

Limbaje și sisteme de tipuri

Note de curs și seminar

Eneia Nicolae Todoran

-2010-

Cuprins

I	Introducere	1
1	Concepte de bază	2
2	Semantică operațională	5
2.1	Sisteme de tranziție	6
2.2	Semantică operațională în Prolog	9
2.3	Semantică operațională în Haskell	11
3	Semantică denotațională	12
3.1	Principiile semanticii denotaționale	13
3.2	Domenii semantice	14
3.3	Semantică denotațională în Haskell	18
II	Semantică operațională	21
4	Secvențiere și recursivitate	22
4.1	Limbaajul \bar{L}_1 : sintaxă și sistem de tranziție	23
4.2	Exerciții	27
4.3	Funcția semantică operațională	32
4.4	Prototip Prolog	33
4.5	Prototip Haskell	36
4.6	Exerciții	39
5	Semantică de continuare pentru \bar{L}_1	42
5.1	Prototip Prolog	45
5.2	Prototip Haskell	48
5.3	Exerciții	52
6	Nedeterminism	54
6.1	Limbaajul \bar{L}_2	57
6.2	Prototip Prolog	58
6.3	Prototip Haskell	67
6.4	Exerciții	73

7	Semantică de continuare pentru \bar{L}_2	76
7.1	Prototip Prolog	77
7.2	Prototip Haskell	81
7.3	Exerciții	85
8	Concurență și nedeterminism	87
8.1	Limbajul \bar{L}_3	88
8.2	Prototip Prolog	89
8.3	Prototip Haskell	95
8.4	Exerciții	100
9	Semantică de continuare pentru concurență	102
9.1	Conceptul de multiset	103
9.2	Limbajul \bar{L}_4	107
9.3	Continuări și configurații pentru \bar{L}_4	108
9.4	Sistemul de tranziție TSS_{4csc} pentru \bar{L}_4	110
9.5	Exerciții	113
9.6	Prototip Prolog	115
9.7	Prototip Haskell	120
III	Semantică denotațională	124
10	Modele semantice pentru un limbaj uniform	125
10.1	Semantică directă	126
10.2	Semantică de continuare	130
10.3	Continuări structurate - tehnica CSC	134
10.3.1	Domenii semantice pentru CSC	138
10.3.2	Mecanismul de evaluare a continuărilor CSC	141
11	Semantica construcțiilor imperative	148
11.1	Conceptul de <i>stare</i>	150
11.1.1	Implementare ca funcție	151
11.1.2	Implementare ca listă de asociație	152
11.2	Semantica expresiilor	153
11.3	Transformări de stare	154
11.3.1	Semantică directă	155
11.3.2	Semantică de continuare	156
11.4	Istории de calcul	158
11.4.1	Semantică de continuare clasică	159
11.4.2	Semantică CSC	160
11.4.3	Istории de valori observabile	161

12	Recursivitate și semantică de punct fix	164
12.1	Notăția letrec	165
12.2	Medii semantice și construcții de punct fix	168
12.3	Notăția μ	173
12.4	Semantică de punct fix în Haskell	176
12.4.1	Semantică directă pentru L_1^{rec} și L_1^μ	179
12.4.2	Semantică CSC pentru L_1^{rec} și L_1^μ	183
13	Semantică de continuare pentru concurență	188
13.1	Compunere paralelă	192
13.1.1	Specificare	194
13.1.2	Prototip	203
13.2	Comunicare sincronă	209
13.2.1	Specificare	217
13.2.2	Prototip	218
IV	Semantica construcțiilor aplicative. Verificare tipuri.	221
14	Un limbaj de tip PCF	222
14.1	Sintaxa L_{pcf}	223
14.2	Verificarea tipurilor	225
14.3	Semantică denotațională	232
A	Definiții pentru starea ca listă de asociație	247
B	Definiții auxiliare pentru secțiunea 13.1	248
C	Definiții auxiliare pentru secțiunea 13.2	249
D	Definiții auxiliare pentru capitolul 14	250

Partea I

Introducere

Capitolul 1

Concepte de bază

- Aceste note de curs și seminar:
 - oferă metode precise de descriere și proiectare a limbajelor de programare și specificare;
 - prezintă concepte și principii ce stau la baza limbajelor imperative, funcționale, orientate pe obiecte și concurente;
 - evidențiază importanța semanticii formale: semantică dinamică (operațională și denotațională) și semantică statică (verificare tipuri);
 - prezintă metode și tehnici generale de proiectare a limbajelor;
 - validează metodele și tehnicile de proiectare semantică prin construirea de interpretoare prototip (executabile)

- Sunt utilizate două metode (discipline) consacrate în proiectarea semantică:
 - **semantica operațională**
 - * oferă o descriere intuitivă a modului de execuție a programelor
 - * are la bază conceptul de *sistem de tranziție*
 - **semantica denotațională**
 - * construcțiile program *denotă* valori dintr-un domeniu matematic de interpretare
 - * se operează cu *definiții compoziționale*
- În proiectarea mașinilor abstracte este utilizată *semantica operațională structurată*.
- Sunt prezentate tehnici semantice: stări, continuări clasice, continuări pentru concurență.
- Sunt oferite exemple de proiectare și verificare matematică a modelelor semantice
- Sunt studiate concepte aplicative ce au la bază calculul lambda simplu tipizat (în stil PCF).
- Sunt prezentate aplicații complexe: interpretoare semantice pentru limbaje imperative, aplicative, orientate pe obiecte și concurente.

- Precizări și note bibliografice:
 - Referințele de bază pentru semantica operațională structurată sunt [25, 10].
 - Materialul legat de semantica operațională se bazează pe [4]. Modelele semantice din [4] sunt adaptate prin construirea de interpretoare (prototipuri declarative) Prolog și Haskell
 - Principiile semanticii denotaționale au fost introduse la sfârșitul anilor '60 și începutul anilor '70 [33, 34, 28, 30].
 - În majoritatea aplicațiilor, domeniile matematice utilizate în semantica denotațională sunt mulțimi parțial ordonate [9, 24] sau spații metrice [4].
 - Materialul despre calculul lambda simplu tipizat și PCF este din [36, 8].
 - Tehnica clasică a semanticii de continuare a fost introdusă ca instrument pentru semantica denotațională în [35].
 - Semantica de continuare pentru concurență a fost introdusă în [38, 39].
 - În cursurile avansate de limbaje se utilizează adesea următoarele monografii [36, 8, 18, 4, 21, 7, 43].

Capitolul 2

Semantică operațională

- În semantica operațională execuția unui program este modelată sub forma unei secvențe de configurații

$$c_1, c_2, \dots, c_n, \dots$$

stabilită pe baza unei relații de tranziție ' \rightarrow ' între configurații astfel încât $(c_i, c_{i+1}) \in \rightarrow$, fapt exprimat mai sugestiv în forma: $c_i \rightarrow c_{i+1}$.

- De exemplu, o configurație poate fi o instrucțiune program, sau o pereche constând dintr-o instrucțiune program împreună cu starea curentă.
- Adesea, se operează cu tranziții etichetate de forma:
 $c_i \xrightarrow{a_i} c_{i+1}$.
- Eticheta de tranziție ar putea fi o acțiune atomică sau starea curentă.

2.1 Sisteme de tranziție

- Începând cu [25, 10], abordarea preferată în majoritatea aplicațiilor este *semantica operațională structurată* (SOS).
- SOS are la bază conceptul de *sistem de tranziție*.
 - Introducem mașinăria semanticii operaționale direct pentru cazul sistemelor de tranziție etichetate (engl. *labeled transition system*) LTS.
 - Sistemele neetichetate se obțin cu ușurință prin specializare.
- Un LTS este un triplet (C, A, \rightarrow) constând dintr-o mulțime¹ $(c \in)C$ de *configurații*, o mulțime $(a \in)A$ de *etichete de tranziție* și o *relație de tranziție* $\rightarrow \subseteq C \times A \times C$.
 - În acest caz faptul că tripletul (c, a, c') este un element al relației de tranziție $((c, a, c') \in \rightarrow)$ se exprimă de obicei sub forma $c \xrightarrow{a} c'$.

¹În această lucrare notația $(x, y, \dots \in)X$ introduce mulțimea X cu elemente tipice x, y, \dots (care iau valori din X).

- În abordarea SOS relația de tranziție ' \rightarrow ' este inclusă de o *specificare de sistem de tranziție* (engl. *transition system specification*) TSS, care este o colecție de *reguli* de forma:

$$\frac{c_1 \xrightarrow{a_1} c'_1 \quad \cdots \quad c_n \xrightarrow{a_n} c'_n}{c \xrightarrow{a} c'}$$

Elementele $c_i \xrightarrow{a_i} c'_i$ se numesc *premise*, iar $c \xrightarrow{a} c'$ se numește *concluzia* regulii. Dacă $n = 0$ (adică nu există premise) regula se numește *axiomă*.

- Tranzițiile date sub formă de axiome au loc prin definiție.
 - În plus, orice tranziție care este concluzia unei reguli este un element al relației de tranziție dacă se poate stabili în baza regulilor TSS că premisele sale sunt adevărate.
- Relația de tranziție specificată printr-un TSS este *cea mai mică* relație care satisface regulile TSS.

• Observații

– În unele aplicații nu sunt necesare tranziții etichetate.

În acest caz

* relația de tranziție ' \rightarrow ' este o submulțime a $C \times C$ ($\rightarrow \subseteq (C \times C)$),

* iar regulile de deducție sunt de forma:

$$\frac{c_1 \rightarrow c'_1 \quad \cdots \quad c_n \rightarrow c'_n}{c \rightarrow c'}$$

– În majoritatea aplicațiilor o configurație este

* fie un element al limbajului (o construcție sintactică reprezentând instrucțiunea curentă a mașinii abstracte de calcul),

* fie o pereche constând dintr-un element al limbajului împreună cu starea curentă a mașinii.

– Termenul *structurat* din SOS se referă la proiectarea ghidată de sintaxă a regulilor TSS.

• Termenul *semantică* denotă o funcție \mathcal{S} de forma:

$$\mathcal{S} : L \rightarrow \mathbb{D}$$

unde L este mulțimea construcțiilor limbajului dat, iar \mathbb{D} este un domeniu matematic de interpretare. Funcția semantică operațională se definește direct pe baza un sistem de tranziție pentru limbajul considerat, așa cum se va vedea din exemple

2.2 Semantică operațională în Prolog

- Un sistem de tranziție este un sistem de deducție alcătuit din axiome și reguli ce permit specificarea unei relații de tranziție. Deducerea tranzițiilor se face, la fel ca în limbajul Prolog, prin înlănțuire înapoi (engl. *backward chaining*).
- Este deci naturală transcrierea unui sistem de tranziție ca program logic sub forma unui set de clauze Horn.
 - Fiecare axiomă TSS poate fi implementată sub forma unui *fapt* în Prolog.
 - Fiecare regulă TSS poate fi implementată sub forma unei *reguli* Prolog corespunzătoare.

De exemplu, o regulă:

$$\frac{c_1 \xrightarrow{a_1} c'_1 \quad \dots \quad c_n \xrightarrow{a_n} c'_n}{c \xrightarrow{a} c'}$$

poate fi implementată în Prolog sub forma unei clauze a unui predicat **tss/3**:

```
tss(C,A,C') :-
    tss(C1,A1,C1'), ..., tss(Cn,An,Cn').
```

- În aceste note de curs și seminar este utilizată o notăție de tip Turbo Prolog (sau Visual Prolog), ce include o secțiune pentru declarații de tipuri.
- Un program include patru secțiuni distincte, delimitate prin cuvintele cheie: **domains** (declarații de tip), **predicates** (declarații predicate cu tipul argumentelor), **clauses** (fapte și reguli pentru predicatele Prolog) și **goal**.
- Declarațiile de tip permit evidențierea domeniilor sintactice utilizate în specificarea unui sistem de tranziție și specificarea structurii matematice a universului semantic corespunzător.

2.3 Semantică operațională în Haskell

- Limbajul Haskell [20] este un limbaj (pur) funcțional. Pentru facilitațiile de calcul simbolic și funcții de ordin superior este adesea utilizat ca metalimbaj în proiectarea semantică, în special în proiectarea denotațională.
- În aceste note de curs și seminar vom oferi și implementări Haskell ale semanticilor operaționale.
- Față de Prolog, limbajul Haskell are dezavantajul că exprimă cu mai multă dificultate un sistem de deducție utilizat în specificarea unui sistem de tranziție.
 - În particular, colecția tuturor tranzițiilor unui sistem nedeterminist va trebui returnată în Haskell ca o lista, în timp ce în Prolog se poate utiliza în mod elegant mecanismul implicit de backtracking, ce permite o exprimare naturală a nedeterminismului.
- Totuși, prin manipularea funcțiilor de ordin superior limbajul Haskell poate oferi avantaje în prototipizarea semantică, știut fiind că modelele semantice utilizează adesea funcții de ordin superior.

Capitolul 3

Semantică denotațională

- Prin definirea unei semantici denotaționale pentru un limbaj de programare se atașează valori matematice fiecărei construcții program și astfel se obține o definiție precisă a limbajului de programare.
- O asemenea definiție permite verificarea corectitudinii diverselor implementări ale limbajului și poate fi utilizată în demonstrarea proprietăților programelor.
- În aceste note de curs și seminar termenul *semantică* denotă o funcție \mathcal{S} de forma:

$$\mathcal{S} : L \rightarrow \mathbb{D}$$

unde L este mulțimea construcțiilor limbajului dat, iar \mathbb{D} este un domeniu matematic de interpretare.

- În reprezentarea acestor funcții se utilizează adesea 'parantezele semantice': $\llbracket \cdot \rrbracket : L \rightarrow \mathbb{D}$.

3.1 Principiile semanticii denotaționale

1. *Construcțiile program denotă valori dintr-un domeniu matematic de interpretare.*
2. *Definițiile denotaționale sunt compoziționale: semantica unei construcții sintactice compuse este exprimată ca funcție de semantica construcțiilor sintactice componente.*

$$\llbracket \cdots s_1 \cdots s_n \cdots \rrbracket = \cdots \llbracket s_1 \rrbracket \cdots \llbracket s_n \rrbracket \cdots$$

Realizarea unei semantici denotaționale cuprinde următoarele activități:

- definirea unui domeniu matematic de interpretare;
- proiectarea unei funcții compoziționale care mapează construcțiile sintactice la valori din domeniul de interpretare.

3.2 Domenii semantice

- Un *domeniu* este o structură matematică care descrie noțiunea de calcul și aproximare [32].
 - Un calcul efectuat pe baza unui algoritm se realizează în pași discreți.
 - După fiecare pas se obține mai multă informație despre rezultatul calculului.
 - Rezultatul obținut după fiecare pas poate fi privit drept o aproximare a rezultatului final.
- În semantică, domeniile sunt descrise prin structuri matematice.
- Intuitiv, un domeniu este o mulțime echipată cu o anumită *structură* (o relație de ordine parțială sau o metrică).
- Domeniile compuse sunt obținute prin utilizarea câtorva construcții de bază: produsul cartezian, reuniunea (disjunctă), spațiul de funcții, domeniile putere (mulțimi de submulțimi).

- Domeniile semantice trebuie să permită reprezentarea proceselor de calcul perpetue; acest tip de comportament apare în prezența recursivității.
- În abordarea clasică, domeniile sunt mulțimi parțial ordonate complete și se operează cu funcții continue [24, 36, 18].
- O altă abordare cunoscută utilizează spații metrice complete și contracții peste spații metrice [4].
- Comportamentul perpetuu (infini) se obține ca limită a unor procese de calcul finite.

- Domeniile se definesc prin *ecuații de domeniu* (în general recursive) de forma:

$$\mathbb{X} = E$$

unde E este o expresie compusă dată în notație BNF printr-o gramatică de forma:

$$E ::= \mathbb{A} \mid \mathbb{X} \mid \mathcal{P}(E) \mid \\ E \times E \mid E + E \mid E \rightarrow E$$

în care \mathbb{A} este un domeniu arbitrar dar constant (care nu depinde de \mathbb{X}) și sunt utilizate următoarele construcții:

– spațiu de funcții:

$$E_1 \rightarrow E_2$$

– domeniu putere (mulțimea submulțimilor):

$$\mathcal{P}(E)$$

– produs cartezian:

$$E_1 \times E_2$$

– reuniune disjunctă (sumă):

$$E_1 + E_2 \quad (= (\{1\} \times E_1) \cup (\{2\} \times E_2))$$

- De exemplu, un domeniu de liste finite sau infinite de numere naturale poate fi definit astfel:

$$\mathbb{L} = \{Nil\} + (\mathbb{N} \times \mathbb{L})$$

Câteva elemente din \mathbb{L} sunt date mai jos:

Nil

$(1, (2, (3, Nil)))$

$(1, (1, (1, \dots)))$

- Metoda consacrată de rezolvare a ecuațiilor de domeniu se bazează pe teoria categoriilor [15], atât în cazul domeniilor clasice [31] cât și în cazul spațiilor metrice [3].

3.3 Semantică denotațională în Haskell

- În cadrul acestor note de curs și seminar, în locul notației matematice este utilizat adesea limbajul Haskell [20] pentru exprimarea modelelor denotaționale.
 - Semantica denotațională este prin natura sa o semantică funcțională în care se operează cu definiții compoziționale.
 - Limbajul Haskell oferă în mod natural suport pentru implementarea tehnicilor majore utilizate în semantica denotațională: stări, medii semantice, continuări, etc.
- Se au în vedere următoarele beneficii:
 - se evită dificultățile legate de prezentarea sistematică a aparatului matematic utilizat în teoria clasică a domeniilor sau în semantica metrică;
 - modelele denotaționale devin interpretoare prototip pentru limbajele considerate și pot fi imediat evaluate și testate.

- Domeniile semantice pot fi exprimate în limbajul Haskell astfel:
 - produsul cartezian prin tuple;
 - reuniunea (disjunctă) prin reuniuni (sume);
 - spațiile de funcții prin constructorul `->`;
 - domeniile putere prin constructorul `[]`.
- De exemplu, dacă **D** și **E** sunt tipuri atunci se pot introduce tipuri noi astfel:

```
type ProdusCartezian    = (D,E)
data ReuniuneDisjuncta = D D | E E
type SpatiuFuncții     = D -> E
type DomeniuPutere     = [D]
```


- Sintaxa limbajelor studiate este dată prin definiții de forma:

```
data X = ...
      | OpSin X X
      ...
```

- Semantica este definită prin funcții compoziționale

```
sem :: X -> D
...
sem (OpSin x1 x2) =
  (sem x1) 'opSem' (sem x2)
...
```

unde

```
opSem :: D -> D -> D
```

Partea II

Semantică operațională

Capitolul 4

Secvențiere și recursivitate

- În acest capitol este studiat un limbaj foarte simplu \bar{L}_1 , care aproximează nucleul de fluentă al limbajelor secvențiale.
- Prezentăm sintaxa și semantica operațională a \bar{L}_1 , și introducem anumite tehnici prin care se poate raționa asupra comportamentului programelor și a proprietăților limbajului \bar{L}_1 .
- Oferim o implementare Prolog a semanticii operaționale
- Oferim o implementare Haskell a semanticii operaționale

4.1 Limbajul \overline{L}_1 : sintaxă și sistem de tranziție

- Considerăm date
 - o mulțime $(a \in) Act$ de *acțiuni atomice*. Aici A este un alfabet de simboluri elementare, neinterpretate. Aceste simboluri sunt reprezentări abstracte ale unor elemente de bază din limbajele de programare, cum sunt instrucțiunile de atribuire, sau expresiile;
 - * Limbajele care abstractizează acțiunile atomice sub formă de simboluri atomice dintr-un alfabet dat se zic *uniforme* [4]
 - o mulțime $(y \in) Y$ de *variabile de procedură*; intuitiv, apariția unei variabile de procedură în textul unui program \overline{L}_1 este un apel recursiv. Aceste mulțimi vor fi utilizate și în capitolele următoare.
- Recursivitatea este tratată pe baza conceptelor de *declarație și instrucțiune gardată*, urmând abordarea din [4]
- Intuitiv, într-o instrucțiune gardată orice apel recursiv se poate realiza doar după execuția a cel puțin o instrucțiune atomică.

• **Sintaxa \bar{L}_1**

(a) Instrucțiuni:

$$x(\in X) ::= a \mid y \mid x; x$$

(b) Instrucțiuni gardate:

$$g(\in G) ::= a \mid g; x$$

(c) Declarații:

$$(D \in) Decl = Y \rightarrow G$$

(d) Programe:

$$(\pi \in) \bar{L}_1 = Decl \times X$$

• **Configurațiile TSS_1 pentru \bar{L}_1 ¹**

$$r(\in Conf) ::= E \mid x$$

(adică $Conf = \{E\} \cup X$)

¹Reamintim că folosim abrevierea TSS pentru a ne referi la o specificare de sistem de tranziție, în engleză *transition system specification (TSS)*.

• **Sistemul de tranziție** TSS_1 **pentru** \bar{L}_1 [4]

Relația de tranziție $\rightarrow \subseteq Conf \times Act \times Decl \times Conf$ cu elemente (r, a, D, r') scrise în notația $r \xrightarrow{a}_D r'$ este cea mai mică relație care satisface următorul set de axiome și reguli. În particular, asta înseamnă că pentru configurația E nu există nici o tranziție.

(Act)

$$a \xrightarrow{a}_D E$$

(Rec)

$$\frac{D(y) \xrightarrow{a}_D r}{y \xrightarrow{a}_D r}$$

(Seq)

$$\frac{x_1 \xrightarrow{a}_D E}{x_1; x_2 \xrightarrow{a}_D x_2}$$

$$\frac{x_1 \xrightarrow{a}_D x'_1}{x_1; x_2 \xrightarrow{a}_D x'_1; x_2}$$

• **Mulțime succesori și complexitate [4]**

- Pentru orice sistem de tranziție etichetat TSS, dat ca un triplet $(Conf, Act, \rightarrow)$, și pentru orice $c \in Conf$, se definește *mulțimea succesori* $S(c)$ pentru configurația c astfel:

$$S(c) = \{(a, c') \mid c \xrightarrow{a} c'\}$$

- Pentru raționamente inductive se utilizează o *măsură de complexitate* care scade la apeluri recursive, $c : (Decl \times Conf) \rightarrow \mathbb{N}$

$$c(D, E) = 0$$

$$c(D, a) = 1$$

$$c(D, y) = 1 + c(D, D(y))$$

$$c(D, x_1; x_2) = 1 + c(D, x_1)$$

4.2 Exerciții

E 4.2.1 Fie $D \in Decl$, astfel încât $D(y_1) = a_1$, $D(y_2) = a_2$. Utilizând regulile sistemului de tranziție TSS_1 să se deducă (tranziția sau) tranzițiile următoarelor programe $\bar{L}_1: (y_1; a_2)$ și $((y_2; a_3); a_4)$.

E 4.2.2 Să se arate că funcția (măsura de complexitate) c este bine definită.

- Trebuie demonstrat că pentru orice $(D, r) \in Decl \times Conf$, $c(D, r) \in \mathbb{N}$.
- **Sugestie:** Evident, $c(D, E) \in \mathbb{N}$, pt. orice $D \in Decl$. Pentru $r \in X$ se poate proceda în doi pași. Se arată mai întâi că $c(D, g) \in \mathbb{N}$, prin inducție structurală² pentru orice instrucțiune gardată $g \in G$; de remarcat că $G \subseteq X$. Apoi se tratează cazul general, și se arată că $c(D, x) \in \mathbb{N}$, tot prin inducție structurală, după structura lui x .

²Adică prin inducție după structura sintactică a lui g .

E 4.2.3 *Să se arate că sistemul de tranziție TSS_1 este determinist. Formal, au loc următoarele, pentru $(D, r) \in Decl \times Conf$:*³

$$(a) |S(D, E)| = 0$$

$$(b) |S(D, x)| = 1, \quad \forall x \in X$$

– **Sugestie:** *Se poate proceda prin inducție după $c(D, r)$.*

³În această lucrare prin $|M|$ notăm **cardinalul** mulțimii M .

E 4.2.4 *Se consideră un limbaj neuniform \bar{L}'_1 (instrucțiunile elementare sunt atribuiri de valori de expresii la variabile) cu sintaxa:*

(a) *Instrucțiuni:*

$$x(\in X) ::= v := n \mid y \mid x; x$$

(b) *Instrucțiuni gardate:*

$$g(\in G) ::= v := n \mid g; x$$

(c) *Declarații:*

$$(D \in) Decl = Y \rightarrow G$$

(d) *Programe:*

$$(\pi \in) \bar{L}'_1 = Decl \times X$$

Construcțiile pentru recursivitate și secvențiere sunt o adaptare a conceptelor introduse în contextul limbajului \bar{L}_1 . $n \in N$ este o clasă dată de expresii numerice de forma:

$$n ::= z \mid v \mid n + n \mid n - n$$

($v \in$) V este o clasă de variabile simple ce pot stoca valori numerice. Pentru simplitate se poate lucra cu numere întregi ($\in \mathbb{Z}$), și se poate considera că $z \in \mathbb{Z}$.

Se presupune că semantica expresiilor este definită printr-o funcție de evaluare dată

$$\mathcal{N}[\cdot] : N \rightarrow \Sigma \rightarrow \mathbb{Z},$$

$$\mathcal{N}[z](\sigma) = z$$

$$\mathcal{N}[v](\sigma) = \sigma(v)$$

$$\mathcal{N}[n_1 + n_2](\sigma) = \mathcal{N}[n_1](\sigma) + \mathcal{N}[n_2](\sigma)$$

...

$(\sigma \in)\Sigma$ este o mulțime de stări ale mașinii, modelate ca funcții de la variabile la valori:

$$(\sigma \in)\Sigma = V \rightarrow \mathbb{Z}$$

Pentru modelarea semanticii instrucțiunii de atribuire se poate utiliza notația $(\sigma \mid v \mapsto z)$.

Dacă $f : X \rightarrow Y$ este o funcție de la mulțimea $(x \in)X$ la mulțimea $(y \in)Y$, atunci

$(f \mid x \mapsto y) : X \rightarrow Y$ este o funcție (cu același domeniu și codomeniu) definită astfel:

$$(f \mid x \mapsto y)(x') = \begin{cases} y & \text{if } x' = x \\ f(x') & \text{if } x' \neq x \end{cases}$$

Intuitiv, $(f \mid x \mapsto y)$ este o 'perturbare' a funcției f (doar) în punctul x .

Se cere:

- Să se proiecteze un sistem de tranziție $TSS_{1'}$ care să specifice comportamentul instrucțiunilor pentru \bar{L}_1 . Configurațiile $TSS_{1'}$ vor fi de forma $(r, \sigma) \in (Conf \times \Sigma)$, unde

$$r ::= E \mid x \quad (Conf = \{E\} \cup X)$$

- Să se arate că $TSS_{1'}$ este determinist. Raționamentul se va efectua prin inducție după o măsură de complexitate adecvată, $c : Conf \rightarrow \mathbb{N}$,
 $c(E) = 0$, $c(v := n) = 1$, $c(y) = 1 + c(D(y))$
 și $c(x_1; x_2) = 1 + c(x_1)$.

4.3 Funcția semantică operațională

- **Univers semantic**

$$\mathbb{P}_O = Act^\infty$$

unde

$$Act^\infty = Act^* \cup Act^\omega$$

adică Act^∞ conține toate secvențele finite (Act^*) și infinite (Act^ω) de elemente din Act . Notăm prin ϵ *secvența vidă* (aici $\epsilon \in Act^\infty$).

- **Semantică operațională pentru \bar{L}_1 [4]**

$\mathcal{O}[\cdot] : \bar{L}_1 \rightarrow \mathbb{P}_O$, $\mathcal{O}[(D, x)] = \mathcal{O}_D(x)$, unde funcția $\mathcal{O}_D : Conf \rightarrow \mathbb{P}_O$ este definită astfel:

$$\begin{aligned} \mathcal{O}_D(E) &= \epsilon \\ \mathcal{O}_D(x) &= a \cdot \mathcal{O}_D(r) \quad \text{if } x \xrightarrow{a}_D r \end{aligned}$$

'.' este utilizat în această lucrare ca operator de concatenare secvențe (aici '.' prefixează un element la o secvență, ceea ce este echivalent cu a concatena o secvență de lungime 1 cu o altă secvență).

4.4 Prototip Prolog

- Se utilizează o notație de tip Turbo Prolog sau Visual Prolog pentru reprezentarea domeniilor

domains

```
a = symbol
y = symbol
x = a(a); y(y); seq(x,x)
r = e; x(x)
p = a*
```

predicates

```
tss(r,a,r)
os(x,p)          osr(r,p)
decl(y,x)
```

clauses

```
os(X,P):-osr(x(X),P).
osr(e,[]):-!.
osr(x(X),[A|P]):-tss(x(X),A,R),osr(R,P).
```

```
tss(x(a(A)),A,e).
tss(x(y(Y)),A,R):-
    decl(Y,G),tss(x(G),A,R).
tss(x(seq(X1,X2)),A,x(X2)):-
    tss(x(X1),A,e).
tss(x(seq(X1,X2)),A,x(seq(X11,X2))):-
    tss(x(X1),A,x(X11)).
```

Pentru testare se consideră următoarea declarație:

```
decl(y1,seq(a(a1),y(y2))).
decl(y2,seq(a(a2),a(a3))).
```

Se poate efectua următorul experiment:

```
?- os(y(y1),P),write(P),fail.
   P = ["a1","a2","a3"]
   No
```

- **Observație** Conform exercițiului 4.2.3, pentru orice program \overline{L}_1 , interpretorul nostru nu face backtracking (programul Prolog de mai sus dă o singură soluție). Este un rezultat netrivial, datorită recursivității din limbajul \overline{L}_1 .

Se mai consideră următoarea declarație:

```
decl(y3, seq(a(a), y(y3))).
```

Această declarație exprimă că avem $D(y_3) = a; y_3$.

Pentru a putea testa acest program \bar{L}_1 rescriem predicatul `osr/2` astfel:

```
osr(e, []) :- !.
osr(x(X), [A|P]) :-
    tss(x(X), A, R), write(A, ' '), osr(R, P).
```

Se poate efectua următorul experiment (se obține o secvență (conceptual) infinită,⁴ iar `T` nu va fi afișat niciodată):

```
?- os(y(y3), P), write(P), fail.
    "a" "a" "a" ...
```

⁴Practic, programul poate produce 'heap overflow'.

4.5 Prototip Haskell

- În implementarea Haskell se face uz de facilitatea de lucru cu funcții de ordin superior.

- Sintaxa \bar{L}_1

```

type Act   = String
type Y     = String
data X     = A Act | Y Y | Seq X X
type Decl  = Y -> X
type L1    = (Decl,X)

```

- Configurații

```

data R = E | X X

```

- Univers semantic

```

type P = [Act]
epsilon :: P
epsilon = []

```

- Implementarea Haskell a TSS_1 (se calculează toate tranzițiile posibile pentru fiecare configurație)

```

tss :: R -> Decl -> [(Act,R)]
tss E                d = []
tss (X (A a))        d = [(a,E)]
tss (X (Y y))        d = tss (X (d y)) d
tss (X (Seq x1 x2)) d =
  [ (a,X x2) | (a,E) <- tss (X x1) ] ++
  [ (a,X (Seq x1' x2))
    | (a,X x1') <- tss (X x1) ]

```

- Funcția semantică operațională 'os'
 - Apelul 'tss' va returna întotdeauna o listă de lungime exact 1 conform exercițiului 4.2.3

```

os :: L1 -> P
os (d,x) = osr d (X x)
  where osr :: Decl -> R -> P
        osr d E      = epsilon
        osr d (X x) =
          let [(a,r)] = tss (X x) d
              in a : osr d r

```

Pentru testarea acestui interpretor semantic se consideră următoarea declarație:

```
d :: Decl
d "y1" = Seq (A "a1") (Y "y2")
d "y2" = Seq (A "a2") (A "a3")
d "y3" = Seq (A "a") (Y "y3")
```

Se pot efectua următoarele experimente:⁵

```
Main> os (d,Y "y1")
["a1","a2","a3"]
Main> os (d,Y "y3")
["a","a","a",...
```

⁵Datorită faptului că limbajul Haskell utilizează un mecanism de evaluare leneșă ultima evaluare *nu* va produce depășire de memorie.

4.6 Exerciții

E 4.6.1 Pornind de la sistemul de tranziție TSS_1' construit în exercițiul 4.2.4 se cere:

- Să se definească semantica operațională (ca funcție matematică) pentru limbajul \bar{L}'_1 , definit în exercițiul 4.2.4.
- Să se implementeze semantica operațională a limbajului \bar{L}'_1 în Prolog sau Haskell.

E 4.6.2 *Se consideră un limbaj neuniform \overline{L}_1 care extinde limbajul \overline{L}'_1 cu o instrucțiune condițională `if b then x else x`. Clasa instrucțiunilor limbajului \overline{L}_1 ($x \in X$) este definită astfel:*

$$x ::= v := n \mid y \mid x; x \mid \text{if } b \text{ then } x \text{ else } x$$

Construcțiile $v := n, y$ și $x; x$ sunt ca la \overline{L}'_1 .

$(b \in) B$ este o clasă de expresii booleene, pentru care se consideră o sintaxă de forma

$$b ::= \text{true} \mid \dots \mid n > n \mid \dots \mid b \text{ and } b \mid \dots$$

La execuția unei instrucțiuni `if b then x_1 else x_2` se evaluează expresia b în starea curentă. Dacă b se evaluează la `true`, atunci se va executa ramura x_1 ; altfel, se va executa ramura x_2 . Semantica instrucțiunilor \overline{L}_1 se va proiecta în raport cu o mulțime $\sigma \in \Sigma (= V \rightarrow \mathbb{Z})$ de stări.

Se presupune că semantica expresiilor booleene este definită printr-o funcție de evaluare dată $\mathcal{B}[\cdot] : B \rightarrow \Sigma \rightarrow \{\text{true}, \text{false}\}$, de forma:

$$\mathcal{B}[\text{true}] (\sigma) = \text{true}$$

$$\mathcal{B}[n_1 > n_2] (\sigma) = \mathcal{N}[n_1] (\sigma) > \mathcal{N}[n_2] (\sigma)$$

$$\mathcal{B}[b_1 \text{ and } b_2] (\sigma) = \mathcal{B}[b_1] (\sigma) \text{ and } \mathcal{B}[b_2] (\sigma)$$

Se cere:

- *Să se proiecteze un sistem de tranziție TSS_1 care să specifice comportamentul instrucțiunilor pentru \bar{L}_1 . Configurațiile TSS_1 vor fi de forma $(r, \sigma) \in (Conf \times \Sigma)$, unde*

$$r ::= E \mid x \quad (Conf = \{E\} \cup X)$$

- *Să se arate că TSS_1 este determinist. Se va efectua un raționament inductiv, după o măsură de complexitate adecvată $c : Conf \rightarrow \mathbb{N}$. Pentru instrucțiunea condițională se poate pune:*

$$c(\text{if } b \text{ then } x_1 \text{ else } x_2) = 1 + \max\{c(x_1), c(x_2)\}.$$

De asemenea, pornind de la sistemul de tranziție TSS_1 se cere:

- *Să se definească semantica operațională (ca funcție matematică) pentru limbajul \bar{L}_1 .*
- *Să se implementeze semantica operațională a limbajului \bar{L}_1 în Prolog sau Haskell.*

Capitolul 5

Semantică de continuare pentru \overline{L}_1

- În semantica de continuare un program este conceptual împărțit într-o instrucțiune curentă și restul programului; o *continuare* este o reprezentare a acestui *rest* de calcul [35]
- O continuare poate fi privită drept *context de evaluare* pentru instrucțiunea sau expresia curentă [6]
- Această noțiune a fost introdusă în contextul semanticii denotaționale dar conceptul este utilizat și în semantica operațională

- Acest capitol oferă o semantică operațională proiectată în semantică de continuare pentru limbajul \bar{L}_1 introdus în capitolul 4.
- Conceptele semantice relevante sunt introduse prin *prototipizare declarativă*. Pentru limbajul \bar{L}_1 este prezentat mai întâi un interpretor semantic proiectat în semantică de continuare și implementat în limbajul Prolog.
- Este prezentat și un prototip implementat în Haskell, proiectat tot în semantică de continuare.
- Specificarea formală (semantica operațională) este construită în exerciții.
 - Reamintim că în dezvoltarea prin prototipizare, rolul prototipului este de permite specificarea sistemului; în aceste note de curs și seminar este vorba de *specificare formală*

- Sintaxa \bar{L}_1 a fost introdusă în capitolul 4.
- **Configurațiile** sistemului de tranziție proiectat în semantică de continuare pentru limbajul \bar{L}_1 [4] sunt date prin:

$$r(\in Conf) ::= E \mid x : r$$

Convenție În cele ce urmează se suprimă declarația din construcțiile semantice, presupunându-se, fără pierdere în generalitate, o declarație $D(\in Decl)$ dată (cf. [4]). Această convenție este utilizată și în capitolele următoare din partea II.

– În loc de

$$r \xrightarrow{a}_D r'$$

scriem doar

$$r \xrightarrow{a} r'$$

– Oricând avem nevoie de utilizarea unei declarații, de exemplu în specificarea regulii pentru recursivitate sau în definirea funcției semantice, se consideră dată o declarație fixată $D(\in Decl)$.

Exercițiu

E 5.0.3 *Să se rescrie sistemul de tranziție TSS_1 (din cap. 4) utilizând această convenție de notație.*

5.1 Prototip Prolog

- Sintaxa și universul semantic pentru limbajul \bar{L}_1 sunt implementate la fel ca în capitolul 4.
- Configurațiile sunt adaptate pentru semantica de continuare, conform definiției formale dată mai sus.

`domains`

`a = symbol`

`y = symbol`

`x = a(a); y(y); seq(x,x)`

`r = e; r(x,r)`

`p = a*`

- Predicatul `tss/3` dat mai jos reprezintă prototipul Prolog al sistemului de tranziție proiectat în semantică de continuare pentru limbajul \bar{L}_1 .
- Predicatul `os/2` implementează funcția semantică operațională.

- Predicatul **tss/3** dat mai jos reprezintă prototipul Prolog al sistemului de tranziție proiectat în semantică de continuare pentru limbajul \bar{L}_1 .
- Predicatul **os/2** implementează funcția semantică operațională.

```
predicates
```

```
    tss(r, a, r)
```

```
    os(x, p)
```

```
    osr(r, p)
```

```
    decl(y, x)
```

```
clauses
```

```
    tss(r(a(A), R), A, R) .
```

```
    tss(r(y(Y), R), A, R1) :-
```

```
        decl(Y, G), tss(r(G, R), A, R1) .
```

```
    tss(r(seq(X1, X2), R), A, R1) :-
```

```
        tss(r(X1, r(X2, R)), A, R1) .
```

```
    os(X, P) :- osr(r(X, e), P) .
```

```
    osr(e, []) :- ! .
```

```
    osr(R, [A|P]) :-
```

```
        tss(R, A, R1), osr(R1, P) .
```

Pentru testarea interpretorului semantic se consideră următoarea declarație:

```
decl(y1,seq(a(a1),y(y2))).  
decl(y2,seq(a(a2),a(a3))).
```

Se poate efectua următorul experiment:

```
?-os(y(y1),P),write(P),fail.  
["a1","a2","a3"]  
No
```

5.2 Prototip Haskell

- Atât în prototipul Prolog cât și în prototipul Haskell este utilizată convenția că se operează cu o declarație fixată, fără a se pierde din generalitate (cf. [4]).
 - În implementarea Haskell acest aspect este mai evident, deoarece aici declarația (care este o funcție) ar putea fi transmisă ca parametru, așa cum s-a procedat în implementarea Haskell din capitolul 4.
- Implementarea Haskell a sintaxei și a universului semantic al limbajului \bar{L}_1 sunt din în capitolul 4.

```

type Act = String
type Y   = String
data X   = A Act | Y Y | Seq X X
type Decl = Y -> X
type L1 = X

```

```

type P = [Act]
epsilon :: P
epsilon = []

```

- Configurațiile sunt specifice semanticii de continuare.

```
data R = E | R X R
```

- Funcția **tss** este un prototip declarativ ce specifică un sistem de tranziție proiectat cu continuări pentru limbajul \bar{L}_1 . **os** implementează semantica operațională a limbajului \bar{L}_1 .

```
tss :: R -> [(Act,R)]
tss E                = []
tss (R (A a) r)     = [(a,r)]
tss (R (Y y) r)     = tss (R (d y) r)
tss (R (Seq x1 x2) r) =
  tss (R x1 (R (x2) r))
```

```
os :: L1 -> P
os x = osr (R x E)
  where osr :: R -> P
        osr E = epsilon
        osr r =
          let [(a,r')] = tss r
              in a : osr r'
```

Pentru testarea interpretorului semantic se consideră următoarea declarație.

```
d :: Decl
d "y1" = Seq (A "a1") (Y "y2")
d "y2" = Seq (A "a2") (A "a3")
```

Se poate efectua următorul experiment.

```
Main> os (Y "y1")
["a1", "a2", "a3"]
```

- Desigur, și acest interpretor Haskell permite testarea programelor \bar{L}_1 care nu se termină, în baza mecanismului de *evaluare leneșă* a limbajului Haskell [20]. De exemplu, dacă declarația este extinsă cu următoarea clauză:

```
d :: Decl
d "y3" = Seq (A "a") (Y "y3")
```

se poate efectua următorul experiment:

```
Main> os (Y "y3")
["a", "a", "a", ...]
```

- Asupra comportamentului sistemului de tranziție proiectat cu continuări pentru limbajul \bar{L}_1 se poate raționa prin inducție utilizând următoarea măsură de complexitate [4]: $c : Conf \rightarrow \mathbb{N}$, $c_x : X \rightarrow \mathbb{N}$,

$$\begin{aligned} c(E) &= 0 \\ c(x : r) &= c_x(x) \end{aligned}$$

$$\begin{aligned} c_x(a) &= 1 \\ c_x(y) &= 1 + c_x(D(y)) \\ c_x(x_1; x_2) &= 1 + c_x(x_1) \end{aligned}$$

5.3 Exerciții

E 5.3.1 Să se arate că funcțiile c și c_x sunt bine definite.

E 5.3.2 Pe baza prototipurilor declarative prezentate în acest capitol să se construiască sistemul de tranziție TSS_{1con} (axiome și reguli în semantică de continuare) pentru limbajul \bar{L}_1 . (Desigur, se utilizează convenția de suprimare a declarației D din regulile TSS_{1con} .)

E 5.3.3 Să se rescrie TSS_{1con} utilizând următoare convenție de notație:

$$r \rightarrow r' \quad \text{este o abreviere pentru} \quad \frac{r' \xrightarrow{a} \bar{r}}{r \xrightarrow{a} \bar{r}}$$

Intuitiv, în această notație $r \rightarrow r'$ este o rescriere a configurației r ca și r' pentru a putea fi ulterior prelucrată.

E 5.3.4 Să se arate că sistemul de tranziție TSS_{1con} este determinist (mai precis, $|S(E)| = 0$ și $\forall x \in X, r \in Conf: |S(x : r)| = 1$).

E 5.3.5 Să se definească funcția semantică operațională bazată pe TSS_{1con} pentru \bar{L}_1 .

E 5.3.6 *Să se proiecteze o semantică de continuare pentru limbajul neuniform \bar{L}_1 ” introdus în exercițiul 4.6.2.*

- *Se va proiecta un sistem de tranziție TSS_{1con} ” care să specifice comportamentul instrucțiunilor pentru \bar{L}_1 ”. Configurațiile TSS_{1con} ” vor fi de forma $(r, \sigma) \in (Conf \times \Sigma)$, unde $(r \in Conf)$*

$$r ::= E \mid x : r$$
- *În proiectarea TSS_{1con} ” se vor urma pașii obișnuiți. Se va alege o măsură de complexitate adecvată pentru raționamentul inductiv. Se va utiliza o pereche de funcții $c : Conf \rightarrow \mathbb{N}$ și $c_x : X \rightarrow \mathbb{N}$, la fel ca în cazul sistemului de tranziție TSS_{1con} . Se va arăta că funcțiile $c : Conf \rightarrow \mathbb{N}$ și $c_x : X \rightarrow \mathbb{N}$ sunt bine definite. Apoi se va arăta că TSS_{1con} ” este determinist și se va defini funcția semantică operațională bazată pe TSS_{1con} ” pentru \bar{L}_1 ”. În final, se va implementa întreg modelul matematic în Haskell sau Prolog.*

Capitolul 6

Nedeterminism

- Un program este nedeterminist dacă la execuții diferite poate produce rezultate diferite chiar dacă de fiecare dată pornește din aceeași stare inițială și primește aceleași date de intrare
- În acest capitol este studiat un limbaj \bar{L}_2 , care extinde limbajul \bar{L}_1 cu un operator '+' de alegere nedeterministă.
- Intuitiv, execuția unei instrucțiuni $x_1 + x_2$ revine la o alegere aleatoare între două alternative: se execută fie instrucțiunea x_1 fie instrucțiunea x_2 , nu amândouă.
- Semantic, trebuie să se înregistreze toate rezultatele (sau execuțiile) posibile, și astfel se ajunge la conceptul de *domeniu putere* (engl. *power domain*), concept introdus de Gordon Plotkin [22].

- Conceptul de domeniu putere este analogul semantic al conceptului de mulțime de submulțimi (engl. *power set*).
- De obicei, dacă X este o mulțime, *mulțimea tuturor submulțimilor lui X* se notează prin $\mathcal{P}(X)$.
- Teoriile semantice (bazate pe mulțimi parțial ordonate sau spații metrice) nu se pot dezvolta însă pentru $\mathcal{P}(X)$. De obicei se impun condiții suplimentare asupra constructorului $\mathcal{P}(\cdot)$.
- Folosim notația $\mathcal{P}_{prop}(\cdot)$ pentru a denota mulțimea tuturor submulțimilor lui \cdot care au proprietatea *prop*.

De exemplu, $\mathcal{P}_{finit}(X)$ este mulțimea tuturor submulțimilor finite ale mulțimii X

- Acest capitol utilizează metode și tehnici prezentate în [4], unde semantica limbajelor este studiată în cadrul matematic al spațiilor metrice complete. Noțiunile legate de spații metrice nu le discutăm în mod sistematic. Cititorul interesat poate consulta, de exemplu, monografia [4].
- În semantica metrică, în locul construcției $\mathcal{P}(\cdot)$, se utilizează adesea construcții de forma $\mathcal{P}_{compact}(\cdot)$ sau $\mathcal{P}_{nevid_compact}(\cdot)$, adică se operează cu *submulțimi compacte*, respectiv *nevide și compacte*.¹

¹ Poate fi util să facem câteva precizări legate de noțiunea de *submulțime compactă*.

- Dacă (M, d) este un spațiu metric (M este o mulțime iar d este o metrice definită pe M), o submulțime X a lui M se zice *compactă*, dacă orice secvență din X are o subsecvență convergentă cu limita în X .
- Într-o caracterizare alternativă, X se zice compactă dacă este limita unei secvențe de submulțimi finite ale lui M , limita fiind calculată în raport cu așa-numita *metrice Hausdorff*.
- Aparatul formal al semanticii metrice este dezvoltat în cadrul matematic al spațiilor metrice complete, unde constructorul $\mathcal{P}_{compact}(\cdot)$ poate da naștere unui spațiu complet; în general însă, constructorul $\mathcal{P}(\cdot)$ sau constructorul $\mathcal{P}_{finit}(\cdot)$ nu va da naștere unui spațiu complet.

6.1 Limbajul \bar{L}_2

- Limbajul \bar{L}_2 extinde limbajul \bar{L}_1 cu operatorul '+' pentru alegere nedeterministă.

(a) Instrucțiuni:

$$x(\in X) ::= a \mid y \mid x; x \mid x + x$$

(b) Instrucțiuni gardate:

$$g(\in G) ::= a \mid g; x \mid g + g$$

(c) Declarații:

$$(D \in) Decl = Y \rightarrow G$$

(d) Programe:

$$(\pi \in) \bar{L}_2 = Decl \times X$$

- Semantica operațională a limbajului \bar{L}_2 poate fi definită pe baza unui sistem de tranziție TSS₂, proiectat în semantică directă (adică *nu* în semantică de continuare), ce operează cu configurații de forma:

$$r(\in Conf) ::= E \mid x$$

6.2 Prototip Prolog

- Oferim aici un prototip Prolog, ce specifică semantica operațională a limbajului \overline{L}_2 .
- Sintaxa instrucțiunilor (\mathbf{x}), configurațiile (\mathbf{r}) și universul semantic (\mathbf{p}) pentru limbajul \overline{L}_2 sunt date în secțiunea **domains**.
- În cazul \overline{L}_2 un element al universului semantic \mathbf{p} este o mulțime (implementată ca listă) de trasări. O trasare este un element de tip \mathbf{q} , adică o secvență (implementată ca listă) de acțiuni atomice.
- Tipurile **succ** și **succ_set** permit utilizarea predicatului **findall/3** în definiția **os/2**.
- Tipul **succ_set** permite implementarea conceptului de *mulțime succesori*.

domains

a = symbol

y = symbol

x = a(a); y(y); seq(x,x); ned(x,x)

r = e; x(x)

q = a*

p = q*

ps = p*

succ = succ(a,r)

succ_set = succ*

- Predicatul **os/2** specifică funcția semantică operațională, mapând instrucțiuni de tip **x** la valori ale universului semantic **p**.
- Predicatul **os/2** utilizează predicatul **osr/2** ce operează pe configurații de tip **r**.
- Predicatul **tss/2** specifică sistemul de tranziție pentru semantica directă a limbajului \bar{L}_2
 - Vom numi acest sistem de tranziție TSS_2 .
- De remarcat că aici predicatul **tss/2** are doi parametri, în vreme ce în capitolele precedente era declarat cu trei parametri.
 - Această decizie este motivată doar de preferința de a realiza o implementare a nedeterminismului bazată pe predicatul **findall/3**.
 - Nedeterminismul necesar generării tuturor rezultatelor² poate fi implementat și prin mecanismul implicit de backtracking din Prolog.
 - Totuși, soluția bazată pe utilizarea **findall/3** permite colectarea tuturor rezultatelor (adică a tuturor trasărilor de execuție) sub forma unei liste de tip **p**.

²În acest model semantic fiecare rezultat este, de fapt, o trasare de execuție.

- Predicatul `prefix/3` prefixează o acțiune atomică la fiecare trasare de tip `q` dintr-o colecție de trasări de tip `p`.
- Predicatele `union/3` și `bigunion` implementează operații de reuniune binară, respectiv n -ară.³
- Predicatul `exe/2` produce o listă de execuții ale tuturor perechilor din mulțimea succesor; perechile succesor sunt primite într-o listă de tip `succ_set`.
 - În definiția semanticii operaționale toate aceste execuții sunt reunite prin utilizarea `bigunion/2`.

predicates

```

os(x,p)
osr(r,p)
tss(r,succ)
exe(succ_set,ps)
elem(q,p)
prefix(a,p,p)
union(p,p,p)
bigunion(ps,p)
decl(y,x)

```

³Nu este folosit aici polimorfismul parametric introdus în Visual Prolog 7. Se utilizează tipuri concrete pentru argumentele predicatelor `elem/2`, `prefix/3`, `union/3`, `bigunion/2`,

clauses

```
os(X,P):-osr(x(X),P).
```

```
osr(e, [[]]):-!.
```

```
osr(X,P):-
```

```
    findall(S,tss(X,S),SS),
```

```
    exe(SS,PS),bigunion(PS,P).
```

```
tss(x(a(A)),succ(A,e)).
```

```
tss(x(y(Y)),succ(A,R)):-
```

```
    decl(Y,G),tss(x(G),succ(A,R)).
```

```
tss(x(seq(X1,X2)),succ(A,x(X2))):-
```

```
    tss(x(X1),succ(A,e)).
```

```
tss(x(seq(X1,X2)),
```

```
    succ(A,x(seq(X11,X2)))):-
```

```
    tss(x(X1),succ(A,x(X1))).
```

```
tss(x(ned(X,_)),succ(A,R)):-
```

```
    tss(x(X),succ(A,R)).
```

```
tss(x(ned(_,X)),succ(A,R)):-
```

```
    tss(x(X),succ(A,R)).
```

```
exe([],[]):-!.
```

```
exe([succ(A,R)|SS],[P|PS]):-
```

```
    osr(R,P1),prefix(A,P1,P),
```

```
    exe(SS,PS).
```

```
bigunion([],[]):-!.
bigunion([P1|PS],P):-
    bigunion(PS,P2),union(P1,P2,P).

union([],P,P):-!.
union([Q|P1],P2,P):-
    elem(Q,P2),!,union(P1,P2,P).
union([Q|P1],P2,[Q|P]):-
    union(P1,P2,P).

elem(Q,[Q|_]):-!.
elem(Q,[_|P]):-elem(Q,P).

prefix(_,[],[]):-!.
prefix(A,[Q|P1],[[A|Q]|P]):-
    prefix(A,P1,P).
```

- Clauza a doua a predicatului **osr/2** implementează o expresie matematică de forma: $\bigcup_{x \xrightarrow{a} r} a \cdot \mathcal{O}(r)$, sau $\bigcup \{ a \cdot \mathcal{O}(r) \mid x \xrightarrow{a} r \}$.
- Predicatul **findall/3** este utilizat în definiția **osr/2** pentru calculul mulțimii succesori pentru o configurație $x \in X$: $S(x) = \{(a, r) \mid x \xrightarrow{a} r\}$; rezultatul apelului **findall/3** este produs în variabila **SS**.
 - Tehnic vorbind, în clauza a doua a predicatului **osr/2** este implementată o expresie de forma $\bigcup \{ a \cdot \mathcal{O}(r) \mid (a, r) \in S(x) \}$
 - Predicatul **exe/2** calculează și colectează într-o listă toate execuțiile $a \cdot \mathcal{O}(r)$.
 - Apoi este utilizat predicatul **bigunion/2** pentru a calcula reuniunea tuturor acestor execuții cu producerea unei valori de tipul **p**.

- Predicatul **tss/2** specifică sistemul de tranziție TSS_2 . Clauzele pentru ațiuni atomice, recursivitate și compunere secvențială sunt la fel ca în capitolul 4.
- Cele doua clauze pentru alegerea nedeterministă pot fi scrise în notație matematică astfel:⁴

(Ned)

$$\frac{x \xrightarrow{a} r}{x + x' \xrightarrow{a} r}$$

$$\frac{x \xrightarrow{a} r}{x' + x \xrightarrow{a} r}$$

⁴Reamintim că în continuare se utilizează în mod sistematic convenția de suprimare a declarației $D(\in Decl)$ din regulile și definițiile semantice.

- Pentru testare se utilizează următoarea declarație.

```
decl(y1, ned(a(a1), seq(a(a2), y(y2)))) .
decl(y2, ned(seq(a(a3), y(y3)),
              seq(a(a4), y(y4)))) .
decl(y3, ned(a(a5), seq(a(a6), y(y4)))) .
decl(y4, a(a7)) .
```

- Aici este specificată o declarație D , pentru care:

$$D(y_1) = a_1 + a_2; y_2, D(y_2) = a_3; y_3 + a_4; y_4,$$

$$D(y_3) = a_5 + a_6; y_4 \text{ și } D(y_4) = a_7.$$

- Se poate efectua următorul experiment:

```
?-os(y(y1), P), write(P) .
[["a1"], ["a2", "a3", "a5"],
 ["a2", "a3", "a6", "a7"],
 ["a2", "a4", "a7"]]
Yes
```

- Într-o interpretare matematică, acest rezultat corespunde următoarei colecții de trasări:

$$\{a_1, a_2a_3a_5, a_2a_3a_6a_7, a_2a_4a_7\}$$

6.3 Prototip Haskell

- Lucrăm cu următoarea implementare Haskell a sintaxei \overline{L}_2 :

```

type Act = String
type Y   = String
data X   = A Act
        | Y Y
        | Seq X X
        | Ned X X
type Decl = Y -> X
type L2 = X

```

- Tipul `R` implementează mulțimea configurațiilor sistemului de tranziție TSS_2 ce specifică comportamentul programelor \overline{L}_2 .

```

data R = E | X X

```


- Tipul **P** implementează universul semantic pentru semantica operațională.
 - La fel ca în cazul prototipului Prolog, un element al domeniului final **P** este o colecție (mulțime) de trasări de execuție; în Haskell acest concept este implementat ca listă împreună cu operatori corespunzători de manipulare mulțimi: **prefix** (prefixează o acțiune atomică la fiecare secvență dintr-o colecție), **union** (reuniune binară) și **bigunion** (reuniune n -ară).
 - * În Haskell acești operatori sunt implementați în mod natural ca funcții polimorfice
 - Fiecare trasare (secvență) este implementată ca o listă Haskell de tipul **Q**; **epsilon** este secvența vidă.

```
type Q = [Act]
type P = [Q]
epsilon :: Q
epsilon = []
```

```
prefix :: (Eq a) => a -> [[a]] -> [[a]]
prefix a p = [ a:q | q <- p ]
```

```
union :: (Eq a) => [a] -> [a] -> [a]
union []      ys = ys
union (x:xs) ys =
    if (x 'elem' ys) then union xs ys
    else x : union xs ys
```

```
bigunion :: (Eq a) => [[a]] -> [a]
bigunion []      = []
bigunion (m:ms) = m 'union' bigunion ms
```

- Funcția **tss** specifică sistemul de tranziție TSS_2 , ce definește semantica construcțiilor limbajului \bar{L}_2 .
- Tehnic, funcția **tss** calculează mulțimea succesor pentru fiecare configurație \bar{L}_2 .
- În cazul alegerii nedeterministe, programul concatenează mulțimile succesor (implementate ca liste Haskell) pentru cele două alternative.
 - Acest comportament este analog cu cel al prototipului Prolog, unde se utilizează predicatul **findall/3**. Formal, ar fi fost mai corect să se efectueze o reuniune (în sens matematic) iar nu o concatenare.
 - Chiar dacă se produce o listă cu mai multe apariții ale unora dintre perechile succesor (Act, R) , acestea vor produce aceeași semantică operațională, iar trasările duplicate sunt eliminate prin utilizarea operatorului de reuniune mulțimi în definiția funcției semantice operaționale **os**.
- Funcția **os** specifică semantică operațională utilizând o notație apropiată de notația matematică.

```

tss :: R -> [(Act,R)]
tss E           = []
tss (X (A a))   = [(a,E)]
tss (X (Y y))   = tss (X (d y))
tss (X (Seq x1 x2)) d =
  [ (a,X x2) | (a,E) <- tss (X x1) ] ++
  [ (a,X (Seq x1' x2))
  | (a,X x1') <- tss (X x1) ]
tss (X (Ned x1 x2)) =
  (tss (X x1)) ++ (tss (X x2))

os :: L2 -> P
os x = osr (X x)
  where osr :: R -> P
        osr E       = [epsilon]
        osr (X x) =
          bigunion [ prefix a (osr r)
                    | (a,r) <- tss (X x) ]

```

- Pentru testare se consideră următoarea declarație:

```
d :: Decl
d "y1" = Ned (A "a1")
           (Seq (A "a2") (Y "y2"))
d "y2" = Ned (Seq (A "a3") (Y "y3"))
           (Seq (A "a4") (Y "y4"))
d "y3" = Ned (A "a5")
           (Seq (A "a6") (Y "y4"))
d "y4" = A "a7"
```

- Se poate efectua următorul experiment:

```
Main> os (Y "y1")
[["a1"], ["a2", "a3", "a5"],
 ["a2", "a3", "a6", "a7"], ["a2", "a4", "a7"]]
```

6.4 Exerciții

E 6.4.1 *Pe baza prototipurilor declarative prezentate în acest capitol să se construiască sistemul de tranziție TSS_2 pentru limbajul \bar{L}_2 .*

E 6.4.2

– *Dacă un TSS are proprietatea că pentru orice configurație $c \in Conf$ mulțimea succesori $S(c)$ este finită ($|S(c)| < \infty$), se spune că TSS este cu ramificare finită.*

– *Să se arate că sistemul de tranziție TSS_2 este cu ramificare finită.*

– **Sugestie** *Se poate proceda prin inducție după următoarea măsură de complexitate:*

$$\begin{aligned} c : Conf &\rightarrow \mathbb{N}, \quad c(E) = 0, \quad c(a) = 1, \\ c(y) &= 1 + c(D(y)), \quad c(x_1; x_2) = 1 + c(x_1), \\ c(x_1 + x_2) &= 1 + \max\{c(x_1), c(x_2)\}. \end{aligned}$$

Desigur, trebuie arătat mai întâi că funcția c este bine definită.

Observații

- Utilizând formalismul semanticii metrice [4] se poate utiliza următorul univers semantic pentru semantica operațională a limbajului \bar{L}_2 :⁵

$$\mathbb{P}_O = \mathcal{P}_{nevid_compact}(Act^\infty)$$

- Semantica operațională poate fi definită astfel:

$$\mathcal{O}[\cdot] : \bar{L}_2 \rightarrow \mathbb{P}_O, \quad \mathcal{O}[s] = \mathcal{O}(s),$$

unde $\mathcal{O} : Conf \rightarrow \mathbb{P}_O$,

$$\mathcal{O}(E) = \{\epsilon\}$$

$$\mathcal{O}(x) = \bigcup \{a \cdot \mathcal{O}(r) \mid x \xrightarrow{a} r\}$$

- * Reamintim că '.' este utilizat în aceste note de curs și seminar ca operator de concatenare secvențe.
- * Aici acest operator realizează prefixarea unei acțiuni atomice fiecărei secvențe dintr-o mulțime (nevidă și compactă).
- Există o teoremă care spune că, dacă un TSS este cu ramificare finită, atunci semantica operațională definită ca mai sus returnează o mulțime compactă. Mai precis, în acest caz, $\forall r \in Conf$, $\mathcal{O}(r) \in \mathbb{P}_O = \mathcal{P}_{nevid_compact}(Act^\infty)$.

⁵Elementele \mathbb{P}_O sunt submulțimi nevide și compacte ale Act^∞ . Noțiunea de *submulțime compactă* a fost descrisă pe scurt la începutul acestui capitol. Pentru mai multe informații cititorul poate consulta, de exemplu, [4].

Exercițiu

E 6.4.3 Pentru următoarele programe \bar{L}_2 să se calculeze semantica operațională

(a) $\mathcal{O}((a_1 + a_2); a_3)$

(b) $\mathcal{O}(a_1; (a_2 + a_3))$

(c) $\mathcal{O}(y)$, în raport cu o declarație $D \in Decl$ pentru care $D(y) = a_1; y + a_2$.

Capitolul 7

Semantică de continuare pentru \overline{L}_2

- Sintaxa limbajului \overline{L}_2 este cea introdusă în capitolul 6.
- Se poate proiecta o semantică de continuare pentru \overline{L}_2 utilizând configurații de forma:

$$r(\in Conf) ::= E \mid x : r$$

- Este oferită specificarea Prolog și Haskell a semanticii operaționale proiectată în semantică de continuare pentru limbajul \overline{L}_2 .
 - Exercițiile date la sfârșitul acestui capitol sugerează modul în care se poate construi întreg modelul matematic.
- În cele ce urmează numim *sistemul de tranziție* ce definește comportamentul limbajului \overline{L}_2 în semantică de continuare TSS_{2con} .

7.1 Prototip Prolog

- Declarațiile de domenii sunt la fel ca în capitolul 6, exceptând tipul utilizat pentru specificarea configurațiilor. Pentru a evita orice ambiguitate reproducem aici declarațiile integral.

`domains`

```
a = symbol
y = symbol
x = a(a); y(y); seq(x,x); ned(x,x)
r = e; r(x,r)
q = a*
p = q*
ps = p*
succ = succ(a,r)
succ_set = succ*
```

- Specificarea utilizează următoarele predicate:

predicates

os(x,p)

osr(r,p)

tss(r,succ)

exe(succ_set,ps)

elem(q,p)

prefix(a,p,p)

union(p,p,p)

bigunion(ps,p)

decl(y,x)

- Implementarea predicatelor **elem/2**, **prefix/3**, **union/3** și **bigunion/2** rămâne ca în capitolul 6.
- Celelalte definiții de predicate sunt date mai jos.¹

¹Strict vorbind, și predicatul **exe/3** rămâne ca în capitolul 6. Repetăm totuși aici și definiția acestui predicat.

$os(X,P) :- osr(r(X,e),P) .$

$osr(e, [[]]) :- ! .$

$osr(R,P) :-$
 $findall(S, tss(R,S), SS),$
 $exe(SS, PS), bigunion(PS, P) .$

$exe([], []) :- ! .$

$exe([succ(A,R) | SS], [P | PS]) :-$
 $osr(R, P1), prefix(A, P1, P), exe(SS, PS) .$

$tss(r(a(A), R), succ(A, R)) .$

$tss(r(y(Y), R), Succ) :-$
 $decl(Y, G), tss(r(G, R), Succ) .$

$tss(r(seq(X1, X2), R), Succ) :-$
 $tss(r(X1, r(X2, R)), Succ) .$

$tss(r(ned(X, _), R), Succ) :-$
 $tss(r(X, R), Succ) .$

$tss(r(ned(_, X), R), Succ) :-$
 $tss(r(X, R), Succ) .$

- Predicatele **os/2** și **osr/2** specifică funcția semantică operațională utilizând aceeași abordare ca în capitolul 6.
- Predicatul **tss/2** oferă o definiție compactă a semanticii de continuare a limbajului \bar{L}_2 .
- Din nou, pentru testare se utilizează următoarea declarație.

```

decl(y1, ned(a(a1), seq(a(a2), y(y2)))) .
decl(y2, ned(seq(a(a3), y(y3)),
              seq(a(a4), y(y4)))) .
decl(y3, ned(a(a5), seq(a(a6), y(y4)))) .
decl(y4, a(a7)) .

```

- Se poate efectua următorul experiment:

```

?- os(y(y1), P), write(P) .
[["a1"], ["a2", "a3", "a5"],
 ["a2", "a3", "a6", "a7"],
 ["a2", "a4", "a7"]]
Yes

```

7.2 Prototip Haskell

- Prototipul Haskell prezentat în acest capitol are multe elemente în comun cu prototipul Haskell dat în capitolul 6.
- Aici însă omitem doar definițiile operatorilor semantici de lucru cu mulțimi **union** și **bigunion**, și operatorul **prefix** de prefixare a unei acțiuni atomice la o colecție de trasări; acești operatori au fost definiți în capitolul 6.
- Se utilizează aceeași implementare a sintaxei \bar{L}_2 .

```

type Act = String
type Y    = String
data X    = A Act
          | Y Y
          | Seq X X
          | Ned X X
type Decl = Y -> X
type L2  = X

```

- Dar configurațiile sunt adaptate la semantica de continuare.

```

data R = E | R X R

```

- Și produsul final al funcției semantice rămâne tot ca în capitolul 6, un domeniu P de colecții de trasări de execuție, fiecare trasare fiind o secvență de tip Q de acțiuni atomice. `epsilon` este secvența vidă.

```

type Q = [Act]
type P = [Q]
epsilon :: Q
epsilon = []

```

- Funcția `os` este specificația Haskell a semanticii operaționale.

```

os :: L2 -> P
os x = osr (R x E)
  where osr :: R -> P
        osr E = [epsilon]
        osr r =
          bigunion [ prefix a (osr r')
                    | (a,r') <- tss r ]

```

- Tehnica continuărilor permite și aici o exprimare foarte concisă a semanticii limbajului \overline{L}_2 . Funcția **tss** reprezintă specificația Haskell a sistemului de tranziție TSS_{2con} .

```

tss :: R -> [(Act,R)]
tss E                               = []
tss (R (A a) r)                     = [(a,r)]
tss (R (Y y) r)                     = tss (R (d y) r)
tss (R (Seq x1 x2) r) = tss (R x1 (R x2 r))
tss (R (Ned x1 x2) r) =
    (tss (R x1 r)) ++ (tss (R x2 r))

```


- Se consideră următoarea declarație:

```
d :: Decl
d "y1" = Ned (A "a1")
           (Seq (A "a2") (Y "y2"))
d "y2" = Ned (Seq (A "a3") (Y "y3"))
           (Seq (A "a4") (Y "y4"))
d "y3" = Ned (A "a5")
           (Seq (A "a6") (Y "y4"))
d "y4" = A "a7"
```

- Se poate efectua următorul experiment:

```
Main> os (Y "y1")
[["a1"], ["a2", "a3", "a5"],
 ["a2", "a3", "a6", "a7"], ["a2", "a4", "a7"]]
```

7.3 Exerciții

E 7.3.1 *Pe baza prototipurilor declarative prezentate în acest capitol să se construiască sistemul de tranziție TSS_{2con} (axiome și reguli în semantică de continuare) pentru limbajul \bar{L}_2 .*

E 7.3.2 *Să se arate că sistemul de tranziție TSS_{2con} este cu ramificare finită.*

- **Sugestie** *Se va proiecta o pereche de funcții $c : Conf \rightarrow \mathbb{N}$, $c_x : X \rightarrow \mathbb{N}$, prin combinarea tehnicilor folosite în capitolele 5 (semantică de continuare) și 6 (nedeterminism).*
- *Trebuie arătat că funcțiile c și c_x sunt bine definite.*
- *Apoi se arată că TSS_{2con} este cu ramificare finită pentru orice configurație $r \in Conf$, prin inducție după $c(r)$.*

E 7.3.3 *Funcția semantică operațională $\mathcal{O}[\cdot] : \bar{L}_2 \rightarrow \mathbb{P}_O$ ($\mathbb{P}_O = \mathcal{P}_{nevid_compact}(Act^\infty)$) poate fi definită pe baza regulilor TSS_{2con} astfel $\mathcal{O}[x] = \mathcal{O}(x : E)$, unde $\mathcal{O} : Conf \rightarrow \mathbb{P}_O$ este*

$$\begin{aligned} \mathcal{O}(E) &= \{\epsilon\} \\ \mathcal{O}(x : r) &= \bigcup \{a \cdot \mathcal{O}(\bar{r}) \mid x : r \xrightarrow{a} \bar{r}\} \end{aligned}$$

După cum s-a explicat în capitolul 6, semantica operațională dată mai sus este bine definită (adică produce întotdeauna o colecție compactă de trasări, mai precis un element al universului semantic \mathbb{P}_O) deoarece TSS_{2con} este cu ramificare finită.

Să se calculeze semantica operațională $\mathcal{O}(\cdot)$ pentru următoarele programe \bar{L}_2 utilizând regulile TSS_{2con} :

- (a) $\mathcal{O}[(a_1 + a_2); a_3]$
- (b) $\mathcal{O}[a_1; (a_2 + a_3)]$

Capitolul 8

Concurență și nedeterminism

- În acest capitol este studiat un limbaj \bar{L}_3 , care extinde limbajul \bar{L}_2 cu un operator '||' de (execuție sau) compunere paralelă sau concurentă.
- Compunerea concurentă a două instrucțiuni o modelăm în *semantică de întrețesere*, (în engleză *interleaving semantics*).
 - De exemplu, dacă a_1 și a_2 sunt acțiuni atomice (elementare / neîntreruptibile) dintr-un limbaj concurent uniform (vezi secțiunea 4.1) atunci semantica compunerii paralele $a_1 \parallel a_2$ este dată de următoarea colecție de două trasări:

$$\{a_1a_2, a_2a_1\}$$

- În această abordare compunerea concurentă este interpretată pe baza conceptului de nedeterminism cu execuția întrețesută a acțiunilor elementare din componentele concurente.

8.1 Limbajul \bar{L}_3

- Limbajul \bar{L}_3 extinde limbajul \bar{L}_2 cu operatorul '||' de compunere paralelă.

(a) Instrucțiuni:

$$x(\in X) ::= a \mid y \mid x; x \mid x + x \mid x \parallel x$$

(b) Instrucțiuni gardate:

$$g(\in G) ::= a \mid g; x \mid g + g \mid g \parallel g$$

(c) Declarații:

$$(D \in) Decl = Y \rightarrow G$$

(d) Programe:

$$(\pi \in) \bar{L}_3 = Decl \times X$$

- Semantica operațională a limbajului \bar{L}_2 poate fi definită pe baza unui sistem de tranziție TSS₃ ce operează cu configurații de forma:

$$r(\in Conf) ::= E \mid x$$

8.2 Prototip Prolog

- Se specifică semantica operațională a limbajului \bar{L}_3 prin construirea unui prototip Prolog.
- Domeniile utilizate în definirea sintaxei și a universului semantic sunt ca în capitolul 6, exceptând tipul **x**, care include acum o construcție pentru compunerea paralelă.

domains

```

a = symbol
y = symbol
x = a(a); y(y);
    seq(x,x); ned(x,x); par(x,x)
r = e; x(x)
q = a*
p = q*
ps = p*
succ = succ(a,r)
succ_set = succ*

```

- Declarațiile de semnătură ale predicatelor rămân ca în secțiunea 6.2, de aceea nu mai sunt repetate aici.
- De asemenea, predicatele **bigunion**, **union**, **elem** și **prefix** rămân ca în secțiunea 6.2.

- Semantica operațională este specificată în maniera obișnuită, prin predicatele `os/2`, `osr/2` și `exe/2`.

```
os(X,P):-osr(x(X),P).
```

```
osr(e, [[]]):-!.
```

```
osr(X,P):-
```

```
    findall(S,tss(X,S),SS),
    exe(SS,PS),bigunion(PS,P).
```

```
exe([],[]):-!.
```

```
exe([succ(A,R)|SS],[P|PS]):-
```

```
    osr(R,P1),prefix(A,P1,P),exe(SS,PS).
```

- Sistemul de tranziție TSS_3 poate fi specificat în Prolog astfel:

```

tss(x(a(A)),succ(A,e)).
tss(x(y(Y)),Succ):-
    decl(Y,G),tss(x(G),Succ).
tss(x(ned(X,_)),Succ):-
    tss(x(X),Succ).
tss(x(ned(_,X)),Succ):-
    tss(x(X),Succ).
tss(x(seq(X1,X2)),succ(A,x(X2))):-
    tss(x(X1),succ(A,e)).
tss(x(seq(X1,X2)),
    succ(A,x(seq(X11,X2)))):-
    tss(x(X1),succ(A,x(X11))).
tss(x(par(X1,X2)),succ(A,x(X2))):-
    tss(x(X1),succ(A,e)).
tss(x(par(X2,X1)),succ(A,x(X2))):-
    tss(x(X1),succ(A,e)).
tss(x(par(X1,X2)),
    succ(A,x(par(X11,X2)))):-
    tss(x(X1),succ(A,x(X11))).
tss(x(par(X2,X1)),
    succ(A,x(par(X2,X11)))):-
    tss(x(X1),succ(A,x(X11))).

```


- Clauzele pentru acțiuni atomice, recursivitate, alegere nedeterministă și compunere secvențială sunt ca în capitolul 6.
- Clauzele pentru compunerea paralelă pot fi scrise în notație matematică astfel:

(Par)

$$\frac{x_1 \xrightarrow{a} E}{x_1 \parallel x_2 \xrightarrow{a} x_2}$$

$$\frac{x_1 \xrightarrow{a} E}{x_2 \parallel x_1 \xrightarrow{a} x_2}$$

$$\frac{x_1 \xrightarrow{a} x'_1}{x_1 \parallel x_2 \xrightarrow{a} x'_1 \parallel x_2}$$

$$\frac{x_1 \xrightarrow{a} x'_1}{x_2 \parallel x_1 \xrightarrow{a} x_2 \parallel x'_1}$$

- Pentru testare se utilizează următoarea declarație:

```
decl(y1,seq(a(a1),par(y(y2),y(y3)))) .  
decl(y2,ned(a(a2),a(a3))) .  
decl(y3,seq(a(a4),a(a5))) .
```

- Se pot efectua următoarele experimente:

```
?- os(par(a(a1),a(a2)),P),write(P) .  
[["a1","a2"],["a2","a1"]]  
?- os(y(y1),P),write(P) .  
[["a1","a2","a4","a5"],  
 ["a1","a3","a4","a5"],  
 ["a1","a4","a2","a5"],  
 ["a1","a4","a3","a5"],  
 ["a1","a4","a5","a2"],  
 ["a1","a4","a5","a3"]]
```

- În notație matematică declarația de mai sus poate fi descrisă astfel:

$$D(y_1) = a_1; (y_2 \parallel y_3)$$

$$D(y_2) = a_2 + a_3$$

$$D(y_3) = a_4; a_5$$

- Definiția semanticii operaționale $\mathcal{O}[\cdot]$ este discutată în secțiunea de exerciții 8.4. Experimentele de mai sus exprimă următoarele:

$$\mathcal{O}[a_1 \parallel a_2] = \{a_1a_2, a_2a_1\}$$

$$\mathcal{O}[y_1] = \{a_1a_2a_4a_5, a_1a_3a_4a_5, a_1a_4a_2a_5, \\ a_1a_4a_3a_5, a_1a_4a_5a_2, a_1a_4a_5a_3\}$$

8.3 Prototip Haskell

- Se lucrează cu următoarea implementare Haskell a sintaxei \overline{L}_3 :

```

type Act = String
type Y   = String
data X   = A Act
        | Y Y
        | Seq X X
        | Ned X X
        | Par X X
type Decl = Y -> X
type L3 = X

```

- Tipul **R** implementează mulțimea configurațiilor sistemului de tranziție TSS_3 ce specifică comportamentul programelor \overline{L}_3 .

```

data R = E | X X

```

- Tipul **P** implementează universul semantic pentru semantica operațională a limbajului \overline{L}_3 .
- Tipul **P** specifică un domeniu de mulțimi de secvențe (trasări) de execuție și este implementat (împreună cu operatorii **bigunion**, **union** și **prefix**) la fel ca în secțiunea 6.3.
- De asemenea, fiecare trasare (secvență) este implementată ca o listă Haskell de tipul **Q**; **epsilon** este secvența vidă.

```
type Q = [Act]
type P = [Q]
epsilon :: Q
epsilon = []
```

- Funcția **tss** dată mai jos specifică sistemul de tranziție TSS_3 , ce definește semantica construcțiilor limbajului \bar{L}_3 .
 - Această definiție extinde funcția **tss** dată în secțiunea 6.3. Doar clauza pentru compunerea paralelă este nouă.

```

tss :: R -> [(Act,R)]
tss E           = []
tss (X (A a))  = [(a,E)]
tss (X (Y y))  = tss (X (d y))
tss (X (Seq x1 x2)) =
  [ (a,X x2) | (a,E) <- tss (X x1) ] ++
  [ (a,X (Seq x1' x2))
  | (a,X x1') <- tss (X x1) ]
tss (X (Ned x1 x2)) =
  (tss (X x1)) ++ (tss (X x2))
tss (X (Par x1 x2)) =
  [ (a,X x2) | (a,E) <- tss (X x1) ] ++
  [ (a,X (Par x1' x2))
  | (a,X x1') <- tss (X x1) ] ++
  [ (a,X x1) | (a,E) <- tss (X x2) ] ++
  [ (a,X (Par x1 x2'))
  | (a,X x2') <- tss (X x2) ]

```

- Funcția **os** specifică semantica operațională a limbajului \bar{L}_3 .

```

os :: L3 -> P
os x = osr (X x)
  where osr :: R -> P
        osr E      = [epsilon]
        osr (X x) =
          bigunion [ prefix a (osr r)
                    | (a,r) <- tss (X x) ]

```

- Pentru testare se consideră următoarea declarație:

```
d :: Decl
d "y1" = Seq (A "a1") (Par (Y "y2") (Y "y3"))
d "y2" = Ned (A "a2") (A "a3")
d "y3" = Seq (A "a4") (A "a5")
```

- Se pot efectua următoarele experimente:

```
Main> os (Par (A "a1") (A "a2"))
[["a1", "a2"], ["a2", "a1"]]
```

```
Main> os (Y "y1")
[["a1", "a2", "a4", "a5"],
 ["a1", "a3", "a4", "a5"],
 ["a1", "a4", "a2", "a5"],
 ["a1", "a4", "a3", "a5"],
 ["a1", "a4", "a5", "a2"],
 ["a1", "a4", "a5", "a3"]]
```


8.4 Exerciții

E 8.4.1 *Pe baza prototipurilor declarative prezentate în acest capitol să se construiască sistemul de tranziție TSS_3 pentru limbajul \bar{L}_3 .*

E 8.4.2 *Să se arate că sistemul de tranziție TSS_3 este cu ramificare finită.*

– **Sugestie** *Se poate proceda prin inducție după următoarea măsură de complexitate:*

$$\begin{aligned} c : Conf &\rightarrow \mathbb{N}, \quad c(E) = 0, \quad c(a) = 1, \\ c(y) &= 1 + c(D(y)), \quad c(x_1; x_2) = 1 + c(x_1), \\ c(x_1 + x_2) &= 1 + \max\{c(x_1), c(x_2)\} \text{ și} \\ c(x_1 \| x_2) &= 1 + \max\{c(x_1), c(x_2)\}. \end{aligned}$$

Se va arăta mai întâi că funcția c este bine definită.

E 8.4.3 *Semantica operațională a limbajului \bar{L}_3 poate fi definită astfel: $\mathcal{O}[\cdot] : \bar{L}_3 \rightarrow \mathbb{P}_O$, $\mathbb{P}_O = \mathcal{P}_{nevid_compact}(Act^\infty)$, $\mathcal{O}[x] = \mathcal{O}(x)$ iar $\mathcal{O} : Conf \rightarrow \mathbb{P}_O$ este:*

$$\begin{aligned}\mathcal{O}(E) &= \{\epsilon\} \\ \mathcal{O}(x) &= \bigcup \{a \cdot \mathcal{O}(r) \mid x \xrightarrow{a} r\}\end{aligned}$$

unde definiția $\mathcal{O}(\cdot)$ este dată în raport cu TSS_3 . Ca de obicei, semantica operațională este bine definită deoarece TSS_3 este cu ramificare finită. Utilizând definiția matematică a semanticii operaționale și a TSS_3 să se calculeze:

- (a) $\mathcal{O}[a_1 \parallel a_2]$
- (b) $\mathcal{O}[y_1]$, în raport cu o declarație $D(\in Decl)$ pentru care $D(y_1) = a_1; (y_2 \parallel y_3)$, $D(y_2) = a_2 + a_3$ și $D(y_3) = a_4; a_5$.

Capitolul 9

Semantică de continuare pentru concurență

- Tehnica clasică a continuărilor [35] furnizează *flexibilitate* în proiectarea semantică, însă continuările clasice prezintă limitări în modelarea sistemelor paralele sau distribuite [11].
- În acest capitol prezentăm o semantică de continuare pentru concurență, tehnică cunoscută prin abrevierea CSC de la denumirea din limba engleză *continuation semantics for concurrency* [38, 39].
- O continuare CSC este o structură de calcule (instrucțiuni, în semantica operațională), ce se pot evalua fie într-o anumită secvență, fie în paralel.
- Structura continuărilor CSC este reprezentativă pentru conceptele de control studiate. În acest capitol se arată cum poate fi utilizat conceptul de *multi-set* pentru modelarea compunerii paralele, într-un limbaj concurent simplu \overline{L}_4 .

9.1 Conceptul de multiset

- Conceptul de multiset a fost introdus de N.G. De Bruijn, în 1970.
- În teoria mulțimilor, un *multiset* peste o mulțime $(x \in)X$ este o pereche (X, m) , unde $m : X \rightarrow \mathbb{N}$ este o funcție de la mulțimea X la mulțimea numerelor naturale \mathbb{N} (m se numește *funcție de multiplicitate*).
- Intuitiv:
 - dacă $m(x) = 0$ atunci elementul x nu este prezent în multiset
 - dacă $m(x) = n > 0$ atunci elementul x apare de n ori în multiset.
- **Observație:** Fiecare element $x \in X$ apare de un număr finit de ori într-un multiset, deoarece $m(x) \in \mathbb{N}$ (adică $m(x) \leq \infty$).

- Un multiset finit poate fi reprezentat prin enumerarea elementelor sale, fiecare element putând apărea de mai multe ori.
 - Acesta este aspectul distinctiv al structurii de multiset în raport cu structura clasică de mulțime, unde un element poate apărea o singură dată.
- Uneori, elementele unui multiset se reprezintă între paranteze '{' și '}'. Exemple:
 - $\{\}$ este multisetul vid, $\{\} = (X, \lambda x . 0)$.
 - $\{2, 3, 2, 2, 5\}$ este multisetul în care elementul 2 apare de trei ori, iar elementele 3 și 5 apar câte odată. Dacă mulțimea suport este \mathbb{N} , atunci

$$\{2, 3, 2, 2, 5\} = (\mathbb{N}, m),$$
 unde

$$m(2) = 3, m(3) = 1, m(5) = 1$$
 și

$$m(x) = 0, \forall x \in \mathbb{N}, x \notin \{2, 3, 5\}.$$

• Se pot defini **operații peste multiseturi**:

– Reuniune:

$$(X, m_1) \cup (X, m_2) = (X, \lambda x . \max\{m_1(x), m_2(x)\})$$

– Sumă:

$$(X, m_1) \uplus (X, m_2) = (X, \lambda x . m_1(x) + m_2(x))$$

– Intersecție:

$$(X, m_1) \cap (X, m_2) = (X, \lambda x . \min\{m_1(x), m_2(x)\})$$

– Diferență:

$$(X, m_1) \setminus (X, m_2) = \\ (X, \lambda x . \max\{0, m_1(x) - m_2(x)\})$$

– Incluziune:

$$(X, m_1) \subseteq (X, m_2) = \\ (\forall x \in X : m_1(x) \leq m_2(x))$$

– Test apartenență (pentru $x \in X$):

$$x \in (X, m) = (x \in X) \wedge (m(x) > 0)$$

– Cardinalitatea unui multiset:

$$|(X, m)| = \sum_{x \in X} m(x)$$

- În semantica de continuare pentru concurență conceptul de multiset este utilizat în mod natural pentru modelarea execuției concurente a proceselor, din fiecare proces putând exista una sau mai multe copii active.
- În general, se utilizează continuări CSC structurate. Structura continuărilor poate fi definită pe baza unei relații de ordine parțială [38, 41, 42]. O structură înrudită este cea de *pomset* [27].
- În acest capitol este studiat un limbaj concurent simplu, pentru care conceptul de multiset (nestructurat) este suficient.
- Fie $(x \in)X$ o mulțime. Se utilizează notația:

$$\{\!\{X\}\!\} = \{(X, m) \mid (X, m) \text{ este un multiset, } |(X, m)| < \infty\}$$

$\{\!\{X\}\!\}$ este mulțimea tuturor multiseturilor finite peste mulțimea suport X .

9.2 Limbajul \bar{L}_4

- Limbajul \bar{L}_4 este asemănător cu \bar{L}_3 , dar utilizează o formă mai particulară de compunere secvențială, bazată pe prefixare de acțiuni atomice/elementare, utilizată, de exemplu, în CCS [16] și calculul π [17].
- În \bar{L}_4 operația de secvențiere are forma $a \cdot x$, operandul stâng al operatorului \cdot fiind întotdeauna o acțiune atomică.

(a) Instrucțiuni:

$$x(\in X) ::= a \mid a \cdot x \mid y \mid x + x \mid x \parallel x$$

(b) Instrucțiuni gardate:

$$g(\in G) ::= a \mid a \cdot x \mid g + g \mid g \parallel g$$

(c) Declarații:

$$(D \in) Decl = Y \rightarrow G$$

(d) Programe:

$$(\pi \in) \bar{L}_4 = Decl \times X$$

9.3 Continuări și configurații pentru \overline{L}_4

- Restricția la operatorul de prefixare acțiuni atomice în limbajul \overline{L}_4 permite modelarea compunerii paralele în semantică de continuare CSC prin utilizarea conceptului de multiset (nestructurat).
- Se subliniază însă că structura continuărilor CSC este specifică conceptelor de control din limbajul (concurrent) studiat.
 - De exemplu, în cazul unui limbaj cum este \overline{L}_3 (ce combină compunerea paralelă cu un operator general de compunere secvențială), continuările CSC sunt structuri arborescente (matematic, se utilizează multiseturile echipate prin relații de ordine parțială [38, 41]).

- În cazul limbajului \overline{L}_4 mulțimea continuărilor este:

$$(r \in)Cont = \{X\}$$

unde X este mulțimea instrucțiunilor \overline{L}_4 .

- O configurație $\rho(\in Conf)$ a sistemului de tranziție TSS_{4csc} ce definește semantica instrucțiunilor \overline{L}_4 este fie o continuare, fie o pereche constând dintr-o instrucțiune \overline{L}_4 și o continuare:

$$(\rho \in)Conf = Cont \cup (X \times Cont)$$

$$= \{X\} \cup (X \times \{X\})$$

9.4 Sistemul de tranziție TSS_{4csc} pentru \bar{L}_4

Relația de tranziție \rightarrow pentru \bar{L}_4 este cea mai mică submulțime a $Conf \times Act \times Cont$ ce satisface următoarele axiome și reguli:

(Act)

$$(a, r) \xrightarrow{a} r$$

(Prefix)

$$(a \cdot x, r) \xrightarrow{a} \{x\} \uplus r$$

(Rec)

$$(y, r) \rightarrow (D(y), r)$$

(Ned)

$$(x_1 + x_2, r) \rightarrow (x_1, r)$$

$$(x_1 + x_2, r) \rightarrow (x_2, r)$$

(Par)

$$(x_1 \parallel x_2, r) \rightarrow (x_1, \{x_2\} \uplus r)$$

$$(x_1 \parallel x_2, r) \rightarrow (x_2, \{x_1\} \uplus r)$$

(Sched)

$$r \rightarrow (x, r \setminus \{x\}) \quad \forall x \in r$$

- Structura de multiset a continuărilor este necesară deoarece o continuare poate include mai multe copii ale oricărei instrucțiuni.
- În definiția TSS_{4csc} se utilizează convenția de notație introdusă în exercițiul 5.3.3.
- Din regulile TSS_{4csc} rezultă că pentru configurația $\{\}\} nu există nici o tranziție. $\{\}\} (\in Cont)$ este continuarea vidă (multisetul vid).$
- În axiomele (Act) și (Prefix) se execută o acțiune atomică a ; în axioma (Prexif) instrucțiunea x se adaugă la continuare.
- Regula (Rec) modelează apelul recursiv prin înlocuirea numelui procedurii cu corpul procedurii.
- Regulile (Ned) modelează alegerea nedeterministă în semantică de continuare, așa cum s-a procedat în exercițiul 7.3.1.

- Regulile (Par) și (Sched) sunt specifice tehnicii CSC. În regula (Par) se alege în mod nedeterminist una dintre cele două instrucțiuni de executat în paralel spre a rămâne activă iar cealaltă se adaugă la continuare. Orice instrucțiune rămâne activă doar până când execută o acțiune atomică. Apoi se planifică (se activează) o altă instrucțiune utilizând regula (Sched). Astfel se obține execuția întretesută (*interleaving semantics*) a instrucțiunilor concurente.
- Intuitiv, tehnica CSC este o formalizare semantică a unui planificator de procese [38].

9.5 Exerciții

E 9.5.1 Pentru orice configurație $\rho \in Conf$ să se arate că $\rho \xrightarrow{a} \rho'$ implică $\rho' \in Cont$.

- **Sugestie** Se poate proceda în doi pași. Mai întâi se demonstrează că dacă $(x, r) \in X \times Cont$ și $(x, r) \xrightarrow{a} \rho'$ atunci $\rho' \in Cont$, prin inducție după $c(x)$, unde $c : X \rightarrow \mathbb{N}$, $c(a) = 1$,
 $c(a \cdot x) = 1$, $c(y) = 1 + c(D(y))$
 și $c(x_1 + x_2) = c(x_1 \parallel x_2) = 1 + \max\{c(x_1), c(x_2)\}$.
 Apoi, utilizând rezultatul obținut în primul pas se arată că dacă $r \in Cont$ și $r \xrightarrow{a} \rho'$ atunci $\rho' \in Cont$, utilizând regula (Sched).
- Desigur, trebuie să se arate că funcția $c : X \rightarrow \mathbb{N}$ este bine definită (în doi pași, mai întâi pentru instrucțiunile gardate $\in G$, apoi pentru toate instrucțiunile $\in X$).

E 9.5.2 Să se arate că sistemul de tranziție TSS_{4csc} este cu ramificare finită.

- **Sugestie** Se poate proceda în doi pași. Mai întâi se arată că $|S(x, r)| < \infty$, pentru orice $(x, r) \in X \times \{\!\!|X\!\!\}$, prin inducție după $c(x)$. Apoi se arată că $|S(r)| < \infty$ pentru orice $r \in \{\!\!|X\!\!\}$, utilizând faptul că r este un multiset finit, conform definiției $\{\!\!|X\!\!\}$.

E 9.5.3 *Semantica operațională a limbajului \bar{L}_4 poate fi definită astfel: $\mathcal{O}[\cdot] : \bar{L}_4 \rightarrow \mathbb{P}_O$, $\mathbb{P}_O = \mathcal{P}_{nevid_compact}(Act^\infty)$, $\mathcal{O}[x] = \mathcal{O}(x, \{\})$ iar $\mathcal{O} : Conf \rightarrow \mathbb{P}_O$ este:*

$$\begin{aligned} \mathcal{O}(\{\}) &= \{\epsilon\} \\ \mathcal{O}(\rho) &= \bigcup \{a \cdot \mathcal{O}(r) \mid \rho \xrightarrow{a} r\} \quad \text{if } \rho \neq \{\} \end{aligned}$$

unde definiția $\mathcal{O}(\cdot)$ este dată în raport cu TSS_{4csc} . Ca de obicei, semantica operațională este bine definită deoarece TSS_{4csc} este cu ramificare finită.

Utilizând definiția matematică a semanticii operaționale și a TSS_{4csc} să se calculeze:

- (a) $\mathcal{O}[a_1 \parallel a_2]$
- (b) $\mathcal{O}[y_1 \parallel y_2]$, în raport cu o declarație $D(\in Decl)$ pentru care $D(y_1) = a_1 \parallel a_2$,
 $D(y_2) = a_3$.

9.6 Prototip Prolog

- Limbajul \overline{L}_4 este asemănător cu limbajul \overline{L}_3 , dar operatorul de compunere secvențială este înlocuit aici cu operatorul de prefixare acțiuni atomice pentru care este utilizat mai jos functorul **prefix**.
- Celelalte definiții de domenii sunt ca în capitolul 8. Diferența principală apare însă la domeniul pentru configurații, care este implementat prin tipul **conf** utilizând tipul **r** care implementează mulțimea continuărilor. Conceptul matematic de *multiset* este aici implementat în mod natural utilizând conceptul de *listă*. De fapt, uneori se spune că un multiset este o listă neordonată.

domains

```

a = symbol
y = symbol
x = a(a); prefix(a,x); y(y);
    ned(x,x); par(x,x)
r = x*
conf = r(r); x(x,r)
q = a*
p = q*
ps = p*
succ = succ(a,r)
succ_set = succ*

```


- Prototipul Prolog utilizează predicatele date mai jos. Predicatul **ms** implementează un algoritm de planificare pe multiseturi, utilizând mecanismul implicit de backtracking din Prolog. Predicatul **ms** realizează descompunerea unei continuări într-o instrucțiune activabilă și o continuare ce conține restul instrucțiunilor din continuarea inițială.

predicates

os(x,p)

osr(conf,p)

tss(conf,succ)

ms(r,x,r)

exe(succ_set,ps)

elem(q,p)

prefix(a,p,p)

union(p,p,p)

bigunion(ps,p)

decl(y,x)

- Clauzele ce implementează aceste predicate sunt date mai jos. Predicatele **bigunion/2**, **union/3**, **elem/2** și **prefix/3** sunt implementate ca în secțiunea 6.2. Predicatul **tss/2** dat mai jos implementează sistemul de tranziție TSS_{4csc} , iar predicatul **os/2** implementează funcția semantică operațională $\mathcal{O}[\cdot]$ pentru limbajul \bar{L}_4 . Predicatul **tss/2** utilizează predicatul **ms** pentru a realiza operația de planificare specificată în clauza (Sched) a TSS_{4csc} . Celelalte clauze ale predicatului **tss/2** implementează regulile (Act), (Prefix), (Rec), (Ned) și (Par) ale TSS_{4csc} .

```
os(X,P) :- osr(x(X, []), P) .
```

```
osr(r([], []), [ [] ]) :- ! .
```

```
osr(Conf, P) :-
    findall(S, tss(Conf, S), SS),
    exe(SS, PS), bigunion(PS, P) .
```

```
exe([], []) :- ! .
```

```
exe([succ(A, R) | SS], [P | PS]) :-
    osr(r(R), P1), prefix(A, P1, P), exe(SS, PS) .
```

```

tss(x(a(A),R),succ(A,R)).
tss(x(prefix(A,X),R),succ(A,[X|R])).
tss(x(y(Y),R),Succ):-
    decl(Y,G),tss(x(G,R),Succ).
tss(x(ned(X,_),R),Succ):-
    tss(x(X,R),Succ).
tss(x(ned(_,X),R),Succ):-
    tss(x(X,R),Succ).
tss(x(par(X1,X2),R),Succ):-
    tss(x(X1,[X2|R]),Succ).
tss(x(par(X1,X2),R),Succ):-
    tss(x(X2,[X1|R]),Succ).
tss(r(R),Succ):-
    ms(R,X,R1),tss(x(X,R1),Succ).

ms([X|R],X,R).
ms([X|R],X1,[X|R1]):-ms(R,X1,R1).

```

- Pentru testare se consideră următoarea declarație.

```
decl(y1,par(a(a1),a(a2))).  
decl(y2,a(a3)).
```

- Se pot efectua următoarele experimente:

```
?- os(par(a(a1),a(a2)),P),write(P).  
[["a1","a2"],["a2","a1"]]
```

```
?- os(par(y(y1),y(y2)),P),write(P).  
[["a1","a2","a3"],["a1","a3","a2"],  
 ["a2","a1","a3"],["a2","a3","a1"],  
 ["a3","a1","a2"],["a3","a2","a1"]]
```

9.7 Prototip Haskell

- Se consideră următoarea implementare Haskell a sintaxei \bar{L}_4 .

```

type Act = String
type Y   = String
data X   = A Act | Prefix Act X
         | Y Y | Ned X X | Par X X
type Decl = Y -> X
type L4 = X

```

- Universul semantic este ca în capitolul 8, unde s-a studiat limbajul \bar{L}_3 .

```

type Q = [Act]
type P = [Q]
epsilon :: Q
epsilon = []

```

- Configurațiile **Conf** utilizate în implementarea sistemului de tranziție TSS_{4csc} sunt specifice semanticii de continuare pentru concurență (tehnica CSC). Tipul **R** implementează mulțimea continuărilor CSC, care, în cazul limbajului \bar{L}_4 , sunt liste (multiseturi) de instrucțiuni ce se evaluează în paralel.

```

type R = [X]
data Conf = R R | X X R

```

- Semantica operațională a limbajului \overline{L}_4 este implementată de funcția **os**. Funcțiile **bigunion** și **prefix** au fost introduse în secțiunea 6.3.

```

os :: L4 -> P
os x = osr (X x [])
  where osr :: Conf -> P
        osr (R []) = [epsilon]
        osr conf   =
            bigunion [ prefix a (osr (R r))
                      | (a,r) <- tss conf ]

```

- Funcția **tss** ce implementează sistemul de tranziție TSS_{4csc} utilizează funcția **ms** care implementează algoritmul de planificare pentru multiseturi.

```

ms :: [a ] -> [(a, [a])]
ms xs = aux xs []
  where aux [] ys      = []
        aux (x:xs) ys =
            (x,xs++ys):aux xs (x:ys)

```

- Funcția `tss` implementează sistemul de tranziție TSS_{4csc} .
- `concat` este o funcție Haskell de bibliotecă care concatenează o listă de liste.¹

```

tss :: Conf -> [(Act,R)]
tss (X (A a) r)           = [(a,r)]
tss (X (Prefix a x) r) = [(a,x:r)]
tss (X (Y y) r)           =
    tss (X (d y) r)
tss (X (Ned x1 x2) r) =
    (tss (X x1 r)) ++
    (tss (X x2 r))
tss (X (Par x1 x2) r) =
    (tss (X x1 (x2:r))) ++
    (tss (X x2 (x1:r)))
tss (R r)                 =
    concat [tss (X x r')
            | (x,r') <- ms r ]

```

¹De exemplu, `concat [[1,2],[3,4]] => [1,2,3,4]`. Aici s-ar fi putut utiliza funcția `bigunion` în loc de `concat` și `union` în loc de `++`. Dar, la fel ca în capitolele precedente, pentru compatibilitate cu prototipul Prolog tranzițiile se colectează într-o listă succesori, iar nu într-o mulțime succesori.

- Pentru testare se utilizează următoarea declarație:

```
d :: Decl
d "y1" = (Par (A "a1") (A "a2"))
d "y2" = A "a3"
```

- Se pot efectua următoarele experimente:

```
Main> os (Par (A "a1") (A "a2"))
[["a1", "a2"], ["a2", "a1"]]
```

```
Main> os (Par (Y "y1") (Y "y2"))
[["a1", "a2", "a3"], ["a1", "a3", "a2"],
 ["a2", "a1", "a3"], ["a2", "a3", "a1"],
 ["a3", "a1", "a2"], ["a3", "a2", "a1"]]
```


Partea III

Semantică denotațională

Capitolul 10

Modele semantice pentru un limbaj uniform

- Se consideră un limbaj secvențial (nerecursiv) cu acțiuni elementare (atomice) neinterpretate L_0 .
- Un limbaj cu acțiuni elementare neinterpretate se zice *uniform* [4].
- Fie $(a \in)Act$ o mulțime de *acțiuni atomice*.
Definim sintaxa L_0 în BNF¹. Clasa $(x \in)X$ a instrucțiunilor L_0 este introdusă prin gramatica:

$$x ::= a \mid x; x$$

unde ' $x_1; x_2$ ' reprezintă compunerea secvențială a instrucțiunilor x_1 și x_2 ; punem $L_0 = X$.

- În Haskell, sintaxa L_0 poate fi implementată astfel:

```
type Act = String
data X    = A Act | Seq X X
```

¹Backus-Naur Formalism

10.1 Semantică directă

- Interpretarea semantică a programelor L_0 o dăm prin secvențe finite peste Act . Domeniul semantic pt. L_0 este²:

$$\mathbb{P} = Act^*$$

$$\mathbb{D} = \mathbb{P}$$

Definim o semantică denotațională directă pentru $L_0(= X)$ astfel:

$$[[\cdot]] : X \rightarrow \mathbb{D}$$

$$[[a]] = a$$

$$[[x_1; x_2]] = [[x_1]] \cdot [[x_2]]$$

unde în partea dreaptă $'\cdot'$ este operatorul semantic de concatenare secvențe peste \mathbb{D} .

² Act^* include secvența vidă. Strict vorbind, L_0 nu poate genera secvența vidă, deci domeniul Act^+ ar fi mai potrivit. Ignorăm însă acest aspect, deoarece în Haskell este convenabil să reprezentăm secvențele ca liste de tipul $[a]$, iar tipul $[a]$ include implicit lista vidă $[]$.

- Semantica denotațională directă a L_0 dată mai sus poate fi implementată în limbajul Haskell cum urmează:

```
type P = [Act]
type D = P

sem :: X -> D
sem (A a)          = [a]
sem (Seq x1 x2) =
    (sem x1) ++ (sem x2)
```

Operatorul '++' (de concatenare liste) este definit în modulul standard Prelude.hs astfel:

```
(++) :: [a] -> [a] -> [a]
[]      ++ bs = bs
(a:as) ++ bs = a:(as ++ bs)
```

– Fie

```
x0, x1 :: X
x0 = Seq (A "a") (A "b")
x1 = Seq (Seq (A "a")
            (Seq (A "b")
                  (A "c")))
        (A "d")
```

– Se pot efectua următoarele experimente³:

```
Main> sem x0
["a","b"]
Main> sem x1
["a","b","c","d"]
```

³Experimentele au fost realizate utilizând interpretorul Hugs, disponibil la adresa <http://www.haskell.org>.

Exerciții

- **E 10.1.1** *Să se arate că în L_0 compunerea secvențială este asociativă, (demonstrând asociativitatea operatorului '++').*

10.2 Semantică de continuare

- Tehnica continuărilor reprezintă un instrument deosebit de util pentru modelarea controlului în semantica denotațională. Tehnica clasică a semanticii de continuare a fost introdusă în [35].
- În această abordare, funcția denotațională depinde de un parametru adițional, numit *continuare*.
- Un program este descompus conceptual într-o instrucțiune *curentă* și *restul* programului.
- Conform definiției originale [35], o *continuare* este o reprezentare semantică a acestui *rest* de program.

- Prezentăm o semantică denotațională pentru L_0 proiectată cu tehnica clasică a continuărilor [35].
- O continuare este o reprezentare semantică a restului programului. Definim deci domeniul continuărilor pentru L_0 astfel:

$$\mathbb{P} = Act^*$$

$$(\gamma \in)Cont = \mathbb{P}$$

- Semantica de continuare pentru L_0 este:

$$\mathbb{D} = Cont \rightarrow \mathbb{P}$$

$$[\cdot] : X \rightarrow \mathbb{D}$$

$$[a] \gamma = a \cdot \gamma$$

$$[x_1; x_2] \gamma = [x_1] ([x_2] \gamma)$$

- unde operatorul \cdot de concatenare secvențe este utilizat aici pentru prefixarea unui element la o secvență, rezultatul fiind o nouă secvență (mai lungă cu un element).

- Oferim o implementare Haskell a semanticii de continuare pentru L_0 .

```

type P      = [Act]
type Cont   = P
type D      = Cont -> P

sem :: X -> D
sem (A a)      c = a:c
sem (Seq x1 x2) c =
    sem x1 (sem x2 c)

```

- Pentru a putea efectua experimente cu această funcție semantică trebuie să definim o *continuarea inițială* c_0 . În acest caz, continuarea inițială este secvență vidă de observabile (acțiuni atomice).

```

c0 :: Cont
c0 = []

```

- Fie $x_0, x_1 :: X$ ca în secțiunea 10.1. Se pot realiza următoarele evaluări:

```

Main> sem x0 c0
["a","b"]
Main> sem x1 c0
["a","b","c","d"]

```

Exerciții

- **E 10.2.1** *Să se arate că semantica de continuare (funcția `sem` din definiția de mai sus) este definită compozițional.*
- **E 10.2.2** *Să se arate că operația de compunere secvențială este asociativă și în cazul semanticii de continuare.*
- **E 10.2.3** *Să se arate că pentru orice $x :: X$ și pentru orice $c :: \text{Cont}$:*

$$\text{semc } x \ c = \text{semd } x \ ++ \ c$$

unde `semd` este semantica directă definită în secțiunea 10.1 iar `semc` este semantica de continuare definită în această secțiune.⁴ În particular avem:

$$\text{semc } x \ [] = \text{semd } x$$

⁴Operatorul '++' are prioritate mai mică decât operația de aplicare a funcției. Urmează că ecuația ce trebuie demonstrată este:

$$\text{semc } x \ c = (\text{semd } x) \ ++ \ c$$

10.3 Continuări structurate - tehnica CSC

- Tehnica continuărilor este cunoscută pentru flexibilitatea pe care o furnizează în proiectarea semantică. Însă continuările clasice prezintă limitări în modelarea sistemelor concurente [11].
- Tehnica CSC (engl. *continuation semantics for concurrency*) a fost introdusă în [38]. Această tehnică permite în același timp modelarea compunerii secvențiale și modelarea compunerii paralele în semantică de întretesere furnizând o foarte bună flexibilitate în proiectarea semantică.
- În abordarea CSC o continuare este o structură de denotații parțial evaluate, sau *calcule*.
- Structura continuărilor CSC depinde de aplicație. Urmând [41], utilizăm conceptul de *stivă* pentru a modela compunerea secvențială și conceptul de *multiset* pentru a modela compunerea paralelă.
- Conceptul de *semantică de întretesere* - utilizat în modelarea sistemelor concurente - va fi introdus într-un capitol ulterior. Deocamdată, tehnica CSC este aplicată în proiectarea unei semantici denotaționale pentru limbajul secvențial L_0 .

- În cazul limbajului L_0 o continuare CSC este o simplă stivă de calcule, pe care o implementăm ca listă în Haskell. **Comp** este domeniul calculelor (engl. *computations*) pentru L_0 .

```

type P      = [Act]

type D      = Cont -> P
data Comp   = D D
type Cont   = [Comp]

```

- Utilizând tehnica CSC se poate implementa o semantică denotațională pentru L_0 astfel:

```

sem :: X -> D
sem (A a)          c = a : cc c
sem (Seq x1 x2) c =
    sem x1 (D (sem x2) : c)

```

- **cc** (engl. *continuation completion*) este funcția ce execută continuarea. În cazul limbajului L_0 , **cc** activează calculul (denotația) din vârful stivei transmitându-i drept continuare restul stivei.

```

cc :: Cont -> P
cc []      = []
cc (D d:c) = d c

```

- În cazul modelului CSC continuarea inițială este stiva vidă de calcule.

```
c0 :: Cont
c0 = []
```

- Fie (din nou) $x0, x1 :: X$ ca în secțiunea 10.1. Se pot efectua următoarele teste:

```
Main> sem x0 c0
["a","b"]
Main> sem x1 c0
["a","b","c","d"]
```

Exerciții

- **E 10.3.1** *Să se arate că operația de compunere secvențială este asociativă și în cazul semanticii CSC.*
- **E 10.3.2** *Să se arate că semantica de continuare clasică coincide cu semantica CSC pentru L_0 . Mai precis, dacă \mathbf{sem}_{CSC} este funcția semantică definită în această secțiune (10.3), iar \mathbf{sem}_c este funcția semantică definită în secțiunea 10.2, atunci dacă \mathbf{c}_{CSC} este o continuare CSC iar \mathbf{c}_c este o continuare clasică astfel încât:*

$$\mathbf{c}_c = \mathbf{cc} \ \mathbf{c}_{CSC}$$

atunci pentru orice $\mathbf{x} :: \mathbf{X}$:

$$\mathbf{sem}_c \ \mathbf{x} \ \mathbf{c}_c = \mathbf{sem}_{CSC} \ \mathbf{x} \ \mathbf{c}_{CSC}$$

În particular asta înseamnă că relația are loc și pentru continuările inițiale:

$$\mathbf{sem}_c \ \mathbf{x} \ \mathbf{[]} = \mathbf{sem}_{CSC} \ \mathbf{x} \ \mathbf{[]}$$

deoarece

$$\mathbf{[]} = \mathbf{cc} \ \mathbf{[]}$$

10.3.1 Domenii semantice pentru CSC

- Conceptul de *domeniu* (sau *domeniu semantic*) oferă o fundamentare matematică pentru programele recursive.
- Pentru aplicații mai avansate domeniile se definesc prin *ecuații recursive de domeniu*. Este și cazul tehnicii CSC, pentru care domeniul continuărilor este soluția unei ecuații recursive în care variabila de domeniu apare în partea stângă a unei construcții de tip spațiu de funcții [38].
 - Soluția acestui tip de ecuații necesită construcții complexe preluate din teoria categoriilor [31, 3].
 - În acest sens modelul CSC este complex.
- Însă din punctul de vedere al proiectantului de limbaj tehnica CSC este ușor de utilizat, continuările putând fi proiectate prin combinarea unor structuri simple ce furnizează intuiție operațională [41].
- Este celebră ecuația de domeniu pentru calculul lambda netipizat (rezolvată de Dana Scott în 1970).

$$\mathbb{D} \cong \mathbb{D} \rightarrow \mathbb{D}$$

Simbolul ' \cong ' denotă *izomorfism* de domenii: soluția unei asemenea ecuații este determinată până la izomorfism, nu până la egalitate.

- În aceste note de curs și seminar vom utiliza în special următoarele clase de domenii (definite prin ecuații recursive):

- liste (finite sau infinite) de elemente dintr-un domeniu \mathbb{A} dat (unde ϵ denotă *terminare*, iar δ denotă *deadlock*):

$$\mathbb{Q} \cong \{\epsilon\} + (\mathbb{A} \times \mathbb{Q}), \quad \text{sau}$$

$$\mathbb{Q} \cong \{\epsilon\} + \{\delta\} + (\mathbb{A} \times \mathbb{Q});$$

- * implementarea Haskell este:

```
data Q = Epsilon | Q A Q
```

- * (sau chiar `type Q = [A]`), respectiv

```
data Q = Epsilon | Deadlock | Q A Q
```

- domenii putere de forma (\mathbb{Q} este un domeniu de liste, cum sunt cele definite mai sus; un element al domeniului \mathbb{P} este o colecție de liste⁵):

$$\mathbb{P} = \mathcal{P}(\mathbb{Q});$$

- * implementarea Haskell este:

```
type P = [Q]
```

⁵Domeniile putere au fost introduse de Gordon Plotkin [22] pentru modelarea comportamentului concurrent și nedeterminist. În exemplul nostru, dacă fiecare listă de tipul \mathbb{Q} reprezintă o trasare de execuție a unui program, atunci un element de tipul \mathbb{P} poate modela colecția tuturor trasărilor posibile pentru un program concurrent (nedeterminist).

- continuări pentru tehnica CSC, care sunt configurații (structuri) de denotații parțial evaluate (sau *calcule*); în ecuația de mai jos, o continuare (de tipul *Cont*) este o listă de denotații (adică o listă de funcții):⁶

$$\mathbb{D} = \text{Cont} \rightarrow \mathbb{P}$$

$$\text{Cont} \cong \{\text{empty}\} + (\mathbb{D} \times \text{Cont})$$

- * implementarea directă ar fi:

```
type D = Cont -> P
```

```
data Cont = Empty | Cont D Cont
```

- * dar în aplicații mai complexe este preferabil să se introducă un domeniu **Comp** de *calcule* (engl. *computations*), care poate fi elaborat în funcție de aplicație:

```
type D = Cont -> P
```

```
data Comp = D D
```

```
type Cont = [Comp]
```

- este important faptul că o asemenea listă de denotații sau calcule, poate modela un multiset⁷ de calcule (proces) executate în paralel.

⁶Dacă dorim să avem acces indexat la denotațiile dintr-o continuare, se poate utiliza un domeniu de forma:

$$\text{Cont} \cong \text{Id} \rightarrow \mathbb{D}$$

unde elementele din *Id* sunt *identificatori de proces*. O asemenea continuare poate modela un multiset de denotații. Oricum, continuările pentru concurență pot fi utilizate și pentru a modela compunerea secvențială; în acest caz, în locul disciplinei de multiset se consideră o disciplină de tip stivă.

⁷Un *multiset* este o colecție ce admite duplicate.

10.3.2 Mecanismul de evaluare a continuărilor CSC

- Există două interpretări posibile pentru conceptul de *continuare*.
 - În interpretarea inițială o continuare este o reprezentare a *restului de calcul* [35].
 - Alternativ, o continuare poate fi interpretată drept *context de evaluare* [7, 6].
- În abordarea CSC spațiul de calcule este împărțit astfel:
 - un calcul *activ* (sau o denotație activă) și
 - restul calculelor care sunt încapsulate în continuare.
- Continuările CSC pot fi împărțite în:
 - continuări active sau deschise (engl. *open continuations*)
 - continuări închise (engl. *closed continuations*)
- Ambele tipuri de continuări sunt reprezentări ale restului de calcul.
- O continuare activă (deschisă) este un context de evaluare pentru calculul activ.
 - Conceptul de continuare închisă nu poate fi descris utilizând conceptul de context de evaluare.

- Funcțiile unui interpretor semantic CSC pot fi grupate în următoarele trei componente:
 1. un evaluator
 2. o funcția de execuție a continuării și o procedură de normalizare
 3. un planificator (de procese / calcule)

- Un interpretor semantic CSC implementează o buclă "evaluate-normalize-schedule" [6, 41].

1. *Evaluatorul* mapează o continuare deschisă la un comportament program.
 - Este alcătuit din definiția (compozițională a) funcției semantice împreună cu operatori de control specifici limbajului.
 - Toate funcțiile evaluatorului manipulează (doar) continuări deschise (contexte de evaluare).
2. *Funcția de execuție a continuării* mapează o continuare deschisă la un comportament program corespunzător:
 - mai întâi apelează *procedura de normalizare* care transformă o continuare deschisă într-o continuare închisă corespunzătoare,
 - apoi apelează planificatorul.
3. *Planificatorul* mapează o continuare închisă la un comportament program.
 - În cazul unui limbaj secvențial planificatorul descompune o continuare închisă într-un calcul activ și o continuare deschisă corespunzătoare.
 - Pentru un limbaj concurent însă sunt posibile mai multe asemenea activări; asta poate da naștere la nedeterminism. În plus, poate fi necesar ca planificatorul să execute un număr de acțiuni de sincronizare înainte de activarea unui calcul.

- În continuare utilizăm
 - tipul **Kont** pentru a implementa continuările închise și
 - tipul **Cont** pentru a implementa continuările deschise.
- Diferența între continuările deschise și continuările închise poate fi codificată explicit în domeniile semantice, dar aceasta este doar o decizie de implementare.
- În cele ce urmează punem **Cont = Kont** și distingem între continuările deschise și continuările închise numai la nivel conceptual.

- Exemplificăm mecanismul de evaluare a continuărilor CSC restructurând interpretorul semantic (pentru limbajul L_0) dat în secțiunea 10.3.
- Domeniul denotațiilor D și domeniul calculelor $Comp$ rămân ca la începutul secțiunii 10.3.

```

type P      = [Act]
type D      = Cont -> P
data Comp  = D D

```

- În cazul L_0 domeniul continuărilor deschise $Cont$ și domeniul continuărilor închise $Kont$ sunt definite pe baza unui domeniu SC de stive de calcule. Conceptul de *stivă* este implementat în mod natural utilizând structura de *listă* din Haskell. Continuarea inițială ($c0 = []$) este o continuare deschisă $c0 :: Cont$.

```

type SC     = [Comp]

type Kont  = SC
type Cont  = Kont

```

- Evaluatorul interpretorului CSC pentru L_0 cuprinde definiția (compozițională a) funcției semantice împreună cu un operator de control **addc** pentru compunerea secvențială. În cazul acestui limbaj simplu **addc** adaugă un calcul în varful stivei ce implementează continuarea. Subliniem faptul că toate funcțiile evaluatorului manipulează continuări deschise.

```
sem :: X -> D
sem (A a)          c = a : cc c
sem (Seq x1 x2) c =
  sem x1 (addc (D (sem x2)) c)
```

```
addc :: Comp -> Cont -> Cont
addc p sc = p : sc
```

- Funcția de execuție a continuării **cc** normalizează continuarea și apelează planificatorul **kc**. În cazul limbajului L_0 procedura de normalizare **re** coincide cu funcția identitate.

```
cc :: Cont -> P
cc c = case re c of
  [] -> empty
  k  -> kc k
```

```
re :: Cont -> Kont
re = id
```

– Constanta `empty` modelează terminarea.

$$\begin{aligned} \text{empty} &:: P \\ \text{empty} &= [] \end{aligned}$$

– Procedura de normalizare transformă o continuare deschisă / activă (adică un context de evaluare pentru calculul activ) într-o continuare închisă (ce conține doar calcule inactive).

- În cazul foarte simplu al limbajului L_0 planificatorul descompune o continuarea închisă într-un calcul activ (o denotație activă) și o continuare deschisă corespunzătoare. Există o singură descompunere posibilă, acest fapt reflectând caracterul determinist al limbajului L_0 . Planificatorul activează denotația din vârful stivei transmitându-i restul stivei drept continuare.

$$\begin{aligned} \text{kc} &:: \text{Kont} \rightarrow P \\ \text{kc} \text{ (D d:c)} &= \text{d c} \end{aligned}$$

Capitolul 11

Semantica construcțiilor imperative

- Un limbaj se zice *neuniform* dacă semantica instrucțiunilor (elementare) depinde de starea mașinii de calcul (terminologie din [4]).
- *Starea* (care este un concept abstract) memorează valorile variabilelor programului.
- Introducem un limbaj neuniform L_1 .
- L_1 este un limbaj imperativ secvențial în care modificarea stării mașinii de calcul poate fi comandată prin instrucțiuni de atribuire, și se pot executa secvențe arbitrare de instrucțiuni.

- Definim sintaxa L_1 . Clasa $(x \in)X$ a instrucțiunilor L_1 este introdusă prin gramatica:

$$x ::= v := n \mid x ; x$$

unde se presupun date

- o clasă $(v \in)V$ de *variabile numerice*
- și o clasă $(n \in)N$ de *expresii numerice* de forma (aici $z \in \mathbb{Z}$):

$$n ::= z \mid v \mid n + n \mid \dots$$

- iar $x_1 ; x_2$ reprezintă compunerea secvențială a instrucțiunilor x_1 și x_2 .

Punem $L_1 = X$.

- În Haskell, implementăm sintaxa L_1 astfel:

```
type V = String
data N = Z Int | V V | Plus N N

data X = Assign V N | Seq X X
```

11.1 Conceptul de *stare*

- În semantica limbajelor de programare o *stare* este o funcție de la variabile la valori:¹

$$(\sigma \in) \Sigma = V \rightarrow Val$$

În implementare vom considera: $Val = \mathbb{Z}$.

- Conceptul abstract de stare este legat de capacitatea de stocare a valorilor în variabile program și de modificare a valorilor stocate în variabile.
- Fie $f : A \rightarrow B$ o funcție, $a \in A$ și $b \in B$. Notăția $(f \mid a \mapsto b)$ denotă o variantă a funcției f care este identică cu f în toate punctele, exceptând punctul a în care ia valoarea b .

$$(f \mid a \mapsto b)(a') = \begin{cases} b & \text{if } a' = a \\ f(a') & \text{if } a' \neq a \end{cases}$$

Operatorul $(f \mid a \mapsto b)$ ”perturbă” funcția f în punctul a .

- Așa cum se va vedea în continuare, acest operator poate fi utilizat
 - * atât în modelarea semanticii instrucțiunilor de atribuire din limbajele imperative,
 - * cât și în modelarea semanticii legărilor de variabile din limbajele applicative.

¹În aplicații mai complexe noțiunea poate fi elaborată în funcție de necesități.

11.1.1 Implementare ca funcție

- Domeniul stărilor Σ poate fi implementat în Haskell astfel:

```
type S = V -> Int
```

- Utilizând această reprezentare, valoarea atașată unei variabile într-o stare dată este:

```
val :: V -> S -> Int
val v s = s v
```

- În Haskell, implementăm operatorul de ”perturbare” ($f \mid a \mapsto b$) ca funcție polimorfică.

```
upd :: Eq a =>
      (a -> b) -> a -> b -> (a -> b)
upd f a b a' =
  if (a' == a) then b else f a'
```

- Această implementare corespunde fidel definiției matematice dar nu face posibilă vizualizarea stărilor.²
- În exemple considerăm că în starea inițială toate variabilele sunt neinițializate:

```
s0 :: S
s0 v =
  error ("var. neinitializata" ++ v)
```

²În general, o funcție nu poate fi vizualizată.

11.1.2 Implementare ca listă de asociație

- În aplicațiile în care dorim să vizualizăm stările preferăm o reprezentare a conceptului de stare ca listă de asocieri variabilă-valoare.

```
type S = [(V,Int)]
```

cu convenția că toate variabilele care nu sunt prezente în listă sunt implicit neinițializate.

- Desigur, această reprezentare este posibilă doar atunci când starea conține doar un număr finit de variabile ce au valori interesante.
- În această reprezentare starea inițială este:

```
s0 :: S
s0 = []
```

- Un alt exemplu este starea:

```
[("v", 10), ("u", 7)]
```

în care variabila "v" are valoarea 10, variabila "u" are valoarea 7, și toate celelalte variabile sunt neinițializate.

- Implementarea operatorilor **val** și **upd** pentru această reprezentare este dată în anexa A.

11.2 Semantica expresiilor

- În general, în L_1 valoarea unei expresii $n (\in N)$ depinde de starea curentă a mașinii.
- Semantica expresiilor este definită cu ușurință printr-o funcție de forma:

$$\mathcal{N}[\cdot] : N \rightarrow \Sigma \rightarrow \mathbb{Z}$$

$$\mathcal{N}[z] \sigma = z$$

$$\mathcal{N}[v] \sigma = \sigma(v)$$

$$\mathcal{N}[n_1 + n_2] \sigma = \mathcal{N}[n_1] \sigma + \mathcal{N}[n_2] \sigma$$

...

pe care o implementăm în Haskell astfel:

```
evN :: N -> S -> Int
```

```
evN (Z z)          s = z
```

```
evN (V v)          s = val v s
```

```
evN (Plus n1 n2) s =
```

```
    evN n1 s + evN n2 s
```

...

11.3 Transformări de stare

- În această secțiune modelăm comportamentul unui program ca funcție de transformare a stării:

`type P = S -> S`

- O asemenea funcție semantică mapează starea inițială la starea finală, care este unic determinată în cazul limbajului L_1 .
- De remarcat faptul că orice program L_1 se termină în timp finit. Acest lucru nu ar fi adevărat dacă limbajul L_1 ar incorpora un mecanism pentru definiții recursive (sau pentru iterație nedefinită). Conceptul de recursivitate va fi introdus formal într-un capitol ulterior.

11.3.1 Semantică directă

- Oferim mai întâi o semantică directă ce implementează o transformare de stare.

```
sem :: X -> P {-- type P = S -> S --}
sem (Assign v n) s =
  upd s v (evN n s)
sem (Seq x1 x2) s =
  sem x2 (sem x1 s)
```

- În semantica directă o compunere secvențială $x_1 ; x_2$ este evaluată astfel. Mai întâi se evaluează instrucțiunea x_1 în starea curentă; aceasta produce o nouă stare ce este utilizată ca stare inițială pentru evaluarea instrucțiunii x_2 .
- Operația de atribuire este implementată direct pe baza operatorului `upd`.
- Testăm acest program utilizând stări reprezentate ca liste de asociație. Fie:

```
x = Seq (Assign "v" (Plus (Z 1) (Z 2)))
        (Assign "u" (Plus (V "v") (Z 4)))
```

- Se poate efectua următorul experiment:

```
Main> sem x s0
[("v",3),("u",7)]
```


11.3.2 Semantică de continuare

- Utilizând tehnica continuărilor clasice se poate defini o semantică de transformare de stare astfel:

```
type Cont = P  {-- type P = S -> S --}
```

```
c0 :: Cont
```

```
c0 = \s -> s
```

```
sem :: X -> Cont -> P
```

```
sem (Assign v n) c s =
  c (upd s v (evN n s))
```

```
sem (Seq x1 x2) c s =
  sem x1 (sem x2 c) s
```

- De data aceasta o continuare este o funcție, iar continuarea inițială este funcția identitate.
- Se poate efectua următorul experiment:

```
Main> sem x c0 s0
[("v",3),("u",7)]
```

Exerciții

- **E 11.3.1** *Să se proiecteze o semantică CSC ca funcție de transformare de stare.*
- **E 11.3.2** *Să se arate că în L_1 compunerea secvențială este asociativă pentru fiecare dintre modelele considerate: semantica directă (dată în 11.3.1), semantica continuare (dată în 11.3.2), și modelul CSC proiectat în exercițiul 11.3.1.*

11.4 Istorii de calcul

- În această secțiune modelăm comportamentul unui program ca istorie de calcul, mai precis sub forma unei funcții ce mapează starea inițială la lista stărilor prin care trece programul:

```
type P = S -> [S]
```

- Din nou, se remarcă faptul că orice program L_1 se termină în timp finit. Acest lucru nu ar fi adevărat dacă limbajul L_1 ar incorpora un mecanism pentru definiții recursive (sau pentru iterație nedefinită). Conceptul de recursivitate va fi introdus formal într-un capitol ulterior.
 - Conceptul de istorie (listă) infinită de stări poate fi implementat pe baza mecanismului de evaluare leneșă a limbajului Haskell [20]. Deocamdată nu avem nevoie de acest mecanism.
- În această secțiune oferim numai modele proiectate în semantică de continuare.

11.4.1 Semantică de continuare clasică

- Utilizând tehnica continuărilor clasice se poate defini o semantică denotațională ce produce listă stărilor prin care trece programul astfel:

```

type Cont = P

c0 :: Cont
c0 = \s -> []

sem :: X -> Cont -> P
sem (Assign v n) c s =
  let s' = upd s v (evN n s)
      in s' : (c s')
sem (Seq x1 x2) c s =
  sem x1 (sem x2 c) s

```

- Se poate efectua următorul experiment (unde x este ca în secțiunea 11.3.1), în care am utilizat în continuare reprezentarea ca listă de asociație pentru stări:

```

Main> sem x c0 s0
[[("v",3)],[("v",3),("u",7)]]

```

11.4.2 Semantică CSC

- Proiectăm un model CSC plecând de la definițiile date în 10.3.2. Prezentăm numai definițiile care se modifică.
- Domeniul **P** este cel definit la începutul secțiunii 11.4, iar pentru domeniul stărilor utilizăm reprezentarea ca listă de asociații.
- Definițiile următoarelor domenii rămân ca în 10.3.2: **D** (denotații), **Comp** (calculare), **Kont** (continuări închise), **Cont** (continuări active / deschise).
- De asemenea, definițiile următoarelor constante și funcții rămân ca în 10.3.2: **c0** :: **Cont** (continuarea inițială), operatorul de control **addc**, funcția de execuție a continuării **cc**, procedura de normalizare **re** și funcția planificator **kc**.
- Se modifică numai constanta **empty**, care devine:

```
empty :: P
empty = \s -> []
```

și prima ecuație din definiția semanticii denotaționale (declarația de tip pentru **sem** rămâne ca în 10.3.2).

```
sem (Assign v n) c s =
  let s' = upd s v (evN n s)
  in s' : (cc c s')
```

11.4.3 Istории de valori observabile

- Reprezentarea stărilor ridică probleme la nivel de implementare. Pentru a simplifica procesul de prototipizare semantică considerăm limbajul L_2 ($(x \in)X = L_2$) ce extinde L_1 cu o construcție `write` (n) ce produce valoarea expresiei n ca element observabil.

$$x ::= v := n \mid \text{write}(n) \mid x_1 ; x_2$$

- Considerăm următoarea implementare Haskell a sintaxei L_2 .

```
data X = Assign V N | Write N | Seq X X
```

- În aceste condiții, utilizăm reprezentarea teoretică a domeniului stărilor ca funcție de la variabile la valori (dată în secțiunea 11.1.1), deoarece ne propunem ca interpretorul semantic să producă o listă de valori observabile (ce pot fi vizualizate cu ușurință), iar nu o listă de stări.

```
type S = V -> Int
```

- Modelăm comportamentul programului sub forma unei funcții ce mapează starea inițială la listă valorilor observabile produse de program.

```
type Obs = Int
type P = S -> [Obs]
```

- Prezentăm doar o semantică CSC pentru L_2 . Toate definițiile rămân ca în secțiunea 11.4.2 (dar în raport cu noul domeniu \mathbf{P}), exceptând funcția denotațională.
- Se modifică doar semantica instrucțiunii de atribuire și se adaugă o clauză pentru producerea unei valori observabile (ecuația semantică pentru compunerea secvențială rămâne neschimbată).

```
sem (Assign v n) c s =
  cc c (upd s v (evN n s))
sem (Write n)    c s =
  evN n s : cc c s
```

- Modelul semantic bazat pe istorii de valori observabile este convenabil din perspectiva prototipizării semantice.
- Fie acum

```
x :: X
x = Seq (Seq (Assign "v" (Plus (Z 1) (Z 2)))
         (Write (V "v")))
        (Seq (Assign "u" (Plus (V "v") (Z 4)))
         (Write (V "u")))
```

- Se poate efectua următorul experiment:

```
Main> sem x c0 s0
[3,7]
```

Exerciții

- **E 11.4.1** *Să se proiecteze o semantică de continuare clasică pentru L_2 .*

Capitolul 12

Recursivitate și semantică de punct fix

- Majoritatea limbajelor de programare admit definiții recursive.
- Recursivitatea introduce posibilitatea comportamentului infinit (processe de calcul perpetue).
- Justificarea riguroasă a recursivității se realizează în cadrul teoriei domeniilor utilizând *medii semantice* și *construcții punct fix*.
- Se utilizează:
 - mulțimi parțial ordonate complete și funcții continue (în acest caz se operează cu conceptul de *cel mai mic punct fix*) [24, 36, 32, 18];
 - spații metrice complete și funcții contractive (o contracție peste un spațiu metric complet are un *punct fix unic*, cf. teor. Banach) [4].

12.1 Notăția letrec

- Introducem limbajul neuniform L_1^{rec} , care extinde L_1 cu o instrucțiune vidă, selecție condițională, și recursivitate.
- Clasa $(x \in)X$ a instrucțiunilor $L_1^{rec}(= X)$ este:

$$\begin{aligned}
 x ::= & \text{ skip} \quad | \quad v := n \\
 & | \quad x; x \quad | \quad \text{if } b \text{ then } x \text{ else } x \\
 & | \quad y \quad | \quad \text{letrec } y \text{ be } x \text{ in } x
 \end{aligned}$$

- Celelalte clase sintactice sunt:
 - o clasă de *variabile numerice* $(v \in)V$ (ca în capitolul 11);
 - o clasă de *expresii numerice* $(n \in)N$ (ca în capitolul 11);
 - o clasă de *expresii booleene* $(b \in)B$ de forma:

$$b ::= n == n \quad | \quad n < n \quad | \quad \dots$$
 - o clasă de *variabile de procedură* $(y \in)Y$.

- **skip** este instrucțiunea inoperantă (nu modifică starea).
- $v := n$ este instrucțiunea de atribuire.
- $x_1; x_2$ este compunerea secvențială.
- **if b then x_1 else x_2** este instrucțiunea condițională.
- b este o expresie booleană; expresiile de acest tip pot fi testate, dar, pentru simplitate, nu pot fi atribuite; variabilele $(v \in)V$ pot păstra numai valori numerice.
- y este un apel de procedură.
- Construcția **letrec y be x_1 in x_2** definește procedura y cu corpul x_1 spre a fi utilizată în blocul x_2 .

- D.p.d.v. sintactic, L_1^{rec} este foarte asemănător cu L_1 (L_1 a fost introdus în capitulul 11).
- Semantic însă, există o mare deosebire între cele două limbaje, deoarece în L_1^{rec} există posibilitatea ca execuția unui program să nu se termine.
- Semantica L_1 poate fi definită utilizând exclusiv conceptele tradiționale de mulțime, funcție (ordinară peste mulțimi), și câteva construcții simple. De asemenea, operatorii semantici pot fi definiți prin argumente inductive.
- Pentru a defini semantica L_1^{rec} fiecare mulțime trebuie să fie înlocuită cu un domeniu, și fiecare funcție (ordinară) cu o funcție continuă. De asemenea, argumentele inductive trebuie să fie suplimentate cu argumente de punct fix.
- Intuitiv, un *domeniu semantic* este o mulțime echipată cu o anumită structură, care permite reprezentarea conceptului de calcul și aproximare în pași discreți [24, 32, 4, 36, 18].
- Deocamdată, vom utiliza conceptul de *domeniu* într-un mod strict intuitiv, dar implementările Haskell ne vor ajuta să îi înțelegem (parțial) semnificația.

12.2 Medii semantice și construcții de punct fix

- Dorim să definim o semantică denotațională pentru L_1^{rec} de forma $\llbracket \cdot \rrbracket : X \rightarrow Env \rightarrow \mathbb{D}$.
- Să zicem că introducem semantica sub forma unei funcții de transformare de stare (vezi 11.3):

$$(p \in) \mathbb{P} = \Sigma \rightarrow \Sigma$$

- Pentru ilustrarea proiectăm o semantică directă, deci punem:

$$(\phi \in) \mathbb{D} = \mathbb{P}$$

- Presupunem date funcții de evaluare pentru expresii numerice și booleene (care depind de stare):

$$\mathcal{N}[\cdot] : N \rightarrow \Sigma \rightarrow \mathbb{Z}$$

$$\mathcal{B}[\cdot] : B \rightarrow \Sigma \rightarrow \{true, false\}$$

- Elementul de noutate este adus de utilizarea unui domeniu $(\eta \in) Env$ de *medii semantice* (engl. *semantic environment*), care mapează fiecare procedură direct la semantica atașată.

$$(\eta \in) Env = Y \rightarrow \mathbb{D}$$

- Majoritatea clauzelor care definesc funcția denotațională $\llbracket \cdot \rrbracket$ pentru L_1^{rec} sunt cunoscute, sau ușor de înțeles. Modelul dat mai jos este proiectat în semantică directă (vezi secțiunea 11.3.1). Pentru lizibilitate folosim aici o notație matematică, dar modelul este implementat în secțiunea 12.4.

$$\begin{aligned}
 \llbracket \text{skip} \rrbracket \eta &= \lambda \sigma. \sigma \\
 \llbracket v := n \rrbracket \eta &= \lambda \sigma. (\sigma \mid v \mapsto \mathcal{N} \llbracket n \rrbracket \sigma) \\
 \llbracket \text{if } b \text{ then } x_1 \text{ else } x_2 \rrbracket \eta &= \lambda \sigma. \begin{cases} \llbracket x_1 \rrbracket \eta \sigma & \text{if } \mathcal{B} \llbracket b \rrbracket \sigma = \text{true} \\ \llbracket x_2 \rrbracket \eta \sigma & \text{if } \mathcal{B} \llbracket b \rrbracket \sigma = \text{false} \end{cases} \\
 \llbracket x_1; x_2 \rrbracket \eta &= \lambda \sigma. \llbracket x_2 \rrbracket \eta (\llbracket x_1 \rrbracket \eta \sigma)
 \end{aligned}$$

- Ultima clauză (care definește comportamentul compunerii secvențiale în semantică directă) este echivalentă cu:

$$\llbracket x_1 ; x_2 \rrbracket \eta = \llbracket x_2 \rrbracket \eta \circ \llbracket x_1 \rrbracket \eta$$

unde 'o' este operatorul de compunere a funcțiilor.

- De asemenea, dorim să avem:

$$\llbracket y \rrbracket \eta = \eta(y)$$

- Urmează că singura dificultatea este legată de definirea unei semantici compoziționale pentru construcția **letrec** y **be** x_1 **in** x_2 , deoarece în x_1 pot exista apeluri recursive y .
- Ideea de bază este că semantica variabilei de procedură y definită printr-o construcție **letrec** y **be** x_1 **in** \dots în raport cu un mediu semantic η este soluția următoarei *ecuații*:

$$\phi = \llbracket x_1 \rrbracket (\eta \mid y \mapsto \phi) \quad (*)$$

- Vedem că apelurile lui y din x_1 sunt interpretate ca fiind tocmai această semantică $\phi (\in \mathbb{D})$.
- Întrebarea este:
 - Există un aparat matematic care permite rezolvarea ecuației (*) ???
- Răspunsul la această întrebare este afirmativ. În această secțiune introducem doar la nivel intuitiv ideea ce stă la baza soluției.

- Desigur, se poate înlocui apariția lui ϕ din partea dreaptă a ecuației (*) cu expresia care îl definește și se obține:

$$\phi = \llbracket x_1 \rrbracket (\eta \mid y \mapsto \llbracket x_1 \rrbracket (\eta \mid y \mapsto \phi))$$

- Repetând de un număr finit de ori o asemenea 'depliere' nu vom ajunge însă la o soluție a ecuației. Totuși, intuitiv, soluția s-ar putea obține ca limită a unui șir de asemenea 'deplieri', când 'efectul' (ne-cunoscut al) lui ϕ din partea dreaptă a ecuației ar putea fi neglijat.
- Fiecare 'depliere' corespunde, intuitiv, unui nivel de imbricare a apelurilor recursive. Deoarece însă dorim să permitem un nivel arbitrar de imbricare a apelurilor recursive este necesar să angajăm un aparat matematic suficient de puternic pentru a obține o soluție generală pentru ecuația (*).
- În matematică, o ecuație de forma

$$\chi = \cdots \chi \cdots$$

se numește ecuație de *punct fix*.

- Formal, soluția ecuației (*) se obține ca punct fix al unei funcții de ordin superior de forma:

$$f : \mathbb{P} \rightarrow \mathbb{P}, \quad f(\phi) = \llbracket x_1 \rrbracket (\eta \mid y \mapsto \phi)$$

pentru orice $\eta \in Env$. Cu alte cuvinte, pentru un $\eta \in Env$ dat, soluția ecuației (*) este

$$fix(f) = fix(\lambda\phi. \llbracket x_1 \rrbracket (\eta \mid y \mapsto \phi))$$

- $fix : (\mathbb{P} \rightarrow \mathbb{P}) \rightarrow \mathbb{P}$ este operatorul ce calculează punctul fix al unei funcții de tipul $\mathbb{P} \rightarrow \mathbb{P}$. Proprietatea caracteristică a acestui operator este:

$$fix(f) = f(fix(f))$$

- Semantică a construcției **letrec** este:

$$\llbracket \text{letrec } y \text{ be } x_1 \text{ in } x_2 \rrbracket \eta = \\ \llbracket x_2 \rrbracket (\eta \mid y \mapsto fix(\lambda\phi. \llbracket x_1 \rrbracket (\eta \mid y \mapsto \phi)))$$

12.3 Notăția μ

- Construcția **letrec** este intuitivă dar mai puțin convenabilă din punct de vedere matematic.
- În studiile matematice este preferată construcția:

$$\mu y. x$$

- Utilizând sintaxa L_1^{rec} , această construcție poate fi privită drept o prescurtare pentru expresia:

$$\text{letrec } y \text{ be } x \text{ in } y$$

- Expresia $\mu y. x$ reprezintă o definiție și în același timp o utilizare a (un apel al) procedurii y .

- Pentru exemplificare considerăm și limbajul L_1^μ , definit prin gramatica $((x \in)X = L_1^\mu)$:

$$\begin{aligned}
 x ::= & \text{ skip} \quad | \quad v := n \\
 & | \quad x; x \quad | \quad \text{if } b \text{ then } x \text{ else } x \\
 & | \quad y \quad | \quad \mu y. x
 \end{aligned}$$

unde clasele sintactice $(v \in)V$, $(n \in)N$, $(b \in)B$ și $(y \in)Y$ rămân ca la L_1^{rec} (vezi secțiunea 12.1).

- Semantica construcției μ se poate defini în raport cu un mediu semantic η astfel:

$$\llbracket \mu y. x \rrbracket \eta = \text{fix}(\lambda\phi. \llbracket x \rrbracket (\eta \mid y \mapsto \phi))$$

Celelalte definiții și ecuații semantice se pot prelua fără modificări din secțiunea 12.1 și se obține o semantică directă pentru L_1^μ .

- Se pot efectua următoarele calcule:

$$\begin{aligned}
 & \llbracket \mu y. x \rrbracket \eta \\
 &= \llbracket \text{letrec } y \text{ be } x \text{ in } y \rrbracket \eta \\
 &= \llbracket y \rrbracket (\eta \mid y \mapsto \text{fix}(\lambda\phi. \llbracket x \rrbracket (\eta \mid y \mapsto \phi))) \\
 &= \text{fix}(\lambda\phi. \llbracket x \rrbracket (\eta \mid y \mapsto \phi))
 \end{aligned}$$

Exerciții

- **E 12.3.1** *Se consideră următorul program scris într-un limbaj de tip Pascal:*

$v := 0 ;$

while $v < 10$ **do** $v := v + 1 ;$

$u := 7 ;$

Se cere să se proiecteze un program cu același comportament:

1. în limbajul L_1^{rec} utilizând construcția **letrec** și
2. în limbajul L_1^μ utilizând construcția μ .

12.4 Semantică de punct fix în Haskell

- Sintaxa L_1^{rec} poate fi implementată în limbajul Haskell astfel:

```
data X = Skip | Assign V N
      | Seq X X | If B X X
      | Call Y | Letrec Y X X
```

unde declarațiile tipurilor V (variabile) și N (expresii numerice) sunt cele introduse la începutul capitolului 11, iar tipurile Y și B sunt definite astfel:

```
type Y = String
data B = Eq N N | Lt N N
```

- În această secțiune oferim numai modele în semantică de transformare de stare (ca în 11.3):

```
type P = S -> S
```

- Pentru stări utilizăm reprezentarea teoretică ca funcții de la variabile la valori (reprezentare introdusă în 11.1.1). Semantica expresiilor numerice ($\mathcal{N}[\cdot]$) este cea definită în secțiunea 11.2.
- Semantica expresiilor booleene ($\mathcal{B}[\cdot]$) este:

```
evB :: B -> S -> Bool
evB (Eq n1 n2) s = evN n1 s == evN n2 s
evB (Lt n1 n2) s = evN n1 s < evN n2 s
```

- Implementăm domeniul mediilor semantice prin tipul Haskell `Env`.

```
type Env = Y -> D
```

- Tipul `D` depinde de tehnica utilizată în proiectarea semantică (de exemplu `type D = P` pentru semantica directă și `type D = Cont -> P` pentru semantica de continuare).
- Pentru "perturbarea" mediilor semantice utilizăm operatorul polimorfic `upd` introdus în secțiunea 11.1.1. Repetăm aici definiția `upd`:

```
upd :: Eq a =>
      (a -> b) -> a -> b -> (a -> b)
upd f a b a' =
  if (a' == a) then b else f a'
```

- În exemplele de test considerăm că inițial toate procedurile sunt nedefinite.

```
e0 :: Env
e0 y = error ("proc. nedefinita " ++ y)
```

- Am ales să reprezentăm stările ca funcții de la variabile la valori.
 - Această opțiune ne permite să utilizăm același operator polimorfic **upd** pentru a modifica medii semantice sau stări.
 - Această strategie corespunde mai bine definițiilor matematice.
 - În absența unei instrucțiuni **write** însă, această strategie îngreunează testarea interpretoarelor.
- În baza mecanismului de evaluare leneșă limbajul Haskell permite următoare definiție succintă a operatorului de punct fix.

```
fix :: (a -> a) -> a
fix f = f (fix f)
```

12.4.1 Semantică directă pentru L_1^{rec} și L_1^μ

- În semantica directă codomeniul semanticii denotaționale este chiar tipul P.

type D = P

- Suntem acum pregătiți pentru prezentarea funcției semantice. Prezentăm aici o semantică directă pentru limbajul L_1^{rec} (oferim implementarea funcției semantice ce a fost introdusă în secțiunea 12.2, acolo utilizând o notație matematică).

```

sem :: X -> Env -> D
sem Skip                e = \s -> s
sem (Assign v n)       e = \s ->
  upd s v (evN n s)
sem (If b x1 x2)       e = \s ->
  if (evB b s) then sem x1 e s
                    else sem x2 e s
sem (Seq x1 x2)        e =
  (sem x2 e) . (sem x1 e)
sem (Call y)           e = e y
sem (Letrec y x1 x2) e =
  sem x2 (upd e y
          (fix
            (\d ->
              sem x1 (upd e y d))))

```


- Pentru testarea acestui interpretor semantic se consideră următorul program L_1^{rec}

```

x :: X
x =
  Letrec "while"
    (If (Lt (V "v") (Z 100))
      (Seq
        (Assign "v" (Plus (V "v") (Z 1)))
        (Call "while"))
      Skip
    )
    (Seq (Assign "v" (Z 0))
      (Call "while"))

```

- Acest program incrementează variabila (V "v") într-o buclă până când ajunge la valoarea 100.
- Reprezentarea stărilor ca funcții îngreunează testarea programelor L_1^{rec} , dar se pot efectua experimente de genul:

```

Main> let s = sem x e0 s0 in s "v"
100

```

- Studiem și limbajul L_1^μ . Implementăm sintaxa L_1^μ astfel:

```
data X = Skip | Assign V N
      | Seq X X | If B X X
      | Call Y | Mu Y X
```

- Putem obține cu ușurință un interpretor semantic pentru limbajul L_1^μ prin simpla înlocuire a ecuației ce definește semantica construcției **letrec** (din interpretorul limbajului L_1^{rec}) cu următoarea ecuație semantică pentru construcția μ :

```
sem (Mu y x) e =
  fix (\d -> sem x (upd e y d))
```

- Toate celelalte definiții (date pentru interpretorul limbajului L_1^{rec}) rămân neschimbate.

- Pentru testarea acestui interpretor semantic se consideră următorul program L_1^μ :

```
x :: X
x =
  Seq
    (Assign "v" (Z 0))
    (Mu "while"
      (If (Lt (V "v") (Z 100))
        (Seq
          (Assign "v" (Plus (V "v") (Z 1)))
          (Call "while"))
        Skip))
```

- Se poate efectua următorul experiment:

```
Main> let s = sem x e0 s0 in s "v"
100
```

12.4.2 Semantică CSC pentru L_1^{rec} și L_1^μ

- Atât continuările clasice cât și continuările structurate CSC pot fi utilizate fără nici o dificultate în semantică de punct fix.
- În această secțiune oferim modele proiectate cu tehnica CSC pentru L_1^{rec} și L_1^μ . Aceste modele pot fi reprojectate fără dificultate și cu continuări clasice.
- Aici discutăm numai semantică de transformare de stare, dar se pot proiecta la fel de bine și modele bazate pe istorii de calcul.

- În proiectarea modelului CSC pentru L_1^{rec} se utilizează toate definițiile și convențiile introduse în secțiunea 12.4 (înainte de subsecțiunea 12.4.1).
- Definițiile tipurilor **Comp**, **SC**, **Kont** și **Cont** rămân ca în secțiunea 10.3.2.
- De asemenea, definițiile următoarelor constante și funcții rămân ca în 10.3.2: **c0** :: **Cont** (continuarea inițială), operatorul de control **addc**, funcția de execuție a continuării **cc**, procedura de normalizare **re**, și funcția planificator **kc**.
- Domeniul **D** (din semnatura funcției denotaționale) este tot ca în secțiunea 10.3.2, dar repetăm aici definiția sa pentru a sublinia faptul că proiectăm o semantică de continuare.

```
type D    = Cont -> P
```

- Constanta **empty** (utilizată în definiția funcției **cc**) devine:

```
empty :: P
empty = \s -> s
```

- Cu aceste pregătiri, semantica CSC pentru L_1^{rec} este:

```

sem :: X -> Env -> D
sem Skip                e c = cc c
sem (Assign v n)       e c =
  \s -> (cc c (upd s v (evN n s)))
sem (If b x1 x2)       e c =
  \s -> if (evB b s) then sem x1 e c s
                        else sem x2 e c s
sem (Seq x1 x2)        e c =
  sem x1 e (addc (D (sem x2 e)) c)
sem (Call y)           e c = e y c
sem (Letrec y x1 x2)   e c =
  sem x2 (upd e y
          (fix
           (\d ->
            sem x1 (upd e y d)))) c

```

- Acest interpretor semantic poate fi testat pe programul L_1^{rec} dat în secțiunea 12.4.1 astfel:

```

Main> let s = sem x e0 c0 s0 in s "v"
100

```

- La fel ca și în secțiunea 12.4.1, plecând de la interpretorul CSC al L_1^{rec} se poate obține un interpretor CSC pentru L_1^μ prin simpla înlocuire a ecuației ce definește semantica construcției **letrec** cu ecuația semantică pentru construcția μ :

```
sem (Mu y x) e c =
  fix (\d -> sem x (upd e y d)) c
```

- Testarea acestui interpretor CSC se poate face pe programul L_1^μ dat în secțiunea 12.4.1 astfel:

```
Main> let s = sem x e0 c0 s0 in s "v"
100
```

Exerciții

- **E 12.4.1** *Să se elaboreze modele semantice proiectate cu continuări clasice pentru L_1^{rec} și L_1^μ .*

Capitolul 13

Semantică de continuare pentru concurență

- Tehnica CSC (engl. *continuation semantics for concurrency*) [38, 39] permite atât modelarea compunerii secvențiale cât și modelarea compunerii paralele în semantică de întretesere, furnizând o foarte bună flexibilitate în proiectarea conceptelor de control concurent.
- Intuitiv, tehnica CSC este o formalizare semantică a unui planificator de procese simulat pe o mașină secvențială. În fiecare moment există o singură denotație *activă*. O denotație rămâne activă numai până când execută o acțiune elementară. Ulterior, o altă denotație, preluată din continuare este planificată spre a deveni activă. În acest mod se poate modela execuția întretesută a proceselor paralele.

- Tehnica CSC permite proiectantului de limbaj să stabilească o relație simplă între o noțiune generală de continuare structurată și conceptele de control ale limbajului (concurrent) studiat.
- Am văzut în capitolele precedente că pentru modelarea compunerii secvențiale continuarea poate fi structurată sub forma unei stive de calcule.
- Pentru modelarea compunerii paralele în *semantica de întretesere*¹ continuările CSC pot fi structurate utilizând conceptul de *multiset*.²
- Împreună, conceptele de stivă și multiset par să furnizeze un fundament pentru semantica de fluentă a controlului.
- În aplicații mai avansate, se utilizează continuări CSC cu structură mai complexă, de exemplu arbori *ps* [41], care au la bază o combinație a conceptelor de stivă și multiset.
- Atât conceptul de stivă cât și conceptul de multiset pot fi implementate ca liste Haskell.

¹De exemplu, dacă a_1 și a_2 sunt acțiuni elementare (atomice / neîntreruptibile) dintr-un limbaj concurrent uniform (noțiunea de limbaj *uniform* a fost introdusă în capitolul 10) atunci semantica compunerii paralele $a_1 \parallel a_2$ este dată de următoarea 'mulțime' de două trasări:

$$\llbracket a_1 \parallel a_2 \rrbracket = \{a_1 a_2, a_2 a_1\}.$$

în care acțiunile a_1 și a_2 sunt 'întretesute'. Tehnic vorbind, $\{a_1 a_2, a_2 a_1\}$ este un element al unui domeniu putere cu structură lineară [4].

În cele ce urmează vom defini semantica denotațională a concurenței utilizând tehnica continuărilor, deci funcția denotațională va mai avea un parametru (continuarea). Semantica compunerii paralele poate fi însă definită și fără continuări [26, 4].

²Un multiset este o colecție ce admite elemente duplicate.

- Comportamentul structurii de stivă poate fi definit direct utilizând operațiile de bază (*car*, *cdr*, *cons*) atașate listelor Haskell.
- Comportamentul structurii de multiset (din punctul de vedere al planificatoarelor CSC) este definit de următorul algoritm de planificare a calculelor dintr-un multiset:

```

ms :: [a] -> [(a, [a])]
ms xs = aux xs []
  where aux [] ys = []
        aux (x : xs) ys =
          (x, xs ++ ys) : aux xs (x : ys)

```

– Se consideră o continuare CSC închisă reprezentată prin multisetul: $[d1, d2, d3]$. Atunci:

$$\begin{aligned} ms [d1, d2, d3] = & \\ & [(d1, [\bullet, d2, d3]), \\ & (d2, [\bullet, d3, d1]), \\ & (d3, [\bullet, d2, d1])] \end{aligned}$$

- * Există trei descompuneri (planificări) posibile. În fiecare este ales un element al multisetului spre a fi activat iar celelalte două elemente formează o continuare deschisă.
- * Cerculețul plin '●' (care sugerează lipsa unui element) indică poziția conceptuală a elementului activabil (care a fost selectat).
- * Un multiset este o structură neordonată; poziția exactă a elementului activ (sau a celorlalte elemente) în lista care implementează multisetul nu este importantă.
- * Alegem să considerăm că elementul activ este plasat în capul listei ce implementează continuarea deschisă.

13.1 Compunere paralelă

- În această secțiune studiem limbajul L_{par} , ce furnizează o construcție $x_1 \parallel x_2$ pentru compunerea paralelă a proceselor. Clasa $(x \in)X$ a instrucțiunilor $L_{par}(= X)$ este introdusă prin urătoarea gramatică:

$$\begin{aligned}
 a & ::= v := n \mid \text{write}(n) \\
 x & ::= \text{skip} \mid a \cdot x \mid \text{if } b \text{ then } x \text{ else } x \\
 & \quad \mid y \mid \text{letrec } y \text{ be } x \text{ in } x \mid x \parallel x
 \end{aligned}$$

unde clasele $(\in)V$ a variabilelor numerice, $(n \in)N$ a expresiilor numerice, $(b \in)B$ a expresiilor booleene și $(y \in)Y$ a variabilelor de procedură sunt ca în capitolul 12.

- Reutilizând declarațiile tipurilor V , N , B și Y din capitolul 12 implementăm sintaxa L_{par} astfel:

```

data A = Assign V N | Write N
data X = Skip | Prefix A X | If B X X
      | Call Y | Letrec Y X X
      | Par X X

```

- Construim o semantică denotațională pentru L_{par} proiectată cu tehnica CSC.
- În această construcție considerăm reprezentarea stărilor (tipul \mathbf{S}) ca funcții de la variabile la valori (vezi

11.1.1), iar semantica expresiilor numerice și booleene rămân ca în capitolul 12.

- Optăm aici pentru o funcție semantică ce depinde de starea curentă și produce istorii de observabile
 1. fie asamblate în colecția tuturor trasărilor posibile,
 2. fie alese în mod (pseudo-)aleator.
- În prima variantă interpretorul semantic implementează o semantică denotațională clasică ce utilizează un *domeniu putere* [22] pentru a implementa conceptul de nedeterminism. În acest caz interpretorul nu este tractabil, putând fi testat numai pe programe ”de jucărie”.
- În cea de a doua variantă, nedeterminismul inerent unui sistem concurent este modelat cu ajutorul unui generator de numere (pseudo-)aleatoare. În acest caz este vorba despre o simulare a nedeterminismului, dar interpretorul semantic este *tractabil* putând fi testat pe programe netriviiale.

13.1.1 Specificare

- Începem prin a prezenta modelul teoretic ce produce toate trasările posibile pentru un program nedeterminist. În acest model semantica denotațională depinde de starea curentă și produce colecții de trasări. Implementăm acest model pe baza următoarelor tipuri Haskell:

```

type P    = S -> [Q]
data Q    = Empty | Observe Obs Q
type Obs  = Int

```

- Constanta `empty` este:

```

empty :: P
empty = \s -> [Empty]

```

- Tipul `Q` implementează secvențe de observabile. În principiu ar fi putut fi implementat printr-o declarație de forma `type Q = [Obs]`, dar declarația pentru care am optat aici va permite o tranziție mai ușoară spre un model semantic ce încorporează noțiunea de *blocare* (engl. *deadlock*) într-o secțiune ulterioară. În anexa B sunt prezentate o instanță `Eq` și o instanță `Show` pentru această declarație a tipului `Q`.

- Pentru tipul P implementăm:

- o operație **put** de prefixare a unei valori observabile la un element de tipul P

```
put :: Obs -> P -> P
```

```
put o p =
```

```
  \s -> [ Observe o q | q <- p s ]
```

- două operații pentru modelarea alegerii nedeterminate

* Operația **ned** modelează o alegere nedeterministă între două alternative; operația **bigned** modelează o alegere nedeterministă între o listă de alternative. Alegerea nedeterministă este aici o reuniune de comportamente.

```
ned :: P -> P -> P
```

```
ned p1 p2 = \s -> union (p1 s) (p2 s)
```

```
bigned :: [P] -> P
```

```
bigned [] = \s -> []
```

```
bigned (p : ps) = ned p (bigned ps)
```

```
union [] ys = ys
```

```
union (x : xs) ys =
```

```
  if (x 'elem' ys) then union xs ys
```

```
  else x : union xs ys
```


- Deși multe definiții se repetă din capitolele anterioare, pentru a evita orice ambiguitate oferim o prezentare completă a interpretorului semantic pentru L_{par} .
- Domeniile **D** (de denotații) și **Env** (de medii semantice) sunt cele uzuale pentru semantica de continuare.

```

type D      = Cont -> P
type Env    = Y  -> D

```

- Definiția mediului semantic inițial **e0** precum și definiția operatorului polimorfic **upd** rămân ca în secțiunea 12.4.
- Deocamdată și definiția domeniului de calcule **Comp** rămâne ca în 10.3.2, dar această definiție va trebui să fie modificată în secțiunea 13.2 odată cu introducerea conceptului de sincronizare.

```

data Comp   = Den D

```

- Pentru modelarea compunerii paralele a proceselor din L_{par} domeniul continuărilor este definit pe baza unui domeniu **PC** de multiseturi de calcule. Conceptul de *multiset* este implementat utilizând structura Haskell de *listă*. Continuarea inițială (**c0** :: **Cont**) este **c0** = [].

```
type PC    = [Comp]

type Kont = PC
type Cont = Kont
```

Reamintim următoarele:

- **Cont** este domeniul continuărilor deschise (sau active), iar **Kont** este domeniul continuărilor închise.
- O continuare deschisă este un context de evaluare pentru denotația activă.
- Continuările deschise sunt manipulate de evaluator, iar continuările închise sunt manipulate de planificator.

- Evaluatorului CSC pentru L_{par} cuprinde definiția funcțiilor semantice **semA** și **sem** împreună cu operatorii de control **addc** și **addp**.

```

semA :: A -> D
semA (Assign v n) c =
  \s -> cc c (upd s v (evN n s))
semA (Write n)      c =
  \s -> put (evN n s) (cc c) s

sem :: X -> Env -> D
sem Skip                e c = cc c
sem (Prefix a x)        e c =
  semA a (addc (Den (sem x e)) c)
sem (Call y)            e c = e y c
sem (Letrec y x1 x2) e c =
  sem x2 (upd e y
    (fix
      (\d ->
        sem x1 (upd e y d)))) c
sem (If b x1 x2)        e c = \s ->
  if evB b s then sem x1 e c s
  else sem x2 e c s
sem (Par x1 x2)         e c =
  sem x1 e (addp (Den (sem x2 e)) c) 'ned'
  sem x2 e (addp (Den (sem x1 e)) c)

```

$$\begin{aligned} \text{addc} &:: \text{Comp} \rightarrow \text{Cont} \rightarrow \text{Cont} \\ \text{addc } p \text{ } sc &= p : sc \end{aligned}$$

$$\begin{aligned} \text{addp} &:: \text{Comp} \rightarrow \text{Cont} \rightarrow \text{Cont} \\ \text{addp } p \text{ } pc &= p : pc \end{aligned}$$

Compunerea paralelă a două instrucțiuni **x1** și **x2** este modelată ca o alegere nedeterministă între două alternative: una începând cu instrucțiunea **x1**, cealaltă începând cu **x2**.

În cazul simplu al limbajului L_{par} operatorii de control **addc** și **addp** adaugă un calcul la multiset. În alte aplicații definițiile operatorilor de control ai evaluatorului pot fi ceva mai complicate [41].

- Funcția de execuție a continuării și procedura de normalizare rămân ca în capitolele precedente; doar constanta **empty** se modifică (repetăm totuși definițiile funcțiilor **cc** și **re**).

```

cc :: Cont -> P
cc c = case re c of
        [] -> empty
        k  -> kc k

```

```

re :: Cont -> Kont
re = id

```

Procedura de normalizare este funcția identitate. Pentru limbaje ce incorporează construcții de control mai avansate (de exemplu pentru limbaje modulate prin continuări structurate sub forma arborilor *ps*) definiția procedurii de normalizare este ceva mai complexă.

- Reamintim că procedura de normalizare transformă o continuare activă (deschisă) într-o continuare închisă corespunzătoare. În esență, continuările CSC sunt normalizare pentru a facilita procesul de planificare.

- Planificatorul CSC al limbajului L_{par} alege în mod nedeterminist între toate planificările posibile. Domeniul planificărilor este **Sched**. Acest domeniu va fi elaborat în secțiunea 13.2 unde modelăm sincronizarea proceselor concurente.

```
data Sched = Scheda D Cont
```

```
kc :: Kont -> P
```

```
kc k = bigned (map exe (scheda k))
      where exe (Scheda d c) = d c
```

```
scheda :: Kont -> [Sched]
```

```
scheda k =
  [ Scheda d c | (Den d,c) <- actc k ]
```

```
actc :: Kont -> [(Comp,Cont)]
```

```
actc = ms
```

Funcția **scheda** utilizează **actc** pentru a calcula toate planificările de activare posibile. În cazul simplu al limbajului L_{par} (unde o continuare este un multiset) funcția **actc** coincide cu **ms**. Pentru structuri de continuări mai complexe definiția **actc** este mai elaborată.

- Pentru testarea acestui interpretor semantic considerăm următoarea instrucțiune L_{par} :

```
x1 :: X
x1 =
  Par
    (Prefix (Write (Z 1))
      (Prefix (Write (Z 2)) Skip))
    (Prefix (Write (Z 3)) Skip)
```

- Se poate efectua următorul experiment:

```
Main> sem x1 e0 c0 s0
[[1,2,3], [1,3,2], [3,1,2]]
```

13.1.2 Prototip

- Pentru testarea unui interpretor semantic modelele bazate pe domenii putere nu sunt convenabile deoarece pot fi evaluate numai pe exemple foarte simple.
 - Problema este că un element al unui domeniu putere este exponențial în lungimea trasărilor de execuție.
- Este însă posibil să se modifice un interpretor CSC astfel încât să producă la fiecare execuție o singură trasare aleasă în mod (pseudo-)aleator. Modificările necesare sunt localizate exclusiv la nivelul domeniului P .
- În acest scop se utilizează un domeniu **RNG** de generatoare de numere aleatoare. O valoare de tipul **RNG** este o pereche constând dintr-un număr aleator și o funcție ce poate produce un număr aleator nou. Un număr aleator este o valoare de tipul **R**.

- În experimente utilizăm următorul generator simplu de numere aleatoare (preluat din [37]).

```
type R = Int
type RNG = (R, R -> R)
```

```
rng0 :: RNG
rng0 = (17489,
        \r -> (25173 * r + 13849)
          'mod' 65536)
```

- Se poate obține un interpretor CSC pentru L_{par} ce produce o unică trasare (arbitrar aleasă) la fiecare execuție modificând exclusiv domeniul P și operatorii atașați cum urmează:

```
type P = S -> RNG -> Q

put :: Obs -> P -> P
put o p = \s r -> Observe o (p s r)
```

```

ned :: P -> P -> P
ned p1 p2 = bigned [p1,p2]

bigned :: [P] -> P
bigned ps =
  \s (r,next) ->
    (rand ps r s (next r,next))
  where rand :: [a] -> R -> a
        rand xs r =
          nth xs (r 'mod' (length xs))
  nth :: [a] -> R -> a
  nth (x:xs) 0 = x
  nth (x:xs) r = nth xs (r-1)

```

- Mai trebuie modificată doar definiția constantei `empty` (care este tot de tip `P`).

```

empty :: P
empty = \s r -> Empty

```

- Toate celelalte definiții rămân așa cum au fost date în secțiunea 13.1.1.

- Pentru testarea acestei noi variante a interpretorului este convenabil să utilizăm următoarea funcție, care execută de un număr specificat de ori o instrucțiune inițializând de fiecare dată generatorul de numere aleatoare cu o altă valoare. În consecință, la execuții succesive se pot obține trăsări diferite pentru același program (dacă programul este nedeterminist).

```

test :: X -> Int -> IO ()
test x n = run x n rng0
  where
    run :: X -> Int -> RNG -> IO ()
    run x 0 rng      = return ()
    run x n (r,next) =
      do print (sem x e0 c0 s0 (r,next))
         run x (n-1) (next r,next)

```

- Pentru testarea interpretorului considerăm instrucțiunea **x1** introdusă în secțiunea 13.1.1, împreună cu următoarea instrucțiune L_{par} :

```

x2 :: X
x2 =
  (Letrec "y1"
    (If (Lt (Z 0) (V "v"))
      (Prefix (Write (Z 1))
        (Prefix (Atrib "v"
          (Minus (V "v") (Z 1)))
          (Call "y1"))))
      Skip)
  (Letrec "y2"
    (If (Lt (Z 0) (V "u"))
      (Prefix (Write (Z 2))
        (Prefix (Atrib "u"
          (Minus (V "u") (Z 1)))
          (Call "y2"))))
      Skip)
  (Letrec "y3"
    (If (Lt (Z 0) (V "w"))
      (Prefix (Write (Z 3))
        (Prefix (Atrib "w"
          (Minus (V "w") (Z 1)))
          (Call "y3"))))
      Skip)

```

```
(Prefix (Atrib "v" (Z 5))
  (Prefix (Atrib "u" (Z 5))
    (Prefix (Atrib "w" (Z 5))
      (Par (Par (Call "y1") (Call "y2"))
        (Call "y3"))))))))
```

- Programul `x2` execută în paralel trei procese; execuția proceselor paralele apare întretesută pentru observatorul extern.
- Se pot efectua următoarele experimente:

```
Main> test x1 5
[3,1,2]
[1,3,2]
[3,1,2]
[1,3,2]
[3,1,2]
Main> test x2 5
[2,3,1,3,2,1,2,3,1,2,3,1,2,1,3]
[3,1,2,3,1,3,2,2,1,3,1,2,3,1,2]
[2,1,3,2,1,3,3,1,2,2,1,3,1,2,3]
[3,1,2,3,3,2,1,3,2,1,3,2,1,2,1]
[2,1,3,3,1,2,2,3,1,2,3,1,3,1,2]
```

- În general, la fiecare execuție se poate obține o trasare diferită.

13.2 Comunicare sincronă

- Introducem un limbaj L_{syn} , care extinde L_{par} cu comunicare sincronă de tip CSP [12, 13].
- Clasa $(x \in)X$ a instrucțiunilor $L_{syn}(= X)$ este dată de următoarea gramatică:

$$\begin{aligned}
 a & ::= v := n \mid \text{write}(n) \\
 x & ::= \text{skip} \mid a \cdot x \mid \text{if } b \text{ then } x \text{ else } x \\
 & \quad \mid y \mid \text{letrec } y \text{ be } x \text{ in } x \mid x \parallel x \\
 & \quad \mid \text{ned}[(\vartheta \rightarrow i)^*]
 \end{aligned}$$

- Unica noutate este dată de construcția $\text{ned}[(\vartheta \rightarrow i)^*]$. Toate celelalte construcții și clase sintactice rămân ca în limbajul L_{par} .
- Se utilizează o clasă $(c \in)Ch$ de *canale de comunicare*. Gărzile alternativelor nedeterminate sunt:

$$\vartheta ::= c!n \mid c?v$$

- Oferim un exemplu de program L_{syn} :

```

letrec  $y_1$  be
  if  $(0 < v)$  then ned [  $c!(v - 1) \rightarrow y_1$ 
                        |  $c!(v + 1) \rightarrow y_1$ 
                        |  $c!(v - 1) \rightarrow y_1$  ]
                    else skip
in letrec  $y_2$  be
  if  $(0 < v)$  then
    ned [  $c?v \rightarrow \text{write}(v) \cdot y_2$  ]
    else skip
in  $v := 3 \cdot \text{write}(v) \cdot (y_2 \parallel y_1 \parallel y_1 \parallel y_2)$ 

```

Acest program creează o rețea de 4 procese comunicante, care tind (în mod nedeterminist) să micșoreze valoarea variabilei v . Nu este dificil de văzut că programul se poate termina normal, dar se poate termina și în *deadlock*.

- Construcțiile $c!n$ și $c?v$ sunt ca în Occam [14].
- Execuția sincronă a două acțiuni $c!n$, $c?v$ care apar în procese paralele revine la transmiterea valorii expresiei n prin canalul c de la procesul care execută instrucțiunea de transmitere $c!n$ la procesul care execută instrucțiunea de recepționare $c?v$. Procesul care recepționează valoarea expresiei n o atribuie variabilei v . Intuitiv, acest mecanism implementează o *atribuire distribuită*.
- O construcție $\text{ned}[(\vartheta \rightarrow i)^*]$ conține un set de alternative nedeterminate gardate prin primitive de comunicare ($c!n$ sau $c?v$).
- Dacă se poate realiza sincronizarea cu procese concurente pe mai multe alternative, atunci se va alege în mod nedeterminist una dintre alternative.
- Un proces este suspendat atât timp cât sincronizarea cu procese paralele nu este posibilă.
- Un program ajunge în stare de blocare, sau *deadlock*, atunci când toate procesele paralele sunt suspendate pe tentative de comunicare nereușite.
- Un program se *termină* normal dacă toate procesele își finalizează activitatea.

- Implementăm sintaxa L_{syn} astfel:

```
data A = Assign V N | Write N
data X = Skip | Prefix A X | If B X X
      | Call Y | Letrec Y X X | Par X X
      | Ned [(C,X)]
```

```
data C = Snd Ch N | Rcv Ch V
type Ch = String
```

unde **Ch** este tipul canalelor de comunicație și **C** este tipul gărzilor alternativelor nedeterminate.

- Clasa instrucțiunilor este modificată în raport cu definiția corespunzătoare pentru L_{par} numai prin adăugarea alternativei **Ned [(C,X)]**.
- Tipurile **V**, **N**, **B** și **Y** sunt la fel ca pentru L_{par} .
- În cele ce urmează prezentăm numai extensiile și modificările ce trebuie să fie aduse definițiilor date în secțiunea 13.1 pentru limbajul L_{par} pentru a se obține un interpretor semantic pentru limbajul L_{syn} .

- Este mai întâi necesar să redefinim domeniul de liste de observabile pentru a reflecta posibilitatea de *blocare* (engl. *deadlock*) a programelor L_{syn} .

```
data Q = Empty | Deadlock
      | Observe Obs Q
```

- Instanțele **Eq** și **Show** atașate tipului **Q** sunt date în anexa C (**Obs** este un sinonim pentru tipul **Int**, `type Obs = Int`, la fel ca în secțiunea 11.4.3).
- Pentru a modela sincronizarea proceselor concurente extindem definiția domeniului de calcule **Comp** astfel:

```
data SemC = SemSnd Ch (S -> Int)
          | SemRcv Ch V
data Comp = Den D | Sync [(SemC, D)]
```

- Definițiile domeniilor pentru denotații (**D**), medii semantice (**Env**) și continuări (tipurile **Cont**, **Kont** și **PC**) rămân ca în secțiunea 13.1, dar domeniul continuărilor utilizează noul domeniu de calcule **Comp**. De asemenea mediul semantic inițial (**e0**) și continuarea inițială (**c0**) rămân ca în 13.1.

- Evaluatorul CSC pentru limbajului L_{syn} se poate obține din evaluatorul CSC al limbajului L_{par} prin adăugarea următoarei ecuații ce definește semantica construcției (**Ned gx**):

```

sem (Ned gx) e c = cc (addc p c)
  where
    p = Sync [(semC g, sem x e)
              | (g, x) <- gx]
    semC          :: C -> SemC
    semC (Snd ch e) = SemSnd ch (evN e)
    semC (Rcv ch v) = SemRcv ch v

```

Toate celelalte clauze ale funcției semantice **sem**, precum și funcția semantică **semA** și operatorii de control **addc** și **addp** rămân ca în secțiunea 13.1.

- De asemenea, funcția de execuție a continuării **cc** (inclusiv constanta **empty**) și procedura de normalizare **re** rămân ca în secțiunea 13.1.

- Planificatorul CSC pentru limbajul L_{syn} reutilizează numai definițiile funcțiilor **scheda** și **actc** date în secțiunea 13.1 pentru L_{par} . Domeniul planificărilor **Sched** și funcția planificator trebuie să fie modificate ca mai jos.

```

data Sched = Scheda D Cont
           | Scheds V (S -> Int) Kont

kc :: Kont -> P
kc k = case scheda k ++ scheds k of
      [] -> deadlock
      ws -> bigned (map exe ws)
  where exe (Scheda d c')      = d c'
        exe (Scheds v pe k') =
          \s -> kc k' (upd s v (pe s))

scheds :: Kont -> [Sched]
scheds k = [ w | (ps,pc) <- ms k,
                w <- sync [ps] pc ]

```

```

sync :: Kont -> Kont -> [Sched]
sync k1 k2 =
  [ Schedules v pe (re (addc (Den d1) c1) ++
                        re (addc (Den d2) c2))
  | (Sync snd, c1)          <- actc k1,
    (Sync rcv, c2)         <- actc k2,
    (SemSnd chs pe, d1)    <- snd,
    (SemRcv chr v, d2)     <- rcv,
    chs == chr ]

```

Reamintim că funcția **scheda** (dată în 13.1) calculează lista planificărilor de activare de forma **(Scheda d c)**. Planificatorul **kc** mai utilizează acum o funcție **schedules** ce calculează lista planificărilor de sincronizare de forma **(Schedules v pe k)**.

- Implementarea constantelor **empty** și **deadlock** depinde de tipul de semantică considerat: toate trasările posibile sau o singură trasare.

13.2.1 Specificare

- Pentru a obține un model semantic ce calculează toate trasările posibile pentru orice program L_{syn} se poate reutiliza tipul P așa cum a fost definit împreună cu operatorii `put`, `ned`, `bigned` și constanta `empty` în secțiunea 13.1.1. În acest caz constanta `deadlock` este:

```
deadlock :: P
deadlock = \s -> [Deadlock]
```

- Pentru testare considerăm următorul program L_{syn} :

```
x3 :: X
x3 =
  Par
    (Ned [(Snd "c" (Z 1),
           Ned [(Rcv "c" "v", Skip)]),
         (Snd "c" (Z 2),
           Prefix (Write (Z 3)) Skip)])
    (Ned [(Rcv "c" "v",
           Prefix (Write (V "v")) Skip)])
```

- Se poate efectua următorul experiment:

```
Main> sem x3 e0 c0 s0
[[1,deadlock],[3,2],[2,3]]
```

13.2.2 Prototip

- Pentru a obține un model semantic ce calculează o singură trasare aleasă în mod (pseudo-)aleator pentru orice program L_{syn} se poate reutiliza tipul P așa cum a fost definit împreună cu operatorii **put**, **ned**, **bigned** și constanta **empty** în secțiunea 13.1.2. În acest caz constanta **deadlock** este:

```
deadlock :: P
deadlock = \s r -> Deadlock
```

- Pentru testare considerăm următorul program L_{syn} :

```

x4 :: X
x4 =
  LetRec "y1"
    (If (Lt (Z 0) (V "v"))
      (Ned
        [(Snd "c" (Minus (V "v") (Z 1)),
          Call "y1"),
         (Snd "c" (Plus (V "v") (Z 1)),
          Call "y1"),
         (Snd "c" (Minus (V "v") (Z 1)),
          Call "y1")
        ])
      Skip)
    (LetRec "y2"
      (If (Lt (Z 0) (V "v"))
        (Ned [(Rcv "c" "v",
              Prefix (Write (V "v"))
                (Call "y2"))])
        Skip)
      (Prefix (Assign "v" (Z 3))
        (Prefix (Write (V "v"))
          (Par (Par (Call "y2")
                  (Call "y1"))
              (Par (Call "y1")
                  (Call "y2"))))))))

```


Constanta **x4** implementează programul dat în sintaxă abstractă la începutul secțiunii 13.2.

- Utilizând funcția **test** dată în secțiunea 13.1.2 se poate efectua următorul experiment:

```
Main> test x4 5
[3,4,3,2,1,0,deadlock]
[3,2,0,0]
[3,2,2,2,2,2,4,4,2,2,3,4,3,4,4,4,2,1,
 1,0,deadlock]
[3,2,4,4,4,4,3,4,2,3,4,4,3,1,0,0]
[3,2,0,0,deadlock]
```

Se remarcă faptul că prima, a treia și ultima trasare de execuție se termină în *deadlock*.

Partea IV

Semantica construcțiilor applicative.
Verificare tipuri.

Capitolul 14

Un limbaj de tip PCF

- În acest capitol studiem un limbaj aplicativ L_{pcf} de tip PCF¹ [23]. Prezentarea pornește de la [8, 36, 7].
- În esență, PCF extinde calculul λ simplu tipizat cu constante și operații peste tipuri primitive.
- Introducem L_{pcf} printr-o definiție BNF împreună cu reguli de deducție ce permit verificarea statică a corectitudinii tipurilor. Regulile de deducție au la bază pe triplete de forma:

$$u \vdash x : t$$

unde u este o *atribuire de tipuri* (noțiune ce va fi explicată imediat), x este un termen L_{pcf} și t este un tip.

- Un triplet $u \vdash x : t$ exprimă faptul că termenul x are tipul t în raport cu atribuirea de tipuri u .

¹Programming Language for Computable Functions

14.1 Sintaxa L_{pcf}

- Introducem sintaxa construcțiilor ($x(\in X)$) și tipurilor (t) L_{pcf} prin următoarea gramatică:

$$\begin{aligned}
 t & ::= \mathbf{bool} \mid \mathbf{nat} \mid t \rightarrow t \\
 x & ::= \mathbf{true} \mid 0 \\
 & \mid \mathbf{not}(x) \mid \mathbf{succ}(x) \mid \mathbf{pred}(x) \\
 & \mid x \mathbf{and} x \mid x == x \mid \dots \\
 & \mid i \mid \mathbf{if} x \mathbf{then} x \mathbf{else} x \mid \mathbf{let} i \mathbf{be} x \mathbf{in} x \\
 & \mid x x \mid \lambda i : t . x \mid \mu i : t . x
 \end{aligned}$$

- Explicații
 - Constantele și operațiile primitive au semantica intuitivă. Se pot introduce și alte constante și alți operatori peste tipurile primitive.
 - Tipul unui identificator i este definit în raport cu o atribuire de tipuri, iar semantica sa este definită în raport cu un mediu semantic.
 - Într-o construcție $x_1 x_2$ termenul x_1 denotă o funcție ce poate fi aplicată asupra valorii termenului x_2 .
 - Construcția $\lambda i : t . x$ reprezintă o abstracție lambda în care t este tipul identificatorului i .
 - Construcția $\mu i : t . x$ este folosită pentru definiții recursive, în care t este tipul identificatorului i .

- Implementăm sintaxa L_{pcf} în Haskell astfel:

```

type I = String

data T = BOOL | NAT | Fun T T
data X = TRUE | ZERO
      | Not X | Succ X | Pred X
      | And X X | Eq X X | Mul X X
      | I I | If X X X | Let I X X
      | Apply X X | Lambda (I,T) X
      | Mu (I,T) X

```

- Pentru a simplifica testarea interpretorului am introdus și construcția (**Mul** **x1** **x2**) pentru operația de înmulțire $x_1 * x_2$ (care nu apare în sintaxa abstractă). Desigur, se pot introduce și alte asemenea operații primitive.
 - Nu este însă dificil de văzut că operații precum adunarea sau înmulțirea se pot implementa prin definiții recursive pe baza operațiilor primitive **succ** și **pred**. De aceea, nu este uzual ca operații precum înmulțirea să fie predefinite în studiile semantice asupra PCF.
- Anexa D conține instanțele **Eq** și **Show** utilizate în acest capitol.

14.2 Verificarea tipurilor

- Fie $(i \in)I$ o clasă de *identificatori*. O *atribuire de tipuri* $u(\in U)$ este o listă $i_1 : t_1, \dots, i_n : t_n$ de perechi identificador-tip, în care identificadorii i_i sunt distincți. Conceptual, o atribuire de tipuri este o *funcție* ce mapează un domeniu (finit de) identificatori la tipuri. În Haskell punem:

```
type U = [(I,T)]
```

- Pentru atribuirile de tipuri definim o operație de "perturbare" ($u \mid i \mapsto t$), implementată aici de funcția `updu`, și o operație de evaluare $u(i)$, implementată de funcția `iu`.²

²`updu` este o simplă redenumire a funcției `upd` dată în anexa A; `iu` este o variantă a funcției `val` din anexa A.

```

updu :: Eq a =>
      [(a,b)] -> a -> b -> [(a,b)]
updu []           a b = [(a,b)]
updu ((a',b') : u) a b =
  if (a == a') then (a,b) : u
    else (a',b') : updu u a b

iu :: I -> U -> T
iu i [] =
  error ("Id. nelegat: " ++ i)
iu i ((i',t') : u) =
  if (i == i') then t' else iu i u

```

- Verificarea tipurilor expresiilor L_{pcf} se poate face pe baza unui set de reguli de forma [36, 8, 18]:

$$\frac{u_1 \vdash x_1 : t_1 \quad \cdots \quad u_n \vdash x_n : t_n}{u \vdash \cdots x_1 \cdots x_n \cdots : t}$$

- De exemplu, regula pentru conjuncția logică este (aici se utilizează aceeași atribuire de tipuri în premise și în concluzie, dar nu toate regulile respectă o asemenea disciplină):

$$\frac{u \vdash x_1 : \mathbf{bool} \quad u \vdash x_2 : \mathbf{bool}}{u \vdash x_1 \mathbf{and} x_2 : \mathbf{bool}}$$

- Regulile sunt proiectate astfel încât să atașeze un tip unic oricărei expresii L_{pcf} . În Haskell, specificăm aceste reguli de deducție printr-o funcție:

```
t :: X -> U -> T
```

De exemplu, specificăm regula pentru conjuncție astfel:

```
t (And x1 x2) u =
  case (t x1 u, t x2 u) of
    (BOOL, BOOL) -> BOOL
    _             -> err "And"
```


- Funcția auxiliară `err` este:

```
err :: String -> a
err s = error ("Eroare de tip: " ++ s)
```

- Oferim acum definiția completă a funcției `t` ce atașează un tip unic oricărei expresii L_{pcf} în raport cu o atribuire de tipuri dată.

```
t :: X -> U -> T
t TRUE u          = BOOL
t ZERO u          = NAT
t (Not x) u       =
  case t x u of
    BOOL -> BOOL
    _    -> err "Not"
t (Succ x) u      =
  case t x u of
    NAT -> NAT
    _   -> err "Succ"
t (Pred x) u      =
  case t x u of
    NAT -> NAT
    _   -> err "Pred"
t (And x1 x2) u   =
  case (t x1 u, t x2 u) of
    (BOOL, BOOL) -> BOOL
    _            -> err "And"
```

```

t (Eq x1 x2) u           =
  case (t x1 u,t x2 u) of
    (BOOL,BOOL) -> BOOL
    (NAT ,NAT ) -> BOOL
    _           -> err "Eq"
t (Mul x1 x2) u         =
  case (t x1 u,t x2 u) of
    (NAT,NAT) -> NAT
    _         -> err "Mul"
t (I i) u               = iu i u
t (If b x1 x2) u        =
  case (t b u,t x1 u,t x2 u) of
    (BOOL,t1,t2) ->
      if (t1 == t2)
        then t1
        else err "If"
    _             -> err "If"
t (Let i x1 x2) u       =
  t x2 (updu u i (t x1 u))
t (Apply x1 x2) u       =
  case (t x1 u,t x2 u) of
    (Fun t1 t1',t2) ->
      if (t1 == t2)
        then t1'
        else err "Apply"
    _                 -> err "Apply"

```

```
t (Lambda (i,ti) x) u =  
  Fun ti (t x (updu u i ti))  
t (Mu (i,ti) x) u      =  
  if (t x (updu u i ti) == ti)  
  then ti  
  else err "Mu"
```

Exerciții

- **E 14.2.1** *Să se descrie regulile implementate de funcția \mathfrak{t} utilizând notația abstractă:*

$$\frac{u_1 \vdash x_1 : t_1 \quad \cdots \quad u_n \vdash x_n : t_n}{u \vdash \cdots x_1 \cdots x_n \cdots : t}$$

14.3 Semantică denotațională

- Datorită recursivității, semantica L_{pcf} poate fi definită în mod riguros doar în cadrul matematic al teoriei domeniilor (vezi, de exemplu, [36, 8, 18]).
- Un *domeniu* este o structură matematică care descrie noțiunea de calcul și aproximare [32].
 - Dana Scott a introdus un element, \perp , pentru reprezentarea noțiunii de calcul ”nedefinit” sau ”divergent” (care nu se termină).
 - \perp este *cel mai mic element* al unui domeniu în abordarea clasică în care se operează cu conceptul de *cel mai mic punct fix* (al unei funcții continue).
 - În această abordare se utilizează adesea noțiunea de mulțime parțial ordonată completă (engl. *complete partial order* (cpo)³) ca sinonim pentru conceptul de *domeniu semantic*.

³Ideile de bază sunt următoarele (pentru mai multe informații cititorul poate consulta [36, 8, 18]):

1. O relație de ordine parțială ' \sqsubseteq ' peste o mulțime $(x, y, z) \in X$ este o relație:
 - reflexivă: $x \sqsubseteq x$
 - tranzitivă: $x \sqsubseteq y$ și $y \sqsubseteq z$ implică $x \sqsubseteq z$, și
 - antisimetrică: $x \sqsubseteq y$ și $y \sqsubseteq x$ implică $x = y$.
2. Foarte pe scurt, o mulțime parțial ordonată $(x \in)X$ este (ω -)completă dacă odată cu toate elementele unui (ω -)șir crescător $x_1 \sqsubseteq x_2 \cdots \sqsubseteq x_n \sqsubseteq \cdots$ conține și elementul *supremum* (engl. *least upper bound*) al șirului.

- De exemplu, dacă op_{syn} este un operator sintactic unar (cum este **succ**, **pred** sau **not**) atunci semantica expresiei $\text{op}_{\text{syn}}(x)$ este:

$$\llbracket \text{op}_{\text{syn}}(x) \rrbracket \eta = \begin{cases} \text{op}_{\text{sem}}(\llbracket x \rrbracket \eta) & \text{dacă } \llbracket x \rrbracket \eta \neq \perp \\ \perp & \text{dacă } \llbracket x \rrbracket \eta = \perp \end{cases}$$

- Se evaluează expresia x (în mediul semantic η) și dacă rezultatul este definit atunci asupra rezultatului se aplica operatorul semantic op_{sem} .
 - Dacă rezultatul evaluării expresiei x este nedefinit (\perp) atunci și rezultatul evaluării expresiei $\text{op}_{\text{syn}}(x)$ este nedefinit.
- În Haskell nu vom implementa explicit constanta \perp . Aceasta este o abstracție ce exprimă un calcul care nu se termină (un calcul divergent).
 - De exemplu, vom implementa semantica unei operații $\text{op}_{\text{syn}}(x)$ (în Haskell $(\text{OpSyn } x)$) în forma:

$$\text{sem } (\text{OpSyn } x) e = \text{opSem } (\text{sem } x e)$$

- Calculele care nu se termină pot fi modelate în Haskell pe baza mecanismului de evaluare leneșă. Dacă însă evaluarea semanticii $(\text{sem } x e)$ nu se termină atunci nici pentru expresia $(\text{OpSyn } x)$ nu se va putea calcula semantica.

- În absența recursivității, semantica tipurilor L_{pcf} ar putea fi definită (inductiv) astfel:

$$\llbracket \mathbf{bool} \rrbracket = \{true, false\}$$

$$\llbracket \mathbf{nat} \rrbracket = \mathbb{N}$$

$$\llbracket t_1 \rightarrow t_2 \rrbracket = \llbracket t_1 \rrbracket \rightarrow \llbracket t_2 \rrbracket$$

unde aici $\llbracket \cdot \rrbracket$ este mulțimea valorilor (interpretărilor) semantice de tipul '·'.

- O expresie de tip **bool** s-ar evalua la o valoare $\in \{true, false\}$,
- o expresie de tip **nat** s-ar evalua la o valoare $\in \mathbb{N}$,
- o expresie de tip **nat** \rightarrow **nat** s-ar evalua la o funcție $\in \mathbb{N} \rightarrow \mathbb{N}$, etc.

- În prezența recursivității, este necesar să se utilizeze *domenii* (iar nu mulțimi ordinare) pentru interpretarea semantică. Schițăm câteva idei:

- Orice mulțime $(m \in)M$ devine un cpo (M, \sqsubseteq) dacă este echipată cu relația de ordine discretă:

$$m_1 \sqsubseteq m_2 \iff m_1 = m_2.$$

- Pentru orice mulțime M se poate construi un cpo $M_\perp = (M \cup \{\perp\}, \sqsubseteq)$ în care

$$m_1 \sqsubseteq m_2 \iff (m_1 = m_2 \text{ sau } m_1 = \perp)$$

- Este posibil să se definească în mod riguros operații ce construiesc spații de funcții (\rightarrow), produse carteziene (\times) și sume ($+$) de domenii, rezultatele păstrând structura de domeniu (cpo).
- Domeniul pentru semantica denotațională a limbajului L_{pcf} poate fi definit (prin inducție după structura tipurilor t) astfel:

$$\llbracket \mathbf{bool} \rrbracket = \{true, false\}_\perp$$

$$\llbracket \mathbf{nat} \rrbracket = \mathbb{N}_\perp$$

$$\llbracket t_1 \rightarrow t_2 \rrbracket = \llbracket t_1 \rrbracket \rightarrow \llbracket t_2 \rrbracket$$

- După cum am explicat înainte, în implementarea Haskell nu există o reprezentare explicită pentru elementul cel mai mic al unui domeniu (\perp), divergența fiind modelată pe baza mecanismului de evaluare leneșă. Domeniul semanticii denotaționale a L_{pcf} poate fi implementat prin următoarea declarație de tip Haskell:

```
data D = B Bool | N Int | F (D -> D)
```

- În L_{pcf} semantica unui termen x depinde de tipul identificatorilor pe care îi conține. În general, semantica unui termen x se definește în raport cu o atribuire de tipuri u față de care x are tipul t : $u \vdash x : t$.
- Există o legătură naturală între mediile semantice (care mapează identificatori la valori) și atribuirile de tipuri (care mapează identificatori la tipuri).

- Spunem despre un mediu semantic η că este *compatibil* [36] cu o atribuire de tipuri u dacă pentru orice identificator i din domeniul lui u ($u = \dots, i : t, \dots$), $\eta(i) \in \llbracket u(i) \rrbracket (= \llbracket t \rrbracket)$.

– Notăm cu $(\eta \in) Env_u$ domeniul mediilor semantice compatibile cu atribuirea de tipuri u .

- Semantica denotațională poate fi definită pentru fiecare atribuire de tipuri u și tip t prin funcții $\llbracket \cdot \rrbracket_{ut}$ ce mapează termeni x pentru care $u \vdash x : t$ la funcții de la Env_u la $\llbracket t \rrbracket$:

$$\llbracket \cdot \rrbracket_{ut} : \{x \mid u \vdash x : t\} \rightarrow Env_u \rightarrow \llbracket t \rrbracket$$

- De exemplu, ecuația ce definește semantica abstracției μ este:

$$\llbracket \mu i : t. x \rrbracket_{ut} \eta = \text{fix}(\lambda d. \llbracket x \rrbracket_{(u|i \mapsto t)t}(\eta \mid i \mapsto d))$$

- Indicii u și t sunt însă adesea omiși din ecuațiile semantice.

Exerciții

- **E 14.3.1** *Să se descrie în forma $\llbracket x \rrbracket_{ut} \eta = \dots$ ecuațiile ce definesc semantica următoarelor construcții L_{pcf} :*

0	– constanta 0
$\text{succ}(x)$	– operația succesor
i	– identificator
$\lambda i : t . x$	– abstracție λ
$x_1 x_2$	– aplicare
$\mu i : t . x$	– abstracție μ

- Interpretorul semantic pentru limbajul L_{pcf} este proiectat în baza presupunerii că *verificarea tipurilor este realizată static* (înainte de execuție) printr-un apel al funcției \mathbf{t} (vezi definiția funcției \mathbf{exe}).
 - În consecință, indicii u și t nu apar în ecuațiile ce definesc funcția semantică \mathbf{sem} .
- Semantica denotațională a L_{pcf} utilizează operatorul \mathbf{fix} definit în secțiunea 12.4.
- Operatorul \mathbf{upde} (utilizat aici doar pentru ”perturbarea” mediilor semantice) coincide cu operatorul \mathbf{upd} definit în secțiunea 11.1.1, dar este redenumit pentru a nu fi confundat cu operatorul \mathbf{updu} (utilizat aici numai pentru atribuiri de tipuri).

```

upde :: Eq a =>
      (a -> b) -> a -> b -> (a -> b)
upde f a b a' =
  if (a' == a) then b else f a'

```

- Domeniul mediilor semantice este:

```

type Env = I -> D

```

- Implementăm semantica denotațională a L_{pcf} astfel (pentru definițiile matematice se poate consulta, de exemplu, [36, 8, 18]):

```

sem :: X -> Env -> D
sem TRUE          e = B True
sem ZERO         e = N 0
sem (Not x)      e = nots (sem x e)
sem (Succ x)     e = succs (sem x e)
sem (Pred x)     e = preds (sem x e)
sem (And x1 x2)  e =
  ands (sem x1 e) (sem x2 e)
sem (Eq x1 x2)   e =
  eqs (sem x1 e) (sem x2 e)
sem (Mul x1 x2)  e =
  muls (sem x1 e) (sem x2 e)
sem (I i)        e = e i
sem (If b x1 x2) e =
  ifs (sem b e) (sem x1 e) (sem x2 e)
sem (Let i x1 x2) e =
  sem x2 (upde e i (sem x1 e))
sem (Apply x1 x2) e =
  applys (sem x1 e) (sem x2 e)
sem (Lambda (i,ti) x) e =
  F (\d -> sem x (upde e i d))
sem (Mu (i,ti) x) e =
  fix (\d -> sem x (upde e i d))

```

- Operatorii semantici sunt:

```

nots :: D -> D
nots (B b) = B (not b)

```

```

succs :: D -> D
succs (N n) = N (n + 1)

```

```

preds :: D -> D
preds (N n) =
  if (n > 0) then N (n - 1) else N 0

```

```

ands :: D -> D -> D
ands (B b1) (B b2) = B (b1 && b2)

```

```

eqs :: D -> D -> D
eqs (B b1) (B b2) = B (b1 == b2)
eqs (N n1) (N n2) = B (n1 == n2)

```

```

muls :: D -> D -> D
muls (N n1) (N n2) = N (n1 * n2)

```

```

ifs :: D -> D -> D -> D
ifs (B b) d1 d2 = if b then d1 else d2

```

```

applies :: D -> D -> D
applies (F f) d = f d

```

- Pentru testarea interpretorului L_{pcf} mai utilizăm următoarele definiții:

```
exe :: X -> IO ()
exe x =
  do print (t x u0)
     print (sem x e0)
  where
    u0 :: U
    u0 = []
    e0 :: Env
    e0 i =
      error ("Id. nedefinit: " ++ i)

n :: Int -> X
n 0 = ZERO
n k = Succ (n (k-1))
```

- Fie acum:

```

x :: X
x =
  Apply
    (Mu ("factorial",Fun NAT NAT)
      (Lambda ("n",NAT)
        (If (Eq (I "n") ZERO)
          (n 1)
          (Mul (I "n")
              (Apply (I "factorial")
                    (Pred (I "n"))))))
      )
    )
  (n 5)

```

- Se poate efectua următorul experiment:

```

Main> exe x
NAT
120

```


Bibliografie

- [1] P. America. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing*, 1(4):366–411, 1989.
- [2] P. America, J.W. de Bakker, J.N. Kok, J.J.M.M. Rutten. Denotational semantics of a parallel object oriented language. *Information and Computation*, 83(2):152–205, 1989.
- [3] P. America and J.J.M.M. Rutten. Solving reflexive domain equations in a category of complete metric spaces. *Journal of Computer System Sciences*, 39:343–375, 1989.
- [4] J.W. De Bakker, E.P. De Vink. *Control flow semantics*. MIT Press, 1996.
- [5] S. Brookes. The essence of parallel Algol. *Information and Computation*, 179(1):118–149, 2002.
- [6] O. Danvy. On evaluation contexts, continuations and the rest of the computation. In H. Thielecke, editor, *Proceedings of the 4th ACM SIGPLAN Continuations Workshop (CW'04)*, pages 13–23, 2004.
- [7] M. Felleisen and M. Flat. *Programming languages and Lambda calculi*. Available from <http://www.cs.utah.edu/plt/publications/pllc.pdf>, 2006.
- [8] C. Gunter. *Semantics of programming languages: structures and techniques*. MIT Press, 1992.
- [9] C.A. Gunter, D.S. Scott. Semantic domains. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 633-674, 1990.
- [10] M. Hennessy, G.D. Plotkin. Full abstraction for a simple parallel programming language. In Proc. of 8th Symposium on Mathematical Foundations of Computer Science, LNCS 74, pages 108-120, 1979.
- [11] R. Hieb, R.K. Dybvig and C.W. Anderson. Subcontinuations. *Lisp and Symbolic Computation*, 7(1):83–110, 1994.
- [12] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21:667-677, 1978.
- [13] C.A.R. Hoare. *Communicating sequential processes*. Prentice Hall, 1985.

- [14] INMOS Ltd. *Occam programming manual*. Prentice Hall, 1984.
- [15] S. Mac Lane. *Categories for the working mathematician*. Springer, 1971.
- [16] R. Milner. *Communication and concurrency*. Prentice Hall, 1989.
- [17] R. Milner. *Communicating and mobile systems: the π calculus*. Cambridge Univ. Press, 1999.
- [18] J.C. Mitchell. *Foundations for programming languages*. MIT Press, 1996.
- [19] P.W. O’Hearn and R.D. Tennent. *Algol-like languages*. Birkhauser, 1997.
- [20] S. Peyton Jones and J. Hughes, editors. Report on the programming language Haskell 98: a non-strict purely functional language. Yale University, Department of Computer Science, Tech. Report YALEU/DCS/RR-1106, 1999.
- [21] B. Pierce. *Types and programming languages*. MIT Press, 2002.
- [22] G. Plotkin. A powerdomain construction. *SIAM Journal on Computing*, 5:522–587, 1976.
- [23] G. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223-255, 1977.
- [24] G. Plotkin. *The category of complete partial orders: a tool for making meanings*. Lecture notes for the Summer School on Foundations of Artificial Intelligence and Computer Science, Pisa, 1978.
- [25] G. Plotkin. A structural approach to operational semantics. Report DAIMI FN-19, Aarhus University, 1981.
- [26] G. Plotkin. *Pisa notes*, 1983.
- [27] V.R. Pratt. The pomset model of parallel processes. *Seminar on Concurrency 1984*, pages 180–196, LNCS 197, 1985.
- [28] D.S. Scott. Outline of a mathematical theory of computation. In *Proc. of 4th Annual Princetown Conference of Inf. Sciences and Systems*, pages 169–176, Princetown, 1970.
- [29] D.S. Scott. Data types as lattices. *SIAM Journal on Computing*, 5:522–587, 1976.
- [30] D.S. Scott and C. Strachey. Towards a mathematical semantics for computer languages. In *Proc. Symp. on Computers and Automata*, vol. 21 of *Microwave Research Institute Symposia Series*, pages 19–46, New York, 1971.
- [31] M.B. Smyth and G.D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal on Computing*, 11:761–783, 1982.

- [32] V. Stoltenberg-Hansen, I. Lindstrom and E.R. Griffor. *Mathematical theory of domains*. Cambridge University Press, 1994.
- [33] C. Strachey. Towards a formal semantics. In *Proc. IFIP TC2 Working Conference on Formal Language Description Languages for Computer Programming*, pages 198–220, Amsterdam, 1966.
- [34] C. Strachey. Fundamental concepts of programming languages. Lecture Notes for a NATO Summer School, Copenhagen, 1967.
- [35] C. Strachey and C. Wadsworth. Continuations: a mathematical semantics for handling full jumps. Tech. Monograph PRG-11, Programming Research Group, Univ. Oxford, 1974.
- [36] R.D. Tennent. *Semantics of programming languages*. Prentice Hall, 1991.
- [37] S. Thompson. *Haskell: the craft of functional programming*. Addison-Wesley, 1999.
- [38] E.N. Todoran. Metric semantics for synchronous and asynchronous communication: a continuation-based approach. In *Proceedings of FCT'99 Workshop on Distributed Systems, Electronic Notes in Theoretical Computer Science (ENTCS)*, 28:119–146, Elsevier, 2000.
- [39] E.N. Todoran, N. Papaspyrou. Continuations for parallel logic programming. In *Proc. of 2nd International ACM-SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'00)*, pages 257–267, ACM Press, 2000.
- [40] E.N. Todoran and N.S. Papaspyrou. Denotational prototype semantics for a simple concurrent language with synchronous communication. Technical Report CSD-SW-TR-1-04, National Technical University of Athens, School of Electrical and Computer Engineering, Software Engineering Laboratory, 2004.
- [41] E.N. Todoran and N.S. Papaspyrou. Continuations for prototyping concurrent languages. Technical Report CSD-SW-TR-1-06, National Technical University of Athens, School of Electrical and Computer Engineering, Software Engineering Laboratory, 2006.
- [42] E.N. Todoran. Comparative semantics for modern communication abstractions. In *Proc. of IEEE 4th International Conference on Intelligent Computer Communication and Processing (ICCP 2008)*, pages 153–161, 2008.
- [43] F. Turbak, D. Gifford. *Design concepts in programming languages*. MIT Press, 2008.

Anexa A

Definiții pentru starea ca listă de asociație

- Această anexă oferă implementarea operatorilor `val` și `upd` pentru reprezentarea conceptului de stare ca listă de asociație dată în secțiunea 11.1.2.
- Operatorul `val` este:

```
val :: V -> S -> Int
val v [] =
    error ("var. neinitializata" ++ v)
val v ((v',z'):s) =
    if (v == v') then z' else val v s
```

- `upd` poate fi implementat ca operator polimorfic:

```
upd :: Eq a =>
    [(a,b)] -> a -> b -> [(a,b)]
upd [] a b = [(a,b)]
upd ((a',b') : s) a b =
    if (a == a') then ((a,b) : s)
    else (a',b') : upd s a b
```

Anexa B

Definiții auxiliare pentru secțiunea 13.1

```
instance Eq Q where
  Empty      == Empty      =
  True
  Observe o1 q1 == Observe o2 q2 =
    (o1 == o2) && (q1 == q2)
  _          == _          = False
```

```
instance Show Q where
  show Empty = "[]"
  show q     = "[" ++ (aux q) ++ "]"
  where aux :: Q -> String
        aux (Observe o Empty) = show o
        aux (Observe o q)     =
          (show o) ++ "," ++ (aux q)
```

Anexa C

Definiții auxiliare pentru secțiunea 13.2

```
instance Eq Q where
  Empty          == Empty          = True
  Deadlock       == Deadlock       = True
  Observe o1 q1 == Observe o2 q2 =
    (o1 == o2) && (q1 == q2)
  _              == _              = False

instance Show Q where
  show Empty     = "[]"
  show Deadlock = "[deadlock]"
  show q        = "[" ++ (aux q) ++ "]"
  where aux :: Q -> String
        aux (Observe o Empty)    = show o
        aux (Observe o Deadlock) =
          (show o) ++ "," ++ "deadlock"
        aux (Observe o q)        =
          (show o) ++ "," ++ (aux q)
```

Anexa D

Definiții auxiliare pentru capitolul 14

```
instance Eq T where
  BOOL      == BOOL      = True
  NAT       == NAT       = True
  (Fun t1 t2) == (Fun t1' t2') =
    (t1 == t1') && (t2 == t2')
  _         == _         = False
```

```
instance Show T where
  show BOOL      = "BOOL"
  show NAT       = "NAT"
  show (Fun t1 t2) =
    "(Fun " ++ (show t1) ++
    " " ++ (show t2) ++)"
```

```
instance Show D where
  show (B b) = show b
  show (N n) = show n
  show (F f) = "{functie}"
```