\texttt{skip}, a_1) \ , \ \sigma_1 \rangle \rightarrow \langle (\texttt{skip}, a_2) \ , \ \sigma_2 \rangle}$$

$$\frac{\langle s_1 \ , \ \sigma_1 \rangle \rightarrow \langle s_2 \ , \ \sigma_2 \rangle}{\langle (s_1, a) \ , \ \sigma_1 \rangle \rightarrow \langle (s_2, a) \ , \ \sigma_2 \rangle}$$

Prolog

```
smallstepP(skip,AE1,S1,skip,AE2,S2) :-
                                   smallstepA(AE1,S1,AE2,S2) .
smallstepP(St1,AE,S1,St2,AE,S2) :-
                                   smallstepS(St1,S1,St2,S2).
```

Executia programelor

Prolog

```
run(skip,I,_,I):- integer(I).
run(St1,AE1,S1,I) :- smallstepP(St1,AE1,S1,St2,AE2,S2),
                     run(St2,AE2,S2,I).

run_program(Name) :- defpg(Name,{P},E), run(P,E, [],I),
                     write(I).
```

Exemplu

```
defpg(pg2, {x = 10 ; sum = 0; while(0 =< x, {
                                  sum = sum + x;
                                  x = x - 1})},sum)
```

?- run_program(pg2).
55
true

Executia programelor: `trace`

Putem defini o functie care ne permite să urmărim executia unui program în implementarea noastră?

Executia programelor: `trace`

Putem defini o functie care ne permite să urmărim executia unui program în implementarea noastră?

Prolog

```
mytrace(skip,I,_) :- integer(I).
mytrace(St1,AE1,S1) :-  smallstepP(St1,AE1,S1,St2,AE2,S2),
                        write(St2),nl,
                        write(AE2),nl,
                        write(S2),nl,
                        mytrace(St2,AE2,S2).

trace_program(Name) :- defpg(Name,{P},E),
                       mytrace(P,E,[]).
```

Executia programelor: `trace_program`

Exemplu

```
?- trace_program(pg2).
. . .
[vi(x,-1),vi(sum,55)]
if(0=<x,(sum=sum+x;x=x-1;while(0=<x,sum=sum+x;x=x-1)),skip)
sum
[vi(x,-1),vi(sum,55)]
if(0=< -1,(sum=sum+x;x=x-1;while(0=<x,sum=sum+x;x=x-1)),skip)
sum
[vi(x,-1),vi(sum,55)]
if(false,(sum=sum+x;x=x-1;while(0=<x,sum=sum+x;x=x-1)),skip)
sum
[vi(x,-1),vi(sum,55)]
skip
sum
[vi(x,-1),vi(sum,55)]
skip
55
[vi(x,-1),vi(sum,55)]
true .
```

Semantica axiomatică

Semantica Axiomatică

- Dezvoltata de Tony Hoare în 1969
  (inspirată de rezultatele lui Robert Floyd).
- Defineşte triplete (triplete Hoare) de forma

$$\{Pre\}\ S\ \{Post\}$$

  unde:
  - S este o instrucțiune (Stmt)
  - *Pre* (precondiție), respectiv *Post* (postcondiție) sunt asertiuni logice asupra stării sistemului înaintea, respectiv după execuția lui S
  - Limbajul aserţiunilor este un limbaj de ordinul I.

Semantica Axiomatică

Tripletul $\{Pre\}\ S\ \{Post\}$ este corect dacă:

- dacă programul se execută dintr-o stare iniţială care satisface Pre
- şi execuţia se termină
- atunci se ajunge într-o stare finală care satisface $Post$.

Putem verifica corectitudine partiala.

Semantica Axiomatică

Tripletul $\{Pre\}\ S\ \{Post\}$ este corect dacă:

- dacă programul se execută dintr-o stare iniţială care satisface *Pre*
- şi execuţia se termină
- atunci se ajunge într-o stare finală care satisface *Post*.

Putem verifica corectitudine partiala.

Exemplu

- $\{\mathtt{x} = 1\}$ `x = x+1` $\{\mathtt{x} = 2\}$ este corect
- $\{\mathtt{x} = 1\}$ `x = x+1` $\{\mathtt{x} = 3\}$ **nu** este corect
- $\{\top\}$ `if (x<=y) z=x; else z=y;` $\{\mathtt{z} = \mathit{min}(\mathtt{x}, \mathtt{y})\}$ este corect

Semantica Axiomatică

Tripletul $\{Pre\}\ S\ \{Post\}$ este corect dacă:

- dacă programul se execută dintr-o stare iniţială care satisface *Pre*
- şi execuţia se termină
- atunci se ajunge într-o stare finală care satisface *Post*.

Putem verifica corectitudine partiala.

Exemplu

- $\{\mathtt{x} = 1\}$ x = x+1 $\{\mathtt{x} = 2\}$ este corect
- $\{\mathtt{x} = 1\}$ `x = x+1` $\{\mathtt{x} = 3\}$ **nu** este corect
- $\{\top\}$ `if (x<=y) z=x; else z=y;` $\{\mathtt{z} = min(\mathtt{x}, \mathtt{y})\}$ este corect

Se asociază fiecărei construcții sintactice Stmt o regulă de deducție care defineşte recursiv tripletele Hoare descrise mai sus.

Sistem de reguli pentru logica Floyd-Hoare

$$(\rightarrow) \quad \frac{P1 \rightarrow P2 \quad \{P2\}\ c\ \{Q2\} \quad Q2 \rightarrow Q1}{\{P1\}\ c\ \{Q1\}}$$

$$(\vee) \quad \frac{\{P1\}\ c\ \{Q\} \quad \{P2\}\ c\ \{Q\}}{\{P1 \vee P2\}\ c\ \{Q\}}$$

$$(\wedge) \quad \frac{\{P\}\ c\ \{Q1\} \quad \{P\}\ c\ \{Q2\}}{\{P\}\ c\ \{Q1 \wedge Q2\}}$$

Logica Floyd-Hoare pentru IMP1

$$(\text{SKIP}) \quad \frac{}{\{P\}\ \{\}\ \{P\}}$$

$$(\text{SEQ}) \quad \frac{\{P\}\ c1\ \{Q\} \quad \{Q\}\ c2\ \{R\}}{\{P\}\ c1; c2\ \{R\}}$$

$$(\text{ASIGN}) \quad \frac{}{\{P[x/e]\}\ x = e\ \{P\}}$$

$$(\text{IF}) \quad \frac{\{b \land P\}\ c1\ \{Q\} \quad \{\neg b \land P\}\ c2\ \{Q\}}{\{P\}\ \texttt{if}\ (b) c1\ \texttt{else}\ c2\ \{Q\}}$$

$$(\text{WHILE}) \quad \frac{\{b \land P\}\ c\ \{P\}}{\{P\}\ \texttt{while}\ (b)\ c\ \{\neg b \land P\}}$$

Logica Floyd-Hoare pentru IMP1

- regula pentru atribuire

$$(\textsc{Asign}) \quad \overline{\{P[x/e]\}\ x = e\ \{P\}}$$

Exemplu

$\{\mathtt{x} + \mathtt{y} = \mathtt{y} + 10\}$ `x = x + y` $\{\mathtt{x} = \mathtt{y} + 10\}$

Logica Floyd-Hoare pentru IMP1

- regula pentru atribuire

$$(\textsc{Asign}) \quad \frac{}{\{P[x/e]\}\ x = e\ \{P\}}$$

Exemplu

$\{\mathtt{x}+\mathtt{y}=\mathtt{y}+10\}$ `x = x + y` $\{\mathtt{x}=\mathtt{y}+10\}$

- regula pentru condiţii

$$(\textsc{If}) \quad \frac{\{b \land P\}\ c1\ \{Q\} \quad \{\neg b \land P\}\ c2\ \{Q\}}{\{P\}\ \mathtt{if}\ (b) c1\ \mathtt{else}\ c2\ \{Q\}}$$

Exemplu

Pentru a demonstra $\{\top\}$ `if (x<=y) z=x; else z=y;` $\{\mathtt{z}=min(\mathtt{x},\mathtt{y})\}$ este suficient să demonstrăm

- $\{\mathtt{x}\leq\mathtt{y}\}$ `z=x` $\{\mathtt{z}=min(\mathtt{x},\mathtt{y})\}$
- $\{\neg(\mathtt{x}\leq\mathtt{y})\}$ `z=y` $\{\mathtt{z}=min(\mathtt{x},\mathtt{y})\}$

Invarianţi pentru `while`

Cum demonstrăm $\{P\}$ `while(b)c` $\{Q\}$?

- Se determină un invariant I şi se foloseşte următoarea regulă:

$$(\textsc{Inv}) \quad \frac{P \to I \quad \{b \land I\}\, c\, \{I\} \quad (I \land \neg b) \to Q}{\{P\}\ \mathtt{while}\ (b)\, c\ \{Q\}}$$

Invarianţi pentru `while`

Cum demonstrăm $\{P\}$ `while(b)c` $\{Q\}$?

- Se determină un invariant I şi se foloseşte următoarea regulă:

$$(\textsc{Inv}) \quad \frac{P \to I \quad \{b \land I\}\ c\ \{I\} \quad (I \land \neg b) \to Q}{\{P\}\ \mathtt{while}\ (b)\ c\ \{Q\}}$$

Invariantul trebuie să satisfacă următoarele proprietăţi:

- să fie adevărat iniţial
- să rămână adevărat după executarea unui ciclu
- să implice postcondiţia la ieşirea din buclă

Invarianţi pentru `while`

$\{\mathtt{x} = 0 \ \wedge \ 0 \leq n \ \wedge \mathtt{y} = 1\}$

```
while (x < n) { x = x + 1; y = y * x}
```

$\{\mathtt{y} = n!\}$

Invarianţi pentru `while`

$\{\mathtt{x} = 0 \ \wedge \ 0 \leq n \ \wedge \mathtt{y} = 1\}$

```
while (x < n) { x = x + 1; y = y * x}
```

$\{\mathtt{y} = n!\}$

- Invariantul I este $\mathtt{y} = x!$

Invarianţi pentru `while`

$\{\mathtt{x} = 0 \ \wedge \ 0 \leq n \ \wedge \mathtt{y} = 1\}$

```
while (x < n) { x = x + 1; y = y * x}
```

$\{\mathtt{y} = n!\}$

- Invariantul I este $\mathtt{y} = x!$
- $(\mathtt{x} = 0 \ \wedge \ 0 \leq n \ \wedge \mathtt{y} = 1) \to I$
- $\{I \wedge (x < n)\}$ `x = x + 1; y = y * x` $\{ I \}$
- $I \wedge \neg(\mathtt{x} < n) \to (\mathtt{y} = n!)$

Dafny

- Dezvoltat la Microsoft Research
- Un limbaj imperativ compilat open-source
- Suportă demonstraţii formale folosind precondiţii, postcondiţii, invarianţi de bucle
- Demonstrează şi terminarea programelor
- Concepte din diferite paradigme de programare
  - Programare imperativă: `if`, `while`, `:=`, ...
  - Programare funcţională: `function`, `datatype`, ...
  - ...

Dafny

- Pagina limbajului Dafny

  https://www.microsoft.com/en-us/research/project/dafny-a-language-and-program-verifier-for-functional-correctness/

- Pagina de Github

  https://github.com/dafny-lang/dafny

- Dafny in browser

  http://cse-212294.cse.chalmers.se/courses/tdv/dafny/

- Tutorial

  https://dafny-lang.github.io/dafny/OnlineTutorial/guide

Dafny - Hello World

Dafny - fără erori

```
method Main() {
  print "hello, Dafny";
  assert 2 < 10;
}
```

Dafny - Hello World

Dafny - fără erori

```
method Main() {
  print "hello, Dafny";
  assert 2 < 10;
}
```

Dafny - erori

```
method Main() {
  print "hello, Dafny";
  assert 10 < 2;
}
```

Dafny - maximul a două numere

Următoarea metodă calculează maximul a doi întregi:

Dafny

```
method max (x : int, y : int) returns (z : int)
{
 if (x <= y) { return y; }
 return x;
}
```

Dar cum verificăm formal acest lucru?

Dafny - maximul a două numere

Adăugăm postcondiţii!

Dafny

```
method max (x : int, y : int) returns (z : int)
ensures (x <= z) && (y <= z)
{
 if (x <= y) { return y; }
 return x;
}
```

Dafny - maximul a două numere

Adăugăm postcondiţii!

Dafny

```
method max (x : int, y : int) returns (z : int)
ensures (x <= z) && (y <= z)
{
 if (x <= y) { return y; }
 return x;
}
```

Dar este de ajuns condiţia de mai sus?

Dafny - maximul a două numere

Adăugăm postcondiţii!

Dafny

```
method max (x : int, y : int) returns (z : int)
ensures (x <= z) && (y <= z)
{
 if (x <= y) { return y; }
 return x;
}
```

Dar este de ajuns condiţia de mai sus?
O metodă este reprezentată prin precondiţiile şi postcondiţiile pe care le satisface, iar codul este "ignorat" ulterior.
În exemplul de mai sus, pentru `x = 3` şi `y = 6`, `z = 7` satisface postcondiţia (dacă ignorăm codul) dar nu este maximul dintre `x` şi `y`.

Dafny - maximul a două numere

Dafny

```
method max (x : int, y : int) returns (z : int)
ensures (x <= z) && (y <= z)
ensures (x == z) || (y == z)
{
 if (x <= y) { return y; }
 return x;
}
```

Dafny - suma primelor n numere impare

Cum demonstram formal că suma primelor n numere impare este n^2?

- $1 = 1 = 1^2$
- $1 + 3 = 4 = 2^2$
- $1 + 3 + 5 = 9 = 3^2$
- $1 + 3 + 5 + 7 = 16 = 4^2$

Dafny - suma primelor n numere impare

Cum demonstram formal că suma primelor n numere impare este n^2?

Dafny

```
method oddSum(n: int) returns (s: int)
{
 var i: int := 0;
 s := 0;
 while i != n
 {
  i := i + 1;
  s := s + (2*i - 1);
 }
}
```

Dafny - suma primelor n numere impare

Adăugăm postcondiţia!

Dafny

```
method oddSum(n: int) returns (s: int)
ensures s == n * n
{
 var i: int := 0;
 s := 0;
 while i != n
 {
  i := i + 1;
  s := s + (2*i - 1);
 }
}
```

Dar Dafny nu reuşeşte să o demonstreze!

Dafny - suma primelor n numere impare

Adăugăm postcondiţia!

Dafny

```
method oddSum(n: int) returns (s: int)
ensures s == n * n
{
 var i: int := 0;
 s := 0;
 while i != n
 {
  i := i + 1;
  s := s + (2*i - 1);
 }
}
```

Dar Dafny nu reuşeşte să o demonstreze! Trebuie să ii spunem ce se întâmplă în buclă prin invarianţi.

Dafny - suma primelor n numere impare

Dafny

```
method oddSum(n: int) returns (s: int)
ensures s == n * n
{
 var i: int := 0;
 s := 0;
 while i != n
 invariant s == i*i
 {
  i := i + 1;
  s := s + (2*i - 1);
 }
}
```

Dafny - suma primelor n numere impare

Dafny

```
method oddSum(n: int) returns (s: int)
ensures s == n * n
{
 var i: int := 0;
 s := 0;
 while i != n
 invariant s == i*i
 {
  i := i + 1;
  s := s + (2*i - 1);
 }
}
```

Acum Dafny se plânge că nu reuşeşte să demonstreze terminarea!
Adăugăm un nou invariant care ne asigură că *i* nu depaşeşte *n*.

Dafny - suma primelor n numere impare

Dafny

```
method oddSum(n: int) returns (s: int)
ensures s == n * n
{
 var i: int := 0;
 s := 0;
 while i != n
 invariant 0 <= i <= n
 invariant s == i*i
 {
  i := i + 1;
  s := s + (2*i - 1);
 }
}
```

Dafny - suma primelor n numere impare

Dafny

```
method oddSum(n: int) returns (s: int)
ensures s == n * n
{
 var i: int := 0;
 s := 0;
 while i != n
 invariant 0 <= i <= n
 invariant s == i*i
 {
  i := i + 1;
  s := s + (2*i - 1);
 }
}
```

Totuşi Dafny nu reuşeşte să demonstreze postcondiţia! De ce?

Dafny - suma primelor n numere impare

Dafny

```
method oddSum(n: int) returns (s: int)
ensures s == n * n
{
 var i: int := 0;
 s := 0;
 while i != n
 invariant 0 <= i <= n
 invariant s == i*i
 {
  i := i + 1;
  s := s + (2*i - 1);
 }
}
```

Totuşi Dafny nu reuşeşte să demonstreze postcondiţia! De ce?
Ce se întâmplă dacă $n < 0$?

Dafny - suma primelor n numere impare

Dafny

```
method oddSum(n: int) returns (s: int)
requires 0 <= n
ensures s == n * n
{
 var i: int := 0;
 s := 0;
 while i != n
 invariant 0 <= i <= n
 invariant s == i*i
 {
  i := i + 1;
  s := s + (2*i - 1);
 }
}
```

Dafny - funcţia factorial

Dafny

```
function factorial(n: nat): nat
{ if n==0 then 1 else n*factorial(n-1) }

method CheckFactorial(n: nat) returns (r: nat)
  ensures r == factorial(n)
{
  var i := 0;
  r := 1;
  while i < n
    invariant r == factorial(i)
    invariant 0 <= i <= n
   {
    i := i + 1;
    r := r * i;
    }}
```

Semantica denotaţională (opţional)

Semantica denotațională

- Introdusă de Christopher Strachey şi Dana Scott (1970)
- Semantica operaţională, ca un interpretor, descrie cum să evaluăm un program.
- Semantica denotaţională, ca un compilator, descrie o traducere a limbajului într-un limbaj diferit cu semantică cunoscută, anume matematica.
- Semantica denotaţională defineşte ce înseamnă un program ca o funcţie matematică.

Semantica denotațională

- Definim stările memoriei ca fiind funcţii parţiale de la mulţimea identificatorilor la mulţimea valorilor:

$$\mathit{State} = \mathit{Id} \rightharpoonup \mathbb{Z}$$

- Asociem fiecărei categorii sintactice o categorie semantică.
- Fiecare construcţie sintactică va avea o denotaţie (interpretare) în categoria semantică respectivă.

Semantica denotațională

- Definim stările memoriei ca fiind funcţii parţiale de la mulţimea identificatorilor la mulţimea valorilor:

$$State = Id \rightharpoonup \mathbb{Z}$$

- Asociem fiecărei categorii sintactice o categorie semantică.
- Fiecare construcţie sintactică va avea o denotaţie (interpretare) în categoria semantică respectivă.De exemplu:
  - denotaţia unei expresii aritmetice este o funcţie parţială de la mulţimea stărilor memoriei la mulţimea valorilor ($\mathbb{Z}$):

$$[[\_]] : AExp \to (State \rightharpoonup \mathbb{Z})$$

  - denotaţia unei instrucţiuni este o funcţie parţială de la mulţimea stărilor memoriei la mulţimea stărilor memoriei:

$$[[\_]] : Stmt \to (State \rightharpoonup State)$$

Semantica denotațională

$State = Id \rightharpoonup \mathbb{Z}$
$[[\_]] : AExp \to (State \rightharpoonup \mathbb{Z})$
$[[\_]] : Stmt \to (State \rightharpoonup State)$

Atribuirea: $x = \mathrm{expr}$

- Asociem expresiilor aritmetice funcții de la starea memoriei la valori:
- Asociem instrucțiunilor funcții de la starea memoriei la starea (următoare) a memoriei.

Semantica denotațională

$State = Id \rightharpoonup \mathbb{Z}$
$[[\_]] : AExp \to (State \rightharpoonup \mathbb{Z})$
$[[\_]] : Stmt \to (State \rightharpoonup State)$

Atribuirea: $x = \mathrm{expr}$

- Asociem expresiilor aritmetice funcții de la starea memoriei la valori:
  - Funcția constantă $[[1]](s) = 1$
  - Funcția care selectează valoarea unui identificator $[[x]](s) = s(x)$
  - „Morfismul de adunare” $[[e1 + e2]](s) = [[e1]](s) + [[e2]](s)$.
- Asociem instrucțiunilor funcții de la starea memoriei la starea (următoare) a memoriei.
  - $[[x = e]](s)(y) = \begin{cases} s(y), \text{ dacă } y \neq x \\ [[e]](s), \text{ dacă } y = x \end{cases}$

Semantica denotațională: expresii

$$State = Id \rightharpoonup \mathbb{Z}$$

- Domenii semantice:
  $[\![\_]\!] : AExp \to (State \rightharpoonup \mathbb{Z})$
  $[\![\_]\!] : BExp \to (State \rightharpoonup \{T, F\})$
  $[\![\_]\!] : Stmt \to (State \rightharpoonup State)$

Semantica denotațională: expresii

$$State = Id \rightharpoonup \mathbb{Z}$$

- Domenii semantice:
  $[[\_]] : AExp \to (State \rightharpoonup \mathbb{Z})$
  $[[\_]] : BExp \to (State \rightharpoonup \{T, F\})$
  $[[\_]] : Stmt \to (State \rightharpoonup State)$
- Semantica denotaţională este compoziţională:
  - semantica expresiilor aritmetice
    $[[n]](s) = n$
    $[[x]](s) = s(x)$
    $[[e1 + e2]](s) = [[e1]](s) + [[e2]](s)$

Semantica denotațională: expresii

$$State = Id \rightharpoonup \mathbb{Z}$$

- Domenii semantice:
  $[[\_]] : AExp \to (State \rightharpoonup \mathbb{Z})$
  $[[\_]] : BExp \to (State \rightharpoonup \{T, F\})$
  $[[\_]] : Stmt \to (State \rightharpoonup State)$
- Semantica denotaţională este compoziţională:
  - semantica expresiilor aritmetice
    $[[n]](s) = n$
    $[[x]](s) = s(x)$
    $[[e1 + e2]](s) = [[e1]](s) + [[e2]](s)$
  - semantica expresiilor booleene
    $[[true]](s) = T$, $[[false]](s) = F$
    $[[!b]](s) = \neg b$
    $[[e1 <= e2]](s) = [[e1]](s) <= [[e2]](s)$

Semantica denotaţională: instrucţiuni

$$State = Id \rightharpoonup \mathbb{Z}$$

- Domenii semantice:
  $[[\_]] : AExp \to (State \rightharpoonup \mathbb{Z})$
  $[[\_]] : BExp \to (State \rightharpoonup \{T, F\})$
  $[[\_]] : Stmt \to (State \rightharpoonup State)$

Semantica denotaţională: instrucţiuni

$$State = Id \rightharpoonup \mathbb{Z}$$

- Domenii semantice:
  $[[\_]] : AExp \to (State \rightharpoonup \mathbb{Z})$
  $[[\_]] : BExp \to (State \rightharpoonup \{T, F\})$
  $[[\_]] : Stmt \to (State \rightharpoonup State)$
- Semantica instrucţiunilor:

$$[[\texttt{skip}]] = id$$

$$[[\texttt{c1;c2}]] = [[c2]] \circ [[c1]]$$

$$[[\texttt{x = e}]](s)(y) = \begin{cases} s(y), \text{ dacă } y \neq x \\ [[e]](s), \text{ dacă } y = x \end{cases}$$

Semantica denotaţională: instrucţiuni

$$State = Id \rightharpoonup \mathbb{Z}$$

- Domenii semantice:
  $[[\_]] : AExp \to (State \rightharpoonup \mathbb{Z})$
  $[[\_]] : BExp \to (State \rightharpoonup \{T, F\})$
  $[[\_]] : Stmt \to (State \rightharpoonup State)$
- Semantica instrucţiunilor:

$$[[\texttt{skip}]] = id$$

$$[[\texttt{c1;c2}]] = [[c2]] \circ [[c1]]$$

$$[[\texttt{x = e}]](s)(y) = \begin{cases} s(y), \text{ dacă } y \neq x \\ [[e]](s), \text{ dacă } y = x \end{cases}$$

$$[[\texttt{if (b) c1 else c2}]](s) = \begin{cases} [[c1]](s), \text{ dacă } [[b]](s) = T \\ [[c2]](s), \text{ dacă } [[b]](s) = F \end{cases}$$

Semantica denotațională

Exemplu

```
if (x<= y) z=x; else z=y;
```

$$[[pgm]](s) = \begin{cases} [[z = x;]](s), \text{dacă } [[x <= y]](s) = T \\ [[z = y;]](s), \text{dacă } [[x <= y]](s) = F \end{cases}$$

Semantica denotațională

Exemplu

```
if (x<= y) z=x; else z=y;
```

$$[[pgm]](s) = \begin{cases} [[z = x;]](s), \text{dacă } [[x <= y]](s) = T \\ [[z = y;]](s), \text{dacă } [[x <= y]](s) = F \end{cases}$$

$$[[pgm]](s)(v) = \begin{cases} s(v), \text{dacă } s(x) \leq s(y), v \neq z \\ s(x), \text{dacă } s(x) \leq s(y), v = z \\ s(v), \text{dacă } s(x) > s(y), v \neq z \\ s(y), \text{dacă } s(x) > s(y), v = z \end{cases}$$

Semantica denotațională

Exemplu

```
if (x<= y) z=x; else z=y;
```

$$[[pgm]](s) = \begin{cases} [[z = x;]](s), \text{ dacă } [[x <= y]](s) = T \\ [[z = y;]](s), \text{ dacă } [[x <= y]](s) = F \end{cases}$$

$$[[pgm]](s)(v) = \begin{cases} s(v), \text{ dacă } s(x) \leq s(y), v \neq z \\ s(x), \text{ dacă } s(x) \leq s(y), v = z \\ s(v), \text{ dacă } s(x) > s(y), v \neq z \\ s(y), \text{ dacă } s(x) > s(y), v = z \end{cases}$$

Cum definim semantica denotaţională pentru `while`?

Mulţimea funcţiilor parţiale

Fie X şi Y două mulţimi.

- $Pfn(X, Y)$ mulţimea funcţiilor parţiale de la X la Y, adică
  $Pfn(X, Y) = X \rightharpoonup Y$
- Pentru $f \in Pfn(X, Y)$ notăm cu $dom(f)$ mulţimea elementelor din X pentru care funcţia este definită.
  Atunci $dom(f) \subseteq X$ şi $f|_{dom(f)} : dom(f) \to Y$ este funcţie.

Mulţimea funcţiilor parţiale

Fie X şi Y două mulţimi.

- $Pfn(X, Y)$ mulţimea funcţiilor parţiale de la X la Y, adică $Pfn(X, Y) = X \rightharpoonup Y$
- Pentru $f \in Pfn(X, Y)$ notăm cu $dom(f)$ mulţimea elementelor din X pentru care funcţia este definită.
  Atunci $dom(f) \subseteq X$ şi $f|_{dom(f)} : dom(f) \to Y$ este funcţie.
- Fie $\perp : X \rightharpoonup Y$ unica funcţie cu $dom(\perp) = \emptyset$ (funcţia care nu este definită în nici un punct).
- Definim pe $Pfn(X, Y)$ următoarea relaţie:

$$f \sqsubseteq g \text{ dacă şi numai dacă } dom(f) \subseteq dom(g) \text{ şi } g|_{dom(f)} = f_{dom(f)}$$

Mulţimea funcţiilor parţiale

Fie X şi Y două mulţimi.

- $Pfn(X, Y)$ mulţimea funcţiilor parţiale de la X la Y, adică $Pfn(X, Y) = X \rightharpoonup Y$
- Pentru $f \in Pfn(X, Y)$ notăm cu $dom(f)$ mulţimea elementelor din X pentru care funcţia este definită.
  Atunci $dom(f) \subseteq X$ şi $f|_{dom(f)} : dom(f) \to Y$ este funcţie.
- Fie $\perp : X \rightharpoonup Y$ unica funcţie cu $dom(\perp) = \emptyset$ (funcţia care nu este definită în nici un punct).
- Definim pe $Pfn(X, Y)$ următoarea relaţie:

$$f \sqsubseteq g \text{ dacă şi numai dacă } dom(f) \subseteq dom(g) \text{ şi } g|_{dom(f)} = f_{dom(f)}$$

$(Pfn(X, Y), \sqsubseteq, \perp)$ este CPO

(mulţime parţial ordonată completă în care $\perp$ este cel mai mic element)

$(Pfn(X, Y), \sqsubseteq, \perp)$ este CPO

Exemplu

Definim $F : Pfn(\mathbb{N}, \mathbb{N}) \to Pfn(\mathbb{N}, \mathbb{N})$ prin

$$F(g)(k) = \begin{cases} 1, & \text{dacă } k = 0, \\ k * g(k-1) & \text{dacă } k > 0 \text{ şi } (k-1) \in dom(g), \\ \text{nedefinit}, & \text{altfel} \end{cases}$$

- F este o funcţie continuă,

$(Pfn(X, Y), \sqsubseteq, \bot)$ este CPO

Exemplu

Definim $F : Pfn(\mathbb{N}, \mathbb{N}) \to Pfn(\mathbb{N}, \mathbb{N})$ prin

$$F(g)(k) = \begin{cases} 1, & \text{dacă } k = 0, \\ k * g(k-1) & \text{dacă } k > 0 \text{ şi } (k-1) \in dom(g), \\ \text{nedefinit}, & \text{altfel} \end{cases}$$

- F este o funcţie continuă,deci putem aplica
- Teorema Knaster-Tarski
  Fie $g_n = F^n(\bot)$ şi $f = \bigvee_n g_n$.
  Ştim că f este cel mai mic punct fix al funcţiei F, deci $F(f) = f$.

$(Pfn(X, Y), \sqsubseteq, \perp)$ este CPO

Exemplu

Definim $F : Pfn(\mathbb{N}, \mathbb{N}) \to Pfn(\mathbb{N}, \mathbb{N})$ prin

$$F(g)(k) = \begin{cases} 1, & \text{dacă } k = 0, \\ k * g(k-1) & \text{dacă } k > 0 \text{ şi } (k-1) \in dom(g), \\ \text{nedefinit}, & \text{altfel} \end{cases}$$

- F este o funcţie continuă,deci putem aplica
- Teorema Knaster-Tarski
  Fie $g_n = F^n(\perp)$ şi $f = \bigvee_n g_n$.
  Ştim că f este cel mai mic punct fix al funcţiei F, deci $F(f) = f$.
- Demonstrăm prin inducţie după n că:

  $dom(g_n) = \{0, \ldots, n\}$ şi $g_n(k) = k!$ oricare $k \in dom(g_n)$
- $f : \mathbb{N} \to \mathbb{N}$ este funcţia factorial.

Semantica denotaţională pentru `while`

`while (b) c`

- Definim $F : Pfn(State, State) \to Pfn(State, State)$ prin
- F este continuă
- Teorema Knaster-Tarski: $fix(F) = \bigcup_n F^n(\bot)$

Semantica denotaţională pentru `while`

`while (b) c`

- Definim $F : Pfn(State, State) \to Pfn(State, State)$ prin

$$F(g)(s) = \begin{cases} g([[c]](s)) & \text{dacă } [[b]](s) = T \\ s & \text{dacă } [[b]](s) = F \\ \text{nedefinit,} & \text{altfel} \end{cases}$$

- F este continuă
- Teorema Knaster-Tarski: $fix(F) = \bigcup_n F^n(\bot)$

Semantica denotaţională pentru `while`

`while (b) c`

- Definim $F : Pfn(State, State) \to Pfn(State, State)$ prin

$$F(g)(s) = \begin{cases} g([[c]](s)) & \text{dacă } [[b]](s) = T \\ s & \text{dacă } [[b]](s) = F \\ \text{nedefinit,} & \text{altfel} \end{cases}$$

- F este continuă
- Teorema Knaster-Tarski: $fix(F) = \bigcup_n F^n(\bot)$
- Semantica denotaţională:

$$[[\_]] : Stmt \to (State \rightharpoonup State)$$

$$[[\texttt{while (b) c}]](s) = fix(F)(s)$$

Avantaje şi dezavantaje

Semantica operațională

- \+ Defineşte precis noțiunea de pas computațional
- \+ Semnalează erorile, oprind execuția
- \+ Execuția devine uşor de urmărit şi depanat
- – Regulile structurale sunt evidente şi deci plictisitor de scris
- – Nemodular: adăugarea unei trăsături noi poate solicita schimbarea întregii definiții

Semantica denotațională

- \+ Formală, matematică, foarte precisă
- \+ Compoziţională (morfisme şi compuneri de funcţii)
- – Domeniile devin din ce în ce mai complexe.

Pe săptămâna viitoare!