Introducere în semantica limbajelor de programare

Note de curs şi seminar

Eneia Nicolae Todoran

-2016-

Cuprins

Partea I

Introducere

Capitolul 1

Concepte de bază

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

- Sunt utilizate două metode (discipline) consacrate în proiectarea semantică:
  - **semantica operaţională**
    - oferă o descriere intuitivă a modului de execuţie a programelor
    - are la bază conceptul de *sistem de tranziţie*
  - **semantica denotaţională**
    - construcţiile program *denotă* valori dintr-un domeniu matematic de interpretare
    - se operează cu *definiţii compoziţionale*
- În proiectarea maşinilor abstracte este utilizată *semantica operaţională structurată*.
- Sunt prezentate tehnici semantice: stări, continuări clasice şi continuări pentru concurenţă.
- Sunt oferite exemple de proiectare şi verificare matematică a modelelor semantice
- Sunt studiate concepte aplicative ce au la bază calculul lambda simplu tipizat [1, 2] (în stil PCF).
- Sunt prezentate aplicaţii complexe: interpretoare semantice pentru limbaje imperative, aplicative, orientate pe obiecte şi concurente.

- Precizări şi note bibliografice:
  - Referinţele de bază pentru semantica operaţională structurată sunt [3, 4]. Semantica operaţională este un instrument de bază în proiectarea limbajelor şi sistemelor formale, instrument preferat în prezent în majoritatea aplicaţiilor.
  - Materialul legat de semantica operaţională se bazeaza pe [5]. Modelele semantice din [5] sunt adaptate prin construirea de interpretoare (prototipuri declarative) Prolog şi Haskell.
  - În majoritatea aplicaţiilor, domeniile matematice utilizate în semantica denotaţională sunt mulţimi parţial ordonate [7, 6] sau spaţii metrice [5].
  - În aceste note de curs tehnicile semantice sunt ilustrate cu precadere in proiectarea limbajelor imperative sau funcţionale. Se subliniaza însă că semantica operaţională şi denotaţională sunt utilizate in domenii diverse, inclusiv: calcul paralel şi distribuit [8, 9, 4, 10, 11, 12], programare orientată obiect [13, 14, 15, 16, 17, 18, 19], programare logică [20, 21, 22, 23, 24, 25, 26], calcul natural [27, 28, 29, 30], standarde de specificare [36, 37, 38, 39], etc.
  - Principiile semanticii denotaţionale au fost in-

troduse la sfârşitul anilor '60 şi începutul anilor '70 [40, 41, 42, 43, 44] utilizând domenii clasice. Abordarea metrică a fost iniţiată în [45, 48].

– Problema stabilirii unei relaţii optime intre semantica denotaţională si semantica operaţională, nunmită abstractizare totală (engl. *full abstractness*) a fost formulată de Robin Milner [46, 47].
– Materialul despre calculul lambda simplu tipizat şi PCF este din [49, 50, 57].
– Tehnica clasică a semanticii de continuare a fost introdusă ca instrument pentru semantica denotaţională în [51].
– Semantica de continuare pentru concurenţă a fost introdusă în [11, 25] şi dezvoltată in [12, 52, 53, 54].
– În cursurile avansate de limbaje se utilizează adesea următoarele monografii [49, 50, 56, 5, 57, 60, 61].

Capitolul 2

Semantică operaţională

- În semantica operaţională execuţia unui program este modelată sub forma unei secvenţe de configuraţii

  $c_1, c_2, ..., c_n, ...$

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

  – De exemplu, o configuraţie poate fi o instrucţiune program, sau o pereche constând dintr-o instrucţiune program împreună cu starea curentă.

- Adesea, se operează cu tranziţii etichetate de forma: $c_i \stackrel{a_i}{\rightarrow} c_{i+1}$.

  – Eticheta de tranziţie ar putea fi o acţiune atomică sau starea curentă.

2.1 Sisteme de tranziţie

- Începând cu [3, 4], abordarea preferată în majoritatea aplicaţiilor este *semantica operaţională structurată* (SOS).
- SOS are la bază conceptul de *sistem de tranziţie*.
  - Introducem maşinăria semanticii operaţionale direct pentru cazul sistemelor de tranziţie etichetate (engl. *labeled transition system*) LTS.
  - Sistemele neetichetate se obţin cu uşurinţă prin specializare.
- Un LTS este un triplet $(C, A, \rightarrow)$ constând dintr-o mulţime[1] $(c \in)C$ de *configuraţii*, o mulţime $(a \in)A$ de *etichete de tranziţie* şi o *relaţie de tranziţie* $\rightarrow \subseteq C \times A \times C$.
  - În acest caz faptul că tripletul (c, a, c') este un element al relaţiei de tranziţie $((c, a, c') \in \rightarrow)$ se exprimă de obicei sub forma $c \xrightarrow{a} c'$.

[1] În această lucrare notaţia $(x, y, ... \in)X$ introduce mulţimea X cu elemente tipice $x, y, ...$ (care iau valori din X).

- În abordarea SOS relaţia de tranziţie '$\rightarrow$' este indusă de o *specificare de sistem de tranziţie* (engl. *transition system specification*) TSS, care este o colecţie de *reguli* de forma:

$$\frac{c_1 \xrightarrow{a_1} c_1' \qquad \cdots \qquad c_n \xrightarrow{a_n} c_n'}{c \xrightarrow{a} c'}$$

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

  - Tranziţiile date sub formă de axiome au loc prin definiţie.
  - În plus, orice tranziţie care este concluzia unei reguli este un element al relaţiei de tranziţie dacă se poate stabili în baza regulilor TSS că premisele sale sunt adevărate.

- Relaţia de tranziţie specificată printr-un TSS este *cea mai mică* relaţie care satisface regulile TSS.

- **Observaţii**
  - În unele aplicaţii nu sunt necesare tranziţii etichetate. În acest caz
    - relaţia de tranziţie '$\rightarrow$' este o submulţime a $C \times C \quad (\rightarrow \subseteq (C \times C))$,
    - iar regulile de deducţie sunt de forma:

$$\frac{c_1 \rightarrow c_1' \qquad \cdots \qquad c_n \rightarrow c_n'}{c \rightarrow c'}$$

  - În majoritatea aplicaţiilor o configuraţie este
    - fie un element a limbajului (o construcţie sintactică reprezintând instrucţiunea curentă a maşinii abstracte de calcul),
    - fie o pereche constând dintr-un element al limbajului împreună cu starea curentă a maşinii.
  - Termenul *structurat* din SOS se referă la proiectarea ghidată de sintaxă a regulilor TSS.
- Termenul *semantică* denotă o funcţie $\mathcal{S}$ de forma:

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

unde L este mulţimea construcţiilor limbajului dat, iar $\mathbb{D}$ este un domeniu matematic de interpretare. Funcţia semantică operaţională se defineşte direct pe baza un sistem de tranziţie pentru limbajul considerat, aşa cum se va vedea din exemple

2.2 Semantică operaţională în Prolog

- Un sistem de tranziţie este un sistem de deducţie alcătuit din axiome şi reguli ce permit specificarea unei relaţii de tranziţie. Deducerea tranziţiilor se face, la fel ca în limbajul Prolog, prin înlănţuire înapoi (engl. *backward chaining*).
- Este deci naturală transcrierea unui sistem de tranziţie 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:

$$\frac{c_1 \xrightarrow{a_1} c_1' \qquad \cdots \qquad c_n \xrightarrow{a_n} c_n'}{c \xrightarrow{a} c'}$$

  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 şi seminar este utilizată o notaţie de tip Turbo Prolog (sau Visual Prolog), ce include o secţiune pentru declaraţii de tipuri.
- Un program include patru secţiuni distincte, delimitate prin cuvintele cheie: `domains` (declaraţii de tip), `predicates` (declaraţii predicate cu tipul argumentelor), `clauses` (fapte şi reguli pentru predicatele Prolog) şi `goal`.
- Declaraţiile de tip permit evidenţierea domeniilor sintactice utilizate în specificarea unui sistem de tranziţie şi specificarea structurii matematice a universului semantic corespunzător.

2.3 Semantică operaţională în Haskell

- Limbajul Haskell [62] este un limbaj (pur) functional. Pentru facilitaţile de calcul simbolic şi funcţii de ordin superior este adesea utilizat ca metalimbaj în proiectarea semantică, în special în proiectarea denotaţională.
- În aceste note de curs şi seminar vom oferi şi implementări Haskell ale semanticilor operaţionale.
- Faţă de Prolog, limbajul Haskell are dezavantajul că exprimă cu mai multă dificultate un sistem de deducţie utilizat în specificarea unui sistem de tranziţie.
  - În particular, colecţia tuturor tranziţiilor 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.
- Totuşi, prin manipularea funcţiilor de ordin superior limbajul Haskell poate oferi avantaje în prototipizarea semantică, ştiut fiind că modelele semantice utilizează adesea funcţii de ordin superior.

Capitolul 3

Semantică denotaţională

- Prin definirea unei semantici denotaţionale pentru un limbaj de programare se ataşează valori matematice fiecărei construcţii program şi astfel se obţine o definiţie precisă a limbajului de programare.
- O asemenea definiţie permite verificarea corectitudinii diverselor implementări ale limbajului şi poate fi utilizată în demonstrarea proprietăţilor programelor.
- În aceste note de curs şi seminar termenul *semantică* denotă o funcţie $\mathcal{S}$ de forma:

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

  unde L este mulţimea construcţiilor limbajului dat, iar $\mathbb{D}$ este un domeniu matematic de interpretare.

  – În reprezentarea acestor funcţii se utilizează adesea 'parantezele semantice': $[\![\cdot]\!] : L \to \mathbb{D}$.

3.1 Principiile semanticii denotaţionale

1. *Construcţiile program denotă valori dintr-un domeniu matematic de interpretare.*
2. *Definiţiile denotaţionale sunt compoziţionale: semantica unei construcţii sintactice compuse este exprimată ca funcţie de semantica construcţiilor sintactice componente.*

$$[\![\cdots s_1 \cdots s_n \cdots]\!] = \cdots [\![s_1]\!] \cdots [\![s_n]\!] \cdots$$

Realizarea unei semantici denotaţionale cuprinde următoarele activităţi:

- definirea unui domeniu matematic de interpretare;
- proiectarea unei funcţii compoziţionale care mapează construcţiile sintactice la valori din domeniul de interpretare.

3.2 Domenii semantice

- Un *domeniu* este o structură matematică care descrie noţiunea de calcul şi aproximare [63].
  - Un calcul efectuat pe baza unui algoritm se realizează în paşi discreţi.
  - După fiecare pas se obţine mai multă informaţie despre rezultatul calculului.
  - Rezultatul obţinut 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 mulţime echipată cu o anumită *structură* (o relaţie de ordine parţială sau o metrică).
- Domeniile compuse sunt obţinute prin utilizarea câtorva construcţii de bază: produsul cartezian, reuniunea (disjunctă), spaţiul de funcţii, domeniile putere (mulţimi de submulţimi).

- Domeniile semantice trebuie să permită reprezentarea proceselor de calcul perpetue; acest tip de comportament apare în prezenţa recursivităţii.
- În abordarea clasică, domeniile sunt mulţimi parţial ordonate complete şi se operează cu funcţii continue [6, 49, 56].
- O altă abordare cunoscută utilizează spaţii metrice complete şi contracţii peste spaţii metrice [5].
- Comportamentul perpetuu (infinit) se obţine ca limită a unor procese de calcul finite.

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

$$\mathbb{X} = E$$

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

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

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

  - spaţiu de funcţii:
    $E_1 \to E_2$
  - domeniu putere (mulţimea submulţimilor):
    $\mathcal{P}(E)$
  - 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
$(1, (2, (3, Nil)))$
$(1, (1, (1, ...)))$

- Metoda consacrată de rezolvare a ecuaţiilor de domeniu se bazează pe teoria categoriilor [65], atât în cazul domeniilor clasice [66] cât şi în cazul spaţiilor metrice [67].

3.3 Semantică denotaţională în Haskell

- În cadrul acestor note de curs şi seminar, în locul notaţiei matematice este utilizat adesea limbajul Haskell [62] pentru exprimarea modelelor denotaţionale [25, 17, 19, 69].
  - Semantica denotaţională este prin natura sa o semantică funcţională în care se operează cu definiţii compoziţionale.
  - Limbajul Haskell oferă în mod natural suport pentru implementarea tehnicilor majore utilizate în semantica denotaţională: stări, medii semantice, continuări, etc.
- Se au în vedere următoarele beneficii:
  - se evită dificultăţile legate de prezentarea sistematică a aparatului matematic utilizat în teoria clasică a domeniilor sau în sematica metrică;
  - modelele denotaţionale devin interpretoare prototip pentru limbajele considerate şi pot fi imediat evaluate şi testate.

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

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

- Sintaxa limbajelor studiate este dată prin definiţii de forma:

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

- Semantica este definită prin funcţii compoziţionale

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

unde

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

Partea II

Semantică operaţională

Capitolul 4

Secvenţiere şi recursivitate

- În acest capitol este studiat un limbaj foarte simplu $\overline{L}_1$, care aproximează nucleul de fluenţă al limbajelor secvenţiale.
- Prezentăm sintaxa şi semantica operaţională a $\overline{L}_1$, şi introducem anumite tehnici prin care se poate raţiona asupra comportamentului programelor şi a proprietăţilor limbajului $\overline{L}_1$.
- Oferim o implementare Prolog a semanticii operaţionale
- Oferim o implementare Haskell a semanticii operaţionale

4.1 Limbajul $\overline{L}_1$: sintaxă şi sistem de tranziţie

- Consideră̆m date
  - o mulţime $(a \in) Act$ de *acţiuni atomice*. Aici A este un alfabet de simboluri elementare, neinterpretate. Aceste simboluri sunt reprezentări abstracte ale unor elemente de bază din limbajele de programre, cum sunt instrucţiunile de atribuire, sau expresiile;
    - Limbajele care abstractizează acţiunile atomice sub formă de simboluri atomice dintr-un alfabet dat se zic *uniforme* [5]
  - o mulţime $(y \in) Y$ de *variabile de procedură*; intuitiv, apariţia unei variabile de procedură în textul unui program $\overline{L}_1$ este un apel recursiv. Aceste mulţimi vor fi utilizate şi în capitolele următoare.
- Recursivitatea este tratată pe baza conceptelor de *declaraţie* şi *instrucţiune gardată*, urmând abordarea din [5]
- Intuitiv, într-o instrucţiune gardată orice apel recursiv se poate realiza doar după execuţia a cel putin o instrucţiune atomică.

- **Sintaxa** $\overline{L}_1$

  (a) Instrucţiuni:

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

  (b) Instrucţiuni gardate:

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

  (c) Declaraţii:

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

  (d) Programe:

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

- **Configuraţiile** TSS_1 **pentru** $\overline{L}_1$[1]

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

  (adică $\quad Conf = \{E\} \cup X$)

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

- **Sistemul de tranziţie** TSS_1 **pentru** $\overline{L}_1$ **[5]**

  Relaţia de tranziţie $\rightarrow \subseteq Conf \times Act \times Decl \times Conf$ cu elemente (r, a, D, r') scrise în notaţia $r \stackrel{a}{\rightarrow}_D r'$ este cea mai mică relaţie care satisface următorul set de axiome şi reguli. În particular, asta înseamnă că pentru configuraţia E nu există nici o tranziţie.

(Act)

$$a \stackrel{a}{\rightarrow}_D E$$

(Rec)

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

(Seq)

$$\frac{x_1 \stackrel{a}{\rightarrow}_D E}{x_1; x_2 \stackrel{a}{\rightarrow}_D x_2}$$

$$\frac{x_1 \stackrel{a}{\rightarrow}_D x_1'}{x_1; x_2 \stackrel{a}{\rightarrow}_D x_1'; x_2}$$

- **Mulţime succesor şi complexitate [5]**
  - Pentru orice sistem de tranziţie etichetat TSS, dat ca un triplet $(Conf, Act, \rightarrow)$, şi pentru orice $c \in Conf$, se defineşte *mulţimea succesor* $S(c)$ pentru configuraţia c astfel:

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

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

$$\begin{aligned} c(D, E) &= 0 \\ c(D, a) &= 1 \\ c(D, y) &= 1 + c(D, D(y)) \\ c(D, x_1; x_2) &= 1 + c(D, x_1) \end{aligned}$$

4.2 Exerciţii

E 4.2.1 *Fie $D \in Decl$, astfel încât $D(y_1) = a_1, D(y_2) = a_2$. Utilizând regulile sistemului de tranziţie TSS_1 să se deducă (tranziţia sau) tranziţiile următoarelor programe $\overline{L}_1$: $(y_1; a2)$ şi $((y_2; a_3); a_4)$.*

E 4.2.2 *Să se arate că funcţia (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 paşi. Se arată mai întâi că $c(D, g) \in \mathbb{N}$, prin inducţie structurală[2] pentru orice instrucţiune gardată $g \in G$; de remarcat că $G \subseteq X$. Apoi se tratează cazul general, şi se arată că $c(D, x) \in \mathbb{N}$, tot prin inducţie structurală, după structura lui x.*

[2]Adică prin inducţie după structura sintactică a lui g.

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

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

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

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

[3]În această lucrare prin $|M|$ notăm **cardinalul** mulţimii M.

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

(a) Instrucţiuni:

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

(b) Instrucţiuni gardate:

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

(c) Declaraţii:

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

(d) Programe:

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

Constructiile pentru recursivitate şi secvenţiere sunt o adaptare a conceptelor introduse în contextul limbajului $\overline{L}_1$. $n \in N$ *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* ($\in \mathbb{Z}$)*, şi se poate considera că* $z \in \mathbb{Z}$.

Se presupune că semantica expresiilor este definită printr-o funcţie de evaluare dată
$\mathcal{N}[\![\cdot]\!] : N \to \Sigma \to \mathbb{Z}$,

$$\mathcal{N}[\![\, z \,]\!](\sigma) = z$$
$$\mathcal{N}[\![\, v \,]\!](\sigma) = \sigma(v)$$
$$\mathcal{N}[\![\, n_1 + n_2 \,]\!](\sigma) = \mathcal{N}[\![\, n_1 \,]\!](\sigma) + \mathcal{N}[\![\, n_2 \,]\!](\sigma)$$
$$\ldots$$

$(\sigma \in)\Sigma$ *este o mulţime de* stări *ale maşinii, modelate ca funcţii de la variabile la valori:*

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

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

Dacă $f : X \to Y$ *este o funcţie de la mulţimea* $(x \in)X$ *la mulţimea* $(y \in)Y$, *atunci* $(f \mid x \mapsto y) : X \to Y$ *este o funcţie (cu acelaşi domeniu şi 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 funcţiei* f *(doar) în punctul* x.

Se cere:

- *Să se proiecteze un sistem de tranziţie* $\mathrm{TSS}_{1'}$ *care să specifice comportamentul instrucţiunilor pentru* $\overline{L}_1'$. *Configuraţiile* $\mathrm{TSS}_{1'}$ *vor fi de forma* $(r, \sigma) \in (Conf \times \Sigma)$, *unde*

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

- *Să se arate că* $\mathrm{TSS}_{1'}$ *este determinist. Raţionamentul se va efectua prin inducţie după o măsură de complexitate adecvată,* $c : Conf \to \mathbb{N}$, $c(E) = 0$, $c(v := n) = 1$, $c(y) = 1 + c(D(y))$ *şi* $c(x_1; x_2) = 1 + c(x_1)$.

4.3 Funcţia semantică operaţională

- **Univers semantic**

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

unde

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

adică Act^∞ conţine toate secvenţele finite (Act^*) şi infinite (Act^ω) de elemente din Act. Notăm prin ϵ *secvenţa vidă* (aici $\epsilon \in Act^\infty$).

- **Semantică operaţională pentru $\overline{L}_1$ [5]**

$\mathcal{O}[\![\cdot]\!] : \overline{L}_1 \to \mathbb{P}_O$, $\mathcal{O}[\![(D, x)]\!] = \mathcal{O}_D(x)$, unde funcţia $\mathcal{O}_D : Conf \to \mathbb{P}_O$ este definită astfel:

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

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

4.4 Prototip Prolog

- Se utilizează o notaţie 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 declaraţie:

```
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
```

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

Se mai consideră următoarea declaratie:

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

Aceasta declaraţie exprimă că avem $D(y_3) = a; y_3$.

Pentru a putea testa acest program $\overline{L}_1$ 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 obţine o secvenţă (conceptual) infinită,[4] iar `T` nu va fi afişat niciodată):

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

[4]Practic, programul poate produce 'heap overflow'.

4.5 Prototip Haskell

- În implementarea Haskell se face uz de facilitatea de lucru cu funcţii de ordin superior.
- Sintaxa $\overline{L}_1$

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

- Configuraţii

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

- Univers semantic

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

- Implementarea Haskell a TSS_1 (se calculează toate tranziţiile posibile pentru fiecare configuraţie)

```
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) ]
```

- Funcţia semantică operaţională '`os`'
  - Apelul '`tss`' va returna întotdeauna o listă de lungime exact 1 conform exerciţiului 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 declaraţie:

```
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",...
```

[5]Datorită faptului că limbajul Haskell utilizează un mecanism de evaluare leneşă ultima evaluare *nu* va produce depăşire de memorie.

4.6 Exerciţii

E 4.6.1 *Pornind de la sistemul de tranziţie* $\mathrm{TSS}_{1'}$ *construit în exerciţiul 4.2.4 se cere:*

- *Să se definească semantica operaţională (ca funcţie matematică) pentru limbajul* $\overline{L}_1'$*, definit în exerciţiul 4.2.4.*
- *Să se implementeze semantica operaţională a limbajului* $\overline{L}_1'$ *în Prolog sau Haskell.*

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

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

Construcţiile $v := n, y$ *şi* $x; x$ *sunt ca la* $\overline{L}_1'$.
$(b \in)B$ *este o clasă de* expresii booleene, *pentru care se consideră o sintaxă de forma*

$$b \ ::= \ true \ \mid \ \cdots \ \mid \ n > n \ \mid \ \cdots \ \mid \ b \,\textsf{and}\, b \ \mid \ \cdots$$

La execuţia unei instrucţiuni 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 instrucţiunilor* $\overline{L}_1$" *se va proiecta în raport cu o mulţime* $\sigma \in \Sigma (= V \to \mathbb{Z})$ *de* stări.
Se presupune că semantica expresiilor booleene este definită printr-o funcţie de evaluare dată $\mathcal{B}[\![\cdot]\!] : B \to \Sigma \to \{true, false\}$, *de forma:*

$$\mathcal{B}[\![true]\!](\sigma) = true$$
$$\mathcal{B}[\![n_1 > n_2]\!](\sigma) = \mathcal{N}[\![n_1]\!](\sigma) > \mathcal{N}[\![n_2]\!](\sigma)$$
$$\mathcal{B}[\![b_1 \,\textsf{and}\, b_2]\!](\sigma) = \mathcal{B}[\![b_1]\!](\sigma) \,\textsf{and}\, \mathcal{B}[\![b_2]\!](\sigma)$$

Se cere:

- *Să se proiecteze un sistem de tranziţie* $\mathrm{TSS}_{1"}$ *care să specifice comportamentul instrucţiunilor pentru* $\overline{L}_1$". *Configuraţiile* $\mathrm{TSS}_{1"}$ *vor fi de forma* $(r, \sigma) \in (Conf \times \Sigma)$, *unde*

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

- *Să se arate că* $\mathrm{TSS}_{1"}$ *este determinist. Se va efectua un raţionament inductiv, după o măsură de complexitate adecvată* $c : Conf \to \mathbb{N}$. *Pentru instrucţiunea condiţională se poate pune:*

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

De asemenea, pornind de la sistemul de tranziţie $\mathrm{TSS}_{1"}$ *se cere:*

- *Să se definească semantica operaţională (ca funcţie matematică) pentru limbajul* $\overline{L}_1$".
- *Să se implementeze semantica operaţională a limbajului* $\overline{L}_1$" *în Prolog sau Haskell.*

Capitolul 5

Semantică de continuare pentru $\overline{L}_1$

- În semantica de continuare un program este conceptual împărţit într-o instrucţiune curentă şi restul programului; o *continuare* este o reprezentare a acestui *rest* de calcul [51]
- O continuare poate fi privită drept *context de evaluare* pentru instrucţiunea sau expresia curentă [70]
- Această noţiune a fost introdusă în contextul semanticii denotaţionale dar conceptul este utilizat şi în semantica operaţională

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

- Sintaxa $\overline{L}_1$ a fost introdusă în capitolul 4.
- **Configuraţiile** sistemului de tranziţie proiectat în semantică de continuare pentru limbajul $\overline{L}_1$ [5] sunt date prin:

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

Convenţie În cele ce urmează se suprimă declaraţia din construcţiile semantice, presupunându-se, fără pierdere în generalitate, o declaraţie $D(\in Decl)$ dată (cf. [5]). Această convenţie este utilizată şi în capitolele următoare din partea II.

– În loc de

$$r \stackrel{a}{\rightarrow}_D r'$$

scriem doar

$$r \stackrel{a}{\rightarrow} r'$$

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

Exerciţiu

E 5.0.3 *Să se rescrie sistemul de tranziţie* TSS_1 *(din cap. 4) utilizând această convenţie de notaţie.*

5.1 Prototip Prolog

- Sintaxa şi universul semantic pentru limbajul $\overline{L}_1$ sunt implementate la fel ca în capitolul 4.
- Configuraţiile sunt adaptate pentru semantica de continuare, conform definiţiei 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 reprezinta prototipul Prolog al sistemului de tranziţie proiectat în semantică de continuare pentru limbajul $\overline{L}_1$.
- Predicatul `os/2` implementează funcţia semantică operaţională.

- Predicatul `tss/3` dat mai jos reprezinta prototipul Prolog al sistemului de tranziţie proiectat în semantică de continuare pentru limbajul $\overline{L}_1$.
- Predicatul `os/2` implementează funcţia semantică operaţională.

```
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 următoare declaraţie:

```
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 şi în prototipul Haskell este utilizată convenţia că se operează cu o declaraţie fixată, fără a se pierde din generalitate (cf. [5]).
  - În implementarea Haskell acest aspect este mai evident, deoarece aici declaraţia (care este o funcţie) ar putea fi transmisă ca parametru, aşa cum s-a procedat în implementarea Haskell din capitolul 4.
- Implementarea Haskell a sintaxei şi a universului semantic al limbajului $\overline{L}_1$ 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 = []
```

- Configuraţiile sunt specifice semanticii de continuare.

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

- Funcţia `tss` este un prototip declarativ ce specifică un sistem de tranziţie proiectat cu continuări pentru limbajul $\overline{L}_1$. `os` implementează semantica operaţională a limbajului $\overline{L}_1$.

```
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 declaraţie.

```
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, şi acest interpretor Haskell permite testarea programelor $\overline{L}_1$ care nu se termină, în baza mecanismului de *evaluare leneşa* a limbajului Haskell [62]. De exemplu, dacă declaraţia 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 tranziţie proiectat cu continări pentru limbajul $\overline{L}_1$ se poate raţiona prin inducţie utilizând următoarea măsură de complexitate [5]: $c : Conf \to \mathbb{N}, c_x : X \to \mathbb{N}$,

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

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

5.3 Exerciţii

E 5.3.1 *Să se arate că funcţiile c şi c_x sunt bine definite.*

E 5.3.2 *Pe baza prototipurilor declarative prezentate în acest capitol să se construiască sistemul de tranziţie* TSS_{1con} *(axiome şi reguli în semantică de continuare) pentru limbajul $\overline{L}_1$. (Desigur, se utilizează convenţia de suprimare a declaraţiei D din regulile* TSS_{1con}*.)*

E 5.3.3 *Să se rescrie* TSS_{1con} *utilizând următoare convenţie de notaţie:*

$$r \to r' \quad \text{este o abreviere pentru} \quad \frac{r' \xrightarrow{a} \overline{r}}{r \xrightarrow{a} \overline{r}}$$

Intuitiv, în această notaţie $r \to r'$ este o rescriere a configuraţiei r ca şi r' pentru a putea fi ulterior prelucrată.

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

E 5.3.5 *Să se definească funcţia semantică operaţională bazata pe* TSS_{1con} *pentru $\overline{L}_1$.*

E 5.3.6 *Să se proiecteze o semantică de continuare pentru limbajul neuniform $\overline{L}_1$" introdus în exerciţiul 4.6.2.*

- *Se va proiecta un sistem de tranziţie* TSS_{1con}" *care să specifice comportamentul instrucţiunilor pentru $\overline{L}_1$". Configuraţiile* 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 paşii obişnuiţi. Se va alege o măsură de complexitate adecvată pentru raţionamentul inductiv. Se va utiliza o pereche de funcţii $c : Conf \to \mathbb{N}$ şi $c_x : X \to \mathbb{N}$, la fel ca în cazul sistemului de tranziţie* TSS_{1con}*. Se va arăta că funcţiile $c : Conf \to \mathbb{N}$ şi $c_x : X \to \mathbb{N}$ sunt bine definite. Apoi se va arăta că* TSS_{1con}" *este determinist şi se va defini funcţia semantică operaţională bazată pe* TSS_{1con}" *pentru $\overline{L}_1$". În final, se va implementa întreg modelul matematic în Haskell sau Prolog.*

Capitolul 6

Nedeterminism

- Un program este nedeterminist dacă la execuţii diferite poate produce rezultate diferite chiar dacă de fiecare dată porneşte din aceeaşi stare iniţială şi primeşte aceleaşi date de intrare
- În acest capitol este studiat un limbaj $\overline{L}_2$, care extinde limbajul $\overline{L}_1$ cu un operator '+' de alegere nedeterministă.
- Intuitiv, execuţia unei instrucţiuni $x_1 + x_2$ revine la o alegere aleatoare între două alternative: se execută fie instrucţiunea x_1 fie instrucţiunea x_2, nu amândouă.
- Semantic, trebuie să se înregistreze toate rezultatele (sau execuţiile) posibile, şi astfel se ajunge la conceptul de *domeniu putere* (engl. *power domain*), concept introdus de Gordon Plotkin [71].

- Conceptul de domeniu putere este analogul semantic al conceptului de mulţime de submulţimi (engl. *power set*).
- De obicei, dacă X este o mulţime, *mulţimea tuturor submulţimilor lui* X se notează prin $\mathcal{P}(X)$.
- Teoriile semantice (bazate pe mulţimi parţial ordonate sau spaţii metrice) nu se pot dezvolta însă pentru $\mathcal{P}(X)$. De obicei se impun condiţii suplimentare asupra constructorului $\mathcal{P}(\cdot)$.
- Folosim notaţia $\mathcal{P}_{prop}(\cdot)$ pentru a denota mulţimea tuturor submulţimilor lui '$\cdot$' care au proprietatea *prop*.

  De exemplu, $\mathcal{P}_{finit}(X)$ este mulţimea tuturor submulţimilor finite ale mulţimii X

- Acest capitol utilizează metode şi tehnici prezentate în [5], unde semantica limbajelor este studiată în cadrul matematic al spaţiilor metrice complete. Noţiunile legate de spaţii metrice nu le disctăm în mod sistematic. Cititorul interesat poate consulta, de exemplu, monografia [5].
- În semantica metrică, în locul construcţiei $\mathcal{P}(\cdot)$, se utilizează adesea construcţii de forma $\mathcal{P}_{compact}(\cdot)$ sau $\mathcal{P}_{nevid\_compact}(\cdot)$, adică se operează cu *submulţimi compacte*, respectiv nevide şi compacte.[1]

[1]Poate fi util să facem câteva precizări legate de noţiunea de *submulţime compactă*.

– Dacă (M, d) este un spaţiu metric (M este o mulţime iar d este o metrica definită pe M), o submulţime X a lui M se zice *compactă*, dacă orice secvenţă din X are o subsecvenţă convergentă cu limita în X.

– Într-o caracterizare alternativă, X se zice compactă dacă este limita unei secvenţe de submulţimi finite ale lui M, limita fiind calculată în raport cu aşa-numita *metrică Hausdorff*.

– Aparatul formal al semanticii metrice este dezvoltat în cadrul matematic al spaţiilor metrice complete, unde constructorul $\mathcal{P}_{compact}(\cdot)$ poate da naştere unui spaţiu complet; în general însă, constructorul $\mathcal{P}(\cdot)$ sau constructorul $\mathcal{P}_{finit}(\cdot)$ nu va da naştere unui spaţiu complet.

6.1 Limbajul $\overline{L}_2$

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

  (a) Instrucţiuni:

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

  (b) Instrucţiuni gardate:

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

  (c) Declaraţii:

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

  (d) Programe:

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

- Semantica operaţională a limbajului $\overline{L}_2$ poate fi definită pe baza unui sistem de tranziţie TSS_2, proiectat în semantică directă (adică *nu* în semantică de continuare), ce operează cu configuraţii de forma:

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

6.2 Prototip Prolog

- Oferim aici un prototip Prolog, ce specifică semantica operaţională a limbajului $\overline{L}_2$.
- Sintaxa instrucţiunilor (`x`), configuraţiile (`r`) şi universul semantic (`p`) pentru limbajul $\overline{L}_2$ sunt date în secţiunea `domains`.
- În cazul $\overline{L}_2$ un element al universului semantic `p` este o mulţime (implementată ca listă) de trasări. O trasare este un element de tip `q`, adică o secvenţă (implementată ca listă) de acţiuni atomice.
- Tipurile `succ` şi `succ_set` permit utilizarea predicatului `findall/3` în definiţia `os/2`.
- Tipul `succ_set` permite implementarea conceptului de *mulţime succesor*.

```
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 funcţia semantică operaţională, mapând instrucţiuni de tip `x` la valori ale universului semantic `p`.
- Predicatul `os/2` utilizează predicatul `osr/2` ce operează pe configuraţii de tip `r`.
- Predicatul `tss/2` specifică sistemul de tranziţie pentru semantica directă a limbajului $\overline{L}_2$
  - Vom numi acest sistem de tranziţie 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 preferinţa de a realiza o implementare a nedeterminismului bazată pe predicatul `findall/3`.
  - Nedeterminismul necesar generării tuturor rezultatelor[2] poate fi implementat şi prin mecanismul implicit de backtracking din Prolog.
  - Totuşi, soluţia bazată pe utilizarea `findall/3` permite colectarea tuturor rezultatelor (adică a tuturor trasărilor de execuţie) sub forma unei liste de tip `p`.

[2]În acest model semantic fiecare rezultat este, de fapt, o trasare de execuţie.

- Predicatul `prefix/3` prefixează o acţiune atomică la fiecare trasare de tip `q` dintr-o colecţie de trasări de tip `p`.
- Predicatele `union/3` şi `bigunion` implementează operaţii de reuniune binară, respectiv n-ară.[3]
- Predicatul `exe/2` produce o listă de execuţii ale tuturor perechilor din mulţimea succesor; perechile succesor sunt primite într-o listă de tip `succ_set`.
  - În definiţia semanticii operaţionale toate aceste execuţii 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)
```

[3]Nu 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(X11))).
   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 definiţia `osr/2` pentru calculul mulţimii succesor pentru o configuraţie $x \in X$: $S(x) = \{(a, r) \mid x \xrightarrow{a} r\}$; rezultatul apelului `findall/3` este produs în varabila `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 şi colectează într-o listă toate execuţiile $a \cdot \mathcal{O}(r)$.
  - Apoi este utilizat predicatul `bigunion/2` pentru a calcula reuniunea tuturor acestor execuţii cu producerea unei valori de tipul `p`.

- Predicatul `tss/2` specifică sistemul de tranziţie TSS_2. Clauzele pentru aţiuni atomice, recursivitate şi compunere secvenţială sunt la fel ca în capitolul 4.
- Cele doua clauze pentru alegerea nedeterministă pot fi scrise în notaţie matematică astfel:[4]

(Ned)

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

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

[4]Reamintim că în continuare se utilizează în mod sistematic convenţia de suprimare a declaraţiei $D(\in \mathit{Decl})$ din regulile şi definiţiile semantice.

- Pentru testare se utilizează următoarea declaraţie.

```
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 declaraţie D, pentru care: $D(y_1) = a_1 + a_2; y_2$, $D(y_2) = a_3; y3 + a_4; y_4$, $D(y_3) = a_5 + a_6; y_4$ şi $D(y_4) = a_7$.
- 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 matematica, acest rezultat corespunde următoareai colecţii 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 mulţimea configuraţiilor sistemului de tranziţie TSS_2 ce specifică comportamentul programelor $\overline{L}_2$.

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

- Tipul `P` implementează universul semantic pentru semantica operaţională.
  - La fel ca în cazul prototipului Prolog, un element al domeniului final `P` este o colecţie (mulţime) de trasări de execuţie; în Haskell acest concept este implementat ca listă împreună cu operatori corespunzători de manipulare mulţimi: **`prefix`** (prefixează o acţiune atomică la fiecare secvenţă dintr-o colecţie), **`union`** (reuniune binară) şi **`bigunion`** (reuniune n-ară).
    - În Haskell aceşti operatori sunt implementaţi în mod natural ca funcţii polimorfice
  - Fiecare trasare (secvenţă) este implementată ca o listă Haskell de tipul `Q`; **`epsilon`** este secvenţa 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
```

- Funcţia `tss` specifică sistemul de tranziţie TSS_2, ce defineşte semantica construcţiilor limbajului $\overline{L}_2$.
- Tehnic, funcţia `tss` calculează mulţimea succesor pentru fiecare configuraţie $\overline{L}_2$.
- În cazul alegerii nedeterministe, programul concatenează mulţimile 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 apariţii ale unora dintre perechile succesor `(Act,R)`, acestea vor produce aceeaşi semantică operaţională, iar trasările duplicate sunt eliminate prin utilizarea operatorului de reuniune mulţimi în definiţia funcţiei semantică operaţională `os`.
- Funcţia `os` specifică semantică operaţională utilizând o notaţie aproapiată de notaţia 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 declaraţie:

```
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 Exerciţii

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

E 6.4.2

- *Dacă un* TSS *are proprietatea că pentru orice configuraţie* $c \in Conf$ *mulţimea succesor* $S(c)$ *este finită* ($|S(c)| < \infty$)*, se spune că* TSS *este cu* ramificare finită.
- *Să se arate că sistemul de tranziţie* TSS_2 *este cu ramificare finită.*
- **Sugestie** *Se poate proceda prin inducţie după următoarea măsură de complexitate:*
  $c : Conf \to \mathbb{N}$, $c(E) = 0$, $c(a) = 1$,
  $c(y) = 1 + c(D(y))$, $c(x_1; x_2) = 1 + c(x_1)$,
  $c(x_1 + x_2) = 1 + max\{c(x_1), c(x_2)\}$.
  Desigur, trebuie arătat mai întâi că funcţia c *este bine definită.*

Observaţii

– Utilizând formalismul semanticii metrice [5] se poate utiliza următorul univers semantic pentru semantica operaţională a limbajului $\overline{L}_2$:[5]
$\mathbb{P}_O = \mathcal{P}_{nevid\_compact}(Act^\infty)$

– Semantica operaţională poate fi definită astfel:
$\mathcal{O}[\![\cdot]\!] : \overline{L}_2 \to \mathbb{P}_O,\ \mathcal{O}[\![s]\!] = \mathcal{O}(s)$,
unde $\mathcal{O} : Conf \to \mathbb{P}_O$,

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

∗ Reamintim că '$\cdot$' este utilizat în aceste note de curs şi seminar ca operator de concatenare secvenţe.

∗ Aici acest operator realizează prefixarea unei acţiuni atomice fiecărei secvenţe dintr-o mulţime (nevidă şi compactă).

– Există o teoremă care spune că, dacă un TSS este cu ramificare finită, atunci semantica operaţională definită ca mai sus returnează o mulţime compactă. Mai precis, în acest caz, $\forall r \in Conf$, $\mathcal{O}(r) \in \mathbb{P}_O = \mathcal{P}_{nevid\_compact}(Act^\infty)$.

[5] Elementele $\mathbb{P}_O$ sunt submulţimi nevide şi compacte ale Act^∞. Noţiunea de *submulţime compactă* a fost descrisă pe scurt la începutul acestui capitol. Pentru mai multe informaţii cititorul poate consulta, de exemplu, [5].

Exerciţiu

E 6.4.3 *Pentru următoarele programe* $\overline{L}_2$ *să se calculeze semantica operaţională*

(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 declaraţie* $D \in Decl$ *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 configuraţii de forma:

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

- Este oferită specificarea Prolog şi Haskell a semanticii operaţionale proiectată în semantică de continuare pentru limbajul $\overline{L}_2$.
  - Exerciţiile date la sfârşitul acestui capitol sugerează modul în care se poate construi întreg modelul matematic.
- În cele ce urmează numim *sistemul de tranziţie* ce defineşte comportamentul limbajului $\overline{L}_2$ în semantică de continuare TSS_{2con}.

7.1 Prototip Prolog

- Declaraţiile de domenii sunt la fel ca în capitolul 6, exceptând tipul utilizat pentru specificarea configuraţiilor. Pentru a evita orice ambiguitate reproducem aici declaraţiile 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` şi `bigunion/2` rămâne ca în capitolul 6.
- Celelalte definiţii de predicate sunt date mai jos.[1]

[1]Strict vorbind, şi predicatul `exe/3` rămâne ca în capitolul 6. Repetăm totuşi aici şi definiţia 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` şi `osr/2` specifică funcţia semantică operaţională utilizând aceeaşi abordare ca în capitolul 6.
- Predicatul `tss/2` oferă o definiţie compactă a semanticii de continuare a limbajului $\overline{L}_2$.
- Din nou, pentru testare se utilizează următoarea declaraţie.

```
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 definiţiile operatorilor semantici de lucru cu mulţimi `union` şi `bigunion`, şi operatorul `prefix` de prefixare a unei acţiuni atomice la o colecţie de trasări; aceşti operatori au fost definiţi în capitolul 6.
- Se utilizează aceeaşi implementare 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
```

- Dar configuraţiile sunt adaptate la semantica de continuare.

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

- Şi produsul final al funcţiei semantice rămâne tot ca în capitolul 6, un domeniu `P` de colecţii de trasări de execuţie, fiecare trasare fiind o secvenţă de tip `Q` de acţiuni atomice. **`epsilon`** este secvenţa vidă.

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

- Funcţia `os` este specificaţia Haskell a semanticii operaţionale.

```
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 şi aici o exprimare foarte concisă a semanticii limbajului $\overline{L}_2$. Funcţia `tss` reprezintă specificaţia Haskell a sistemului de tranziţie 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 declaraţie:

```
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 Exerciţii

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

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

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

E 7.3.3 *Funcţia semantică operaţională* $\mathcal{O}[\![\,\cdot\,]\!] : \overline{L}_2 \to \mathbb{P}_O$ *(*$\mathbb{P}_O = \mathcal{P}_{nevid\_compact}(Act^\infty)$*) poate fi definită pe baza regulilor* TSS_{2con} *astfel* $\mathcal{O}[\![\,x\,]\!] = \mathcal{O}(x : E)$*, unde* $\mathcal{O} : Conf \to \mathbb{P}_O$ *este*

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

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

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

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

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

Capitolul 8

Concurenţă şi nedeterminism

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

$$\{a_1a_2\,,\ a_2a_1\}$$

- În această abordare compunerea concurentă este interpretată pe baza conceptului de nedeterminism cu execuţia întreţesută a acţiunilor elementare din componentele concurente.

8.1 Limbajul $\overline{L}_3$

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

  (a) Instrucţiuni:

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

  (b) Instrucţiuni gardate:

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

  (c) Declaraţii:

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

  (d) Programe:

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

- Semantica operaţională a limbajului $\overline{L}_2$ poate fi definită pe baza unui sistem de tranziţie TSS_3 ce operează cu configuraţii de forma:

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

8.2 Prototip Prolog

- Se specifică semantica operaţională a limbajului $\overline{L}_3$ prin construirea unui prototip Prolog.
- Domeniile utilizate în definirea sintaxei şi a universului semantic sunt ca în capitolul 6, exceptând tipul `x`, care include acum o construcţie 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*
```

- Declaraţiile de semnătură ale predicatelor rămân ca în secţiunea 6.2, de aceea nu mai sunt repetate aici.
- De asemenea, predicatele `bigunion`, `union`, `elem` şi `prefix` rămân ca în secţiunea 6.2.

- Semantica operaţională este specificată în maniera obişnuită, prin predicatele `os/2`, `osr/2` şi `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 tranziţie 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 acţiuni atomice, recursivitate, alegere nedeterministă şi compunere secvenţială sunt ca în capitolul 6.
- Clauzele pentru compunerea paralelă pot fi scrise în notaţie 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 declaraţie:

```
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 notaţie matematică declaraţia de mai sus poate fi descrisă astfel:

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

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

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

- Definiţia semanticii operaţionale $\mathcal{O}[\![\cdot]\!]$ este discutată în secţiunea de exerciţii 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 mulţimea configuraţiilor sistemului de tranziţie TSS_3 ce specifică comportamentul programelor $\overline{L}_3$.

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

- Tipul `P` implementează universul semantic pentru semantica operaţională a limbajului $\overline{L}_3$.
- Tipul `P` specifică un domeniu de mulţimi de secvenţe (trasări) de execuţie şi este implementat (împreună cu operatorii **`bigunion`**, **`union`** şi **`prefix`**) la fel ca în secţiunea 6.3.
- De asemenea, fiecare trasare (secvenţă) este implementată ca o listă Haskell de tipul `Q`; **`epsilon`** este secvenţa vidă.

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

- Funcţia `tss` dată mai jos specifică sistemul de tranziţie TSS_3, ce defineşte semantica construcţiilor limbajului $\overline{L}_3$.
  - Această definiţie extinde funcţia `tss` dată în secţiunea 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) ]
```

- Funcţia `os` specifică semantică operaţională a limbajului $\overline{L}_3$.

```
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 declaraţie:

```
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 Exerciţii

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

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

– **Sugestie** *Se poate proceda prin inducţie după următoarea măsură de complexitate:*
$c : Conf \to \mathbb{N}$, $c(E) = 0$, $c(a) = 1$,
$c(y) = 1 + c(D(y))$, $c(x_1; x_2) = 1 + c(x_1)$,
$c(x_1 + x_2) = 1 + max\{c(x_1), c(x_2)\}$ *şi*
$c(x_1 \| x_2) = 1 + max\{c(x_1), c(x_2)\}$.
Se va arăta mai întâi că funcţia c este bine definită.

E 8.4.3 *Semantica operaţională a limbajului* $\overline{L}_3$ *poate fi definită astfel:* $\mathcal{O}[\![\,\cdot\,]\!] : \overline{L}_3 \to \mathbb{P}_O$, $\mathbb{P}_O = \mathcal{P}_{nevid\_compact}(Act^\infty)$, $\mathcal{O}[\![\,x\,]\!] = \mathcal{O}(x)$ *iar* $\mathcal{O} : Conf \to \mathbb{P}_O$ *este:*

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

unde definiţia $\mathcal{O}(\cdot)$ *este dată în raport cu* TSS_3. *Ca de obicei, semantica operaţională este bine definită deoarece* TSS_3 *este cu ramificare finită. Utilizând definiţia matematică a semanticii operaţionale şi a* TSS_3 *să se calculeze:*

(a) $\mathcal{O}[\![\,a_1 \parallel a_2\,]\!]$

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

Capitolul 9

Semantică de continuare pentru concurenţă

- Tehnica clasică a continuărilor [51] furnizează *flexibilitate* în proiectarea semantică, însă continuările clasice prezintă limitări în modelarea sistemelor paralele sau distribuite [72].
- În acest capitol prezentăm o semantică de continuare pentru concurenţă, în limba engleză *continuation semantics for concurrency* (CSC) [11, 25, 12, 52].
- În abordarea CSC o continuare este o structură de calcule (instrucţiuni, în semantica operaţională), ce se pot evalua fie în secvenţă, fie în paralel.
- Structura continuărilor CSC este reprezentativă pentru conceptele de control studiate. În acest capitol se arată cum poate fi utilizat conceptul de *multiset* pentru modelarea compunerii paralele [25, 74, 34], într-un limbaj concurent simplu $\overline{L}_4$.

- In general, structura unei continuări CSC poate fi mai complexa; de exemplu, atunci când compunerea paralelă se combină cu un operator general de compunere secvenţială, o continuare CSC poate avea structura de stivă cactus, anume o stiva cu mai multe varfuri active concurent [55].

9.1 Conceptul de multiset

- Conceptul de multiset a fost introdus de N.G. De Bruijn, în 1970.
- În teoria mulţimilor, un *multiset* peste o mulţime $(x \in)X$ este o pereche (X, m), unde $m : X \to \mathbb{N}$ este o funcţie de la mulţimea X la mulţimea numerelor naturale $\mathbb{N}$ (m se numeşte *funcţie 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.
- **Observaţie:** 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 mulţime, unde un element poate apărea o singură dată.
- Uneori, elementele unui multiset se reprezintă între paranteze '$\{\!|$' şi '$|\!\}$'. 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 şi 5 apar câte odată. Dacă mulţimea suport este $\mathbb{N}$, atunci

    $\{\!|2, 3, 2, 2, 5|\!\} = (\mathbb{N}, m)$, unde
    $m(2) = 3, m(3) = 1, m(5) = 1$ şi
    $m(x) = 0, \forall x \in \mathbb{N}, x \notin \{2, 3, 5\}$.

- Se pot defini **operaţii peste multiseturi**:
  - Reuniune:
    $(X, m_1) \cup (X, m_2) = (X, \lambda\, x\, .\, max\{m_1(x), m_2(x)\})$
  - Sumă:
    $(X, m_1) \uplus (X, m_2) = (X, \lambda\, x\, .\, m_1(x) + m_2(x))$
  - Intersecţie:
    $(X, m_1) \cap (X, m_2) = (X, \lambda\, x\, .\, min\{m_1(x), m_2(x)\})$
  - Diferenţă:
    $(X, m_1) \setminus (X, m_2) =$
    $(X, \lambda\, x\, .\, max\{0, m_1(x) - m_2(x)\})$
  - Incluziune:
    $(X, m_1) \subseteq (X, m_2) =$
    $(\forall x \in X : m_1(x) \leq m_2(x))$
  - Test apartenenţă (pentru $x \in X$):
    $x \in (X, m) = (x \in X) \wedge (m(x) > 0)$
  - Cardinalitatea unui multiset:
    $|(X, m)| = \Sigma_{x \in X} m(x)$

- În semantica de continuare pentru concurenţă conceptul de multiset este utilizat în mod natural pentru modelarea execuţiei concurente a proceselor, din fiecare proces putând exista una sau mai multe copii active.
- În general, se utilizează contînări CSC structurate. Structura continuărilor poate fi definită pe baza unei relaţii de ordine parţială [11, 73, 75]. O structură înrudită este cea de *pomset* [76].
- În acest capitol este studiat un limbaj concurent simplu, pentru care conceptul de multiset (nestructurat) este suficient.
- Fie $(x \in)X$ o mulţime. Se utilizează notaţia:

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

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

9.2 Limbajul $\overline{L}_4$

- Limbajul $\overline{L}_4$ este asemănător cu $\overline{L}_3$, dar utilizează o formă mai particulară de compunere secvenţială, bazată pe prefixare de acţiuni atomice/elementare, utilizată, de exemplu, în CCS [77] şi calculul π [78].
- În $\overline{L}_4$ operaţia de secvenţiere are forma $a \cdot x$, operandul stâng al operatorului ' $\cdot$ ' fiind întotdeauna o acţiune atomică.

  (a) Instrucţiuni:

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

  (b) Instrucţiuni gardate:

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

  (c) Declaraţii:

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

  (d) Programe:

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

9.3 Continuări şi configuraţii pentru $\overline{L}_4$

- Restricţia la operatorul de prefixare acţiuni 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 (concurent) studiat.
  - De exemplu, în cazul unui limbaj cum este $\overline{L}_3$ (ce combină compunerea paralelă cu un operator general de compunere secvenţială), continuările CSC sunt structuri arborescente (matematic, se utilizează multiseturi echipate prin relaţii de ordine parţială [11, 73]).

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

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

  unde X este mulţimea instrucţiunilor $\overline{L}_4$.

- O configuraţie $\rho(\in Conf)$ a sistemului de tranziţie TSS_{4csc} ce defineşte semantica instrucţiunilor $\overline{L}_4$ este fie o continuare, fie o pereche constând dintr-o instrucţiune $\overline{L}_4$ şi o continuare:

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

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

9.4 Sistemul de tranziţie TSS_{4csc} pentru $\overline{L}_4$

Relaţia de tranziţie $\rightarrow$ pentru $\overline{L}_4$ este cea mai mică submulţime a $Conf \times Act \times Cont$ ce satisface următoarele axiome şi 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)

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

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

(Sched)

$$r \rightarrow (x, r \setminus \{\!|x|\!\}) \qquad \forall x \in r$$

- Structura de multiset a continuărilor este necesară deoarece o continuare poate include mai multe copii ale oricărei instrucţiuni.
- În definiţia TSS_{4csc} se utilizează convenţia de notaţie introdusă în exerciţiul 5.3.3.
- Din regulile TSS_{4csc} rezultă că pentru configuraţia $\{\!|\}$ nu există nici o tranziţie. $\{\!|\}(\in Cont)$ este continuarea vidă (multisetul vid).
- În axiomele (Act) şi (Prefix) se execută o acţiune atomică a; în axioma (Prexif) instrucţiunea 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, aşa cum s-a procedat în exerciţiul 7.3.1.

- Regulile (Par) şi (Sched) sunt specifice tehnicii CSC. În regula (Par) se alege în mod nedeterminist una dintre cele două instrucţiuni de executat în paralel spre a rămâne activă iar cealaltă se adaugă la continuare. Orice instrucţiune rămâne activă doar până când execută o acţiune atomică. Apoi se planifică (se activează) o altă instrucţiune utilizând regula (Sched). Astfel se obţine execuţia întreţesută (*interleaving semantics*) a instrucţiunilor concurente.
- Intuitiv, tehnica CSC este o formalizare semantică a unui planificator de procese [11].

9.5 Exerciţii

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

– **Sugestie** *Se poate proceda în doi paşi. Mai întâi se demonstrează că dacă $(x, r) \in X \times Cont$ şi $(x, r) \xrightarrow{a} \rho'$ atunci $\rho' \in Cont$, prin inducţie după $c(x)$, unde $c : X \to \mathbb{N}$, $c(a) = 1$, $c(a \cdot x) = 1, c(y) = 1 + c(D(y))$ şi $c(x_1 + x_2) = c(x_1 \parallel x_2) = 1 + max\{c(x_1), c(x_2)\}$. Apoi, utilizând rezultatul obţinut în primul pas se arată că dacă $r \in Cont$ şi $r \xrightarrow{a} \rho'$ atunci $\rho' \in Cont$, utilizând regula (Sched).*

– *Desigur, trebuie să se arate că funcţia $c : X \to \mathbb{N}$ este bine definită (în doi paşi, mai întâi pentru instrucţiunile gardate $\in G$, apoi pentru toate instrucţiunile $\in X$).*

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

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

E 9.5.3 *Semantica operaţională a limbajului* $\overline{L}_4$ *poate fi definită astfel:* $\mathcal{O}[\![\,\cdot\,]\!] : \overline{L}_4 \to \mathbb{P}_O$, $\mathbb{P}_O = \mathcal{P}_{nevid\_compact}(Act^\infty)$, $\mathcal{O}[\![\,x\,]\!] = \mathcal{O}(x, \{\!|\,|\!\})$ *iar* $\mathcal{O} : Conf \to \mathbb{P}_O$ *este:*

$$\mathcal{O}(\{\!|\,|\!\}) = \{\epsilon\}$$
$$\mathcal{O}(\rho) = \bigcup\{a \cdot \mathcal{O}(r) \mid \rho \xrightarrow{a} r\} \quad \text{if } \rho \neq \{\!|\,|\!\}$$

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

Utilizând definiţia matematică a semanticii operaţionale şi 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 declaraţie* $D(\in Decl)$ *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 secvenţială este înlocuit aici cu operatorul de prefixare acţiuni atomice pentru care este utilizat mai jos functorul `prefix`.
- Celelalte definiţii de domenii sunt ca în capitolul 8. Diferenţa principală apare însă la domeniul pentru configuraţii, care este implementat prin tipul `conf` utilizând tipul `r` care implementează mulţimea 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 instrucţiune activabilă şi o continuare ce conţine restul instrucţiunilor din continuarea iniţială.

```
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` şi `prefix/3` sunt implementate ca în secţiunea 6.2. Predicatul `tss/2` dat mai jos implementează sistemul de tranziţie TSS_{4csc}, iar predicatul `os/2` implementează funcţia semantică operaţională $\mathcal{O}[\![\cdot]\!]$ pentru limbajul $\overline{L}_4$. Predicatul `tss/2` utilizează predicatul `ms` pentru a realiza operaţia de planificare specificată în clauza (Sched) a TSS_{4csc}. Celelalte clauze ale predicatului `tss/2` implementează regulile (Act), (Prefix), (Rec), (Ned) şi (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 declaraţie.

```
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 $\overline{L}_4$.

```
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 $\overline{L}_3$.

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

- Configuraţiile `Conf` utilizate în implementarea sistemului de tranziţie TSS_{4csc} sunt specifice semanticii de continuare pentru concurenţă (tehnica CSC). Tipul `R` implementează mulţimea continuărilor CSC, care, în cazul limbajului $\overline{L}_4$, sunt liste (multiseturi) de instrucţiuni ce se evaluează în paralel.

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

- Semantica operaţională a limbajului $\overline{L}_4$ este implementată de funcţia `os`. Funcţiile `bigunion` şi `prefix` au fost introduse în secţiunea 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 ]
```

- Funcţia `tss` ce implementează sistemul de tranziţie TSS_{4csc} utilizează funcţia `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)
```

- Funcţia `tss` implementează sistemul de tranziţie TSS_{4csc}.
- `concat` este o funcţie 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 ]
```

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

- Pentru testare se utilizează următoarea declaraţie:

```
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 denotaţională

Capitolul 10

Modele semantice pentru un limbaj uniform

- Se consideră un limbaj secvenţial (nerecursiv) cu acţiuni elementare (atomice) neinterpretate L_0.
- Un limbaj cu acţiuni elementare neinterpretate se zice *uniform* [5].
- Fie $(a \in) Act$ o mulţime de *acţiuni atomice*. Definim sintaxa L_0 în BNF[1]. Clasa $(x \in) X$ a instrucţiunilor L_0 este introdusă prin gramatica:

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

  unde '$x_1; x_2$' reprezintă compunerea secvenţială a instrucţiunilor x_1 şi 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
```

[1] Backus-Naur Formalism

10.1 Semantică directă

- Interpretarea semantică a programelor L_0 o dăm prin secvenţe finite peste Act. Domeniul semantic pt. L_0 este[2]:

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

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

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

$$\begin{aligned}
&[\![\cdot]\!] : X \to \mathbb{D} \\
&[\![a]\!] = a \\
&[\![x_1; x_2]\!] = [\![x_1]\!] \cdot [\![x_2]\!]
\end{aligned}$$

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

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

- Semantică denotaţională 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 experimente[3]:

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

[3]Experimentele au fost realizate utilizând interpretorul Hugs, disponibil la adresa `http://www.haskell.org`.

Exerciţii

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

10.2 Semantică de continuare

- Tehnica continuărilor reprezintă un instrument deosebit de util pentru modelarea controlului în semantica denotaţională. Tehnica clasică a semanticii de continuare a fost introdusă în [51].
- În această abordare, funcţia denotaţională depinde de un parametru adiţional, numit *continuare*.
- Un program este descompus conceptual într-o instrucţiune *curentă* şi *restul* programului.
- Conform definiţiei originale [51], o *continuare* este este o reprezentare semantică a acestui *rest* de program.

- Prezentăm o semantică denotaţională pentru L_0 proiectată cu tehnica clasică a continuărilor [51].
- 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:

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

$$[\![\cdot]\!] : X \to \mathbb{D}$$
$$[\![a]\!] \gamma = a \cdot \gamma$$
$$[\![x_1; x_2]\!] \gamma = [\![x_1]\!] ([\![x_2]\!] \gamma)$$

- unde operatorul '$\cdot$' de concatenare secvenţe este utilizat aici pentru prefixarea unui element la o secvenţă, rezultatul fiind o nouă secvenţă (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 funcţie semantică trebuie să definim o *continuarea iniţială* `c0`. În acest caz, continuarea iniţială este secvenţă vidă de observabile (acţiuni atomice).

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

- Fie `x0, x1 :: X` ca în secţiunea 10.1. Se pot realiza următoarele evaluări:

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

Exerciţii

- **E 10.2.1** *Să se arate că semantica de continuare (funcţia* `sem` *din definiţia de mai sus) este definită compoziţional.*
- **E 10.2.2** *Să se arate că operaţia de compunere secvenţială este asociativă şi în cazul semanticii de continuare.*
- **E 10.2.3** *Să se arate că pentru orice* `x :: X` *şi pentru orice* `c :: Cont`*:*

  ```
  semc x c = semd x ++ c
  ```

  unde `semd` *este semantica directă definită în secţiunea 10.1 iar* `semc` *este semantica de continuare definită în această secţiune.*[4] *În particular avem:*

  ```
  semc x [] = semd x
  ```

[4]Operatorul '`++`' are prioritate mai mică decât operaţia de aplicare a funcţiei. Urmează că ecuaţia ce trebuie demonstrata este:

```
semc x c = (semd x) ++ c
```

10.3 Continuări structurate - tehnica CSC

- Continuările clasice prezintă limitări în modelarea sistemelor concurente [72].
- Tehnica CSC (engl. *continuation semantics for concurrency*) a fost introdusă în [11]. Această tehnică permite în acelaşi timp modelarea compunerii secvenţiale şi modelarea compunerii paralele în sematică de întreţesere furnizând o foarte bună flexibilitate în proiectarea semantică.
- În CSC o continuare este o structură de denotaţii parţial evaluate, sau *calcule* [25, 73, 31, 32].
- Structura continuărilor CSC depinde de aplicaţie. Urmând [73], utilizăm conceptul de *stivă* pentru a modela compunerea secvenţială şi conceptul de *multiset* pentru a modela compunerea paralelă.
- Tehnica CSC permite prototipizarea semantică a limbajelor concurente [73, 79]. Relaţia între CSC şi abordarea clasică directă este investigată în [80].
- Conceptul de *semantică de întreţesere* - utilizât în modelarea sistemelor concurente - va fi introdus într-un capitol ulterior. Deocamdată, tehnica CSC este aplicată în proiectarea unei semantici denotaţionale pentru limbajul secvenţial 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 denotaţională 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 funcţia ce execută continuarea. În cazul limbajului L_0, `cc` activează calculul (denotaţia) din vârful stivei transmiţându-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 secţiunea 10.1. Se pot efectua următoarele teste:

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

Exerciţii

- **E 10.3.1** *Să se arate că operaţia de compunere secvenţială este asociativă şi 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* $\texttt{sem}_{csc}$ *este funcţia semantică definită în această secţiune (10.3), iar* $\texttt{sem}_c$ *este funcţia semantică definită în secţiunea 10.2, atunci dacă* $\texttt{c}_{csc}$ *este o continuare CSC iar* $\texttt{c}_c$ *este o continuare clasică astfel încât:*

  $\texttt{c}_c$ `=` `cc` $\texttt{c}_{csc}$

  atunci pentru orice `x :: X`*:*

  $\texttt{sem}_c$ `x` $\texttt{c}_c$ `=` $\texttt{sem}_{csc}$ `x` $\texttt{c}_{csc}$

  În particular asta înseamnă că relaţia are loc şi pentru continuările iniţiale:

  $\texttt{sem}_c$ `x []` `=` $\texttt{sem}_{csc}$ `x []`

  deoarece

  `[]` `=` `cc []`

10.3.1 Domenii semantice pentru CSC

- Conceptul de *domeniu* (sau *domeniu semantic*) oferă o fundamentare matematică pentru programele recursive.
- Pentru aplicaţii mai avansate domeniile se definesc prin *ecuaţii recursive de domeniu*. Este şi cazul tehnicii CSC, pentru care domeniul continuărilor este soluţia unei ecuaţii recursive în care variabila de domeniu apare în partea stângă a unei construcţii de tip spaţiu de funcţii [11].
  - Soluţia acestui tip de ecuaţii necesită construcţii complexe preluate din teoria categoriilor [66, 67]. În acest sens modelul CSC este complex.
- Însă din punctul de vedere al proiectantului de limbaj tehnica CSC este uşor de utilizat, continuările putând fi proiectate prin combinarea unor structuri simple ce furnizează intuiţie operaţională [73].
- Este celebră ecuaţia de domeniu pentru calculul lambda netipizat [68] (rezolvată de Dana Scott în 1970).

  $$\mathbb{D} \cong \mathbb{D} \to \mathbb{D}$$

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

- În aceste note de curs şi seminar vom utiliza în special următoarele clase de domenii (definite prin ecuaţii recursive):
  - liste (finite sau infinite) de elemente dintr-un domeniu $\mathbb{A}$ dat (unde ϵ denotă *terminare*, iar δ denotă *deadlock*):

    $\mathbb{Q} \cong \{\epsilon\} + (\mathbb{A} \times \mathbb{Q})$, 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 colecţie de liste[5]):

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

    * implementarea Haskell este:

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

[5]Domeniile putere au fost introduse de Gordon Plotkin [71] pentru modelarea comportamentului concurent şi nedeterminist. În exemplul nostru, dacă fiecare listă de tipul $\mathbb{Q}$ reprezintă o trasare de execuţie a unui program, atunci un element de tipul $\mathbb{P}$ poate modela colecţia tuturor trasărilor posibile pentru un program concurent (nedeterminist).

– continuări pentru tehnica CSC, care sunt configuraţii (structuri) de denotaţii parţial evaluate (sau *calcule*); în ecuaţia de mai jos, o continuare (de tipul $Cont$) este o listă de denotaţii (adică o listă de funcţii):[6]

$$\mathbb{D} = Cont \to \mathbb{P}$$
$$Cont \cong \{empty\} + (\mathbb{D} \times Cont)$$

∗ implementarea directă ar fi:

```
type D = Cont -> P
data Cont = Empty | Cont D Cont
```

∗ dar în aplicaţii mai complexe este preferabil să se introducă un domeniu `Comp` de *calcule* (engl. *computations*), care poate fi elaborat în funcţie de aplicaţie:

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

· este important faptul că o asemenea listă de denotaţii sau calcule, poate modela un multiset[7] de calcule (procese) executate în paralel.

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

$$Cont \cong Id \to \mathbb{D}$$

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

[7]Un *multiset* este o colecţie 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* [51].
  - Alternativ, o continuare poate fi interpretată drept *context de evaluare* [60, 70].
- 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.

- Funcţiile unui interpretor semantic CSC pot fi grupate în următoarele trei componente:
  1. un evaluator
  2. o funcţia de execuţie a continuării şi o procedura de normalizare
  3. un planificator (de procese / calcule)

- Un interpretor semantic CSC implementează o buclă "evaluate-normalize-schedule" [70, 73].

1. *Evaluatorul* mapează o continuare deschisă la un comportament program.
   - Este alcătuit din definiţia (compoziţională a) funcţiei semantice împreună cu operatori de control specifici limbajului.
   - Toate funcţiile evaluatorului manipulează (doar) continuări deschise (contexte de evaluare).
2. *Funcţia de execuţie 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 secvenţial planificatorul descompune o continuare închisă într-un calcul activ şi o continuare deschisă corespunzătoare.
   - Pentru un limbaj concurent însă sunt posibile mai multe asemenea activări; asta poate da naştere la nedeterminism. În plus, poate fi necesar ca planificatorul să execute un număr de acţiuni de sincronizare înainte de activarea unui calcul.

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

- Exemplificăm mecanismul de evaluare a continuărilor CSC restructurând interpretorul semantic (pentru limbajul L_0) dat în secţiunea 10.3.
- Domeniul denotaţiilor `D` şi domeniul calculelor `Comp` rămân ca la începutul secţiunii 10.3.

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

- În cazul L_0 domeniul continuărilor deschise `Cont` şi 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 iniţială (`c0 = []`) este o continuare deschisă `c0 :: Cont`.

```
type SC   = [Comp]

type Kont = SC
type Cont = Kont
```

- Evaluatorul interpretorului CSC pentru L_0 cuprinde definiţia (compoziţională a) funcţiei semantice împreună cu un operator de control `addc` pentru compunerea secvenţială. În cazul acestui limbaj simplu `addc` adaugă un calcul în varful stivei ce implementează continuarea. Subliniem faptul că toate funcţiile 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
```

- Funcţia de execuţie a continuării `cc` normalizează continuarea şi apelează planificatorul `kc`. În cazul limbajului L_0 procedura de normalizare `re` coincide cu funcţia identitate.

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

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

  - Constanta `empty` modelează terminarea.

```
empty :: P
empty = []
```

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

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

```
kc :: Kont -> P
kc (D d:c) = d c
```

Capitolul 11

Semantica construcţiilor imperative

- Un limbaj se zice *neuniform* dacă semantica instrucţiunilor (elementare) depinde de starea maşinii de calcul (terminologie din [5]).
- *Starea* (care este un concept abstract) memorează valorile variabilelor programului.
- Introducem un limbaj neuniform L_1.
- L_1 este un limbaj imperativ secvenţial în care modificarea stării maşinii de calcul poate fi comandată prin instrucţiuni de atribuire, şi se pot executa secvenţe arbitrare de instrucţiuni.

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

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

unde se presupun date

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

$$n \ ::= \ z \ | \ v \ | \ n + n \ | \ \cdots$$

  - iar $x_1 \ ; \ x_2$ reprezintă compunerea secvenţială a instrucţiunilor x_1 şi 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 funcţie de la variabile la valori:[1]

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

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

- Conceptul abstract de stare este legat de capacitatea de stocare a valorilor în variabile program şi de modificare a valorilor stocate în variabile.

- Fie $f : A \to B$ o funcţie, $a \in A$ şi $b \in B$. Notaţia $(f \mid a \mapsto b)$ denotă o variantă a funcţiei 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" funcţia f în punctul a.

  – Aşa cum se va vedea în continuare, acest operator poate fi utilizat

    * atât în modelarea semanticii instrucţiunilor de atribuire din limbajele imperative,
    * cât şi în modelarea semanticii legărilor de variabile din limbajele aplicative.

[1] În aplicaţii mai complexe noţiunea poate fi elaborată în funcţie de necesităţi.

11.1.1 Implementare ca funcţie

- Domeniul stărilor Σ poate fi implementat în Haskell astfel:

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

- Utilizând această reprezentare, valoarea ataşată 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 funcţie 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 definiţiei matematice dar nu face posibilă vizualizarea stărilor.[2]
- În exemple considerăm că în starea iniţială toate variabilele sunt neiniţializate:

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

[2]În general, o funcţie nu poate fi vizualizată.

11.1.2 Implementare ca listă de asociaţie

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

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

  cu convenţia că toate variabilele care nu sunt prezente în listă sunt implicit neiniţializate.

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

- În această reprezentare starea iniţială 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`, şi toate celelalte variabile sunt neiniţializate.

- Implementarea operatorilor `val` şi `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 maşinii.
- Semantica expresiilor este definită cu uşurinţă printr-o funcţie de forma:

  $$\mathcal{N}[\![\cdot]\!] : N \to \Sigma \to \mathbb{Z}$$
  $$\mathcal{N}[\![z]\!]\sigma = z$$
  $$\mathcal{N}[\![v]\!]\sigma = \sigma(v)$$
  $$\mathcal{N}[\![n_1 + n_2]\!]\sigma = \mathcal{N}[\![n_1]\!]\sigma + \mathcal{N}[\![n_2]\!]\sigma$$
  $$\ldots$$

  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 secţiune modelăm comportamentul unui program ca funcţie de transformare a stării:

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

- O asemenea funcţie semantică mapează starea iniţială 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 definiţii recursive (sau pentru iteraţie 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 secvenţială x_1 ; x_2 este evaluată astfel. Mai întâi se evaluează instrucţiunea x_1 în starea curentă; aceasta produce o nouă stare ce este utilizată ca stare iniţială pentru evaluarea instrucţiunii x_2.
- Operaţia de atribuire este implementată direct pe baza operatorului `upd`.
- Testăm acest program utilizând stări reprezentate ca liste de asociaţie. 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 funcţie, iar continuarea iniţială este funcţia identitate.
- Se poate efectua următorul experiment:

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

Exerciţii

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

11.4 Istorii de calcul

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

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

- Din nou, se 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 definiţii recursive (sau pentru iteraţie 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 leneşă a limbajului Haskell [62]. Deocamdată nu avem nevoie de acest mecanism.
- În această secţiune oferim numai modele proiectate în semantică de continuare.

11.4.1 Semantică de continuare clasică

- Utilizând tehnica continuărilor clasice se poate defini o semantică denotaţională 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 secţiunea 11.3.1), în care am utilizat în continuare reprezentarea ca listă de asociaţie 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 definiţiile date în 10.3.2. Prezentăm numai definiţiile care se modifică.
- Domeniul `P` este cel definit la începutul secţiunii 11.4, iar pentru domeniul stărilor utilizăm reprezentarea ca listă de asociaţii.
- Definiţiile următoarelor domenii rămân ca în 10.3.2: `D` (denotaţii), `Comp` (calcule), `Kont` (continuări închise), `Cont` (continuări active / deschise).
- De asemenea, definiţiile următoarelor constante şi funcţii rămân ca în 10.3.2: `c0 :: Cont` (continuarea iniţială), operatorul de control `addc`, funcţia de execuţie a continuării `cc`, procedura de normalizare `re` şi funcţia planificator `kc`.
- Se modifică numai constanta `empty`, care devine:

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

şi prima ecuaţie din definiţia semanticii denotaţionale (declaraţia 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 Istorii 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 construcţie $\mathsf{write}\,(n)$ ce produce valoarea expresiei n ca element observabil.

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

- Considerăm următoarea implementare Haskell a sintaxei L_2.

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

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

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

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

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

- Prezentăm doar o semantică CSC pentru L_2. Toate definiţiile rămân ca în secţiunea 11.4.2 (dar în raport cu noul domeniu `P`), exceptând funcţia denotaţională.
- Se modifică doar semantica instrucţiunii de atribuire şi se adaugă o clauză pentru producerea unei valori observabile (ecuaţia semantică pentru compunerea secvenţială 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]
```

Exerciţii

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

Capitolul 12

Recursivitate şi semantică de punct fix

- Majoritatea limbajelor de programare admit definiţii recursive.
- Recursivitatea introduce posibilitatea comportamentului infinit (procese de calcul perpetue).
- Justificarea riguroasă a recursivităţii se realizează în cadrul teoriei domeniilor utilizând *medii semantice* şi *construcţii punct fix*.
- Se utilizează:
  - mulţimi parţial ordonate complete şi funcţii continue (în acest caz se operează cu conceptul de *cel mai mic punct fix*) [6, 49, 63, 56];
  - spaţii metrice complete şi funcţii contractive [5] (o contracţie peste un spaţiu metric complet are un *punct fix unic*, cf. teor. Banach [64]).

12.1 Notaţia letrec

- Introducem limbajul neuniform L_1^{rec}, care extinde L_1 cu o instrucţiune vidă, selecţie condiţională, şi recursivitate.
- Clasa $(x \in)X$ a instrucţiunilor $L_1^{rec}(= X)$ este:

$$\begin{array}{rcl} x & ::= & \mathsf{skip} \mid v := n \\ & \mid & x; x \mid \mathsf{if}\ b\ \mathsf{then}\ x\ \mathsf{else}\ x \\ & \mid & y \mid \mathsf{letrec}\ y\ \mathsf{be}\ x\ \mathsf{in}\ x \end{array}$$

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

- **skip** este instrucţiunea inoperantă (nu modifică starea).
- $v := n$ este instrucţiunea de atribuire.
- $x_1; x_2$ este compunerea secvenţială.
- **if** b **then** x_1 **else** x_2 este instrucţiunea condiţională.
- 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.
- Construcţia **letrec** y **be** x_1 **in** x_2 defineşte 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 execuţia unui program să nu se termine.
- Semantica L_1 poate fi definită utilizând exclusiv conceptele tradiţionale de mulţime, funcţie (ordinară peste mulţimi), şi câteva construcţii simple. De asemenea, operatorii semantici pot fi definiţi prin argumente inductive.
- Pentru a defini semantica L_1^{rec} fiecare mulţime trebuie să fie înlocuită cu un domeniu, şi fiecare funcţie (ordinară) cu o funcţie continuă. De asemenea, argumentele inductive trebuie să fie suplimentate cu argumente de punct fix.
- Intuitiv, un *domeniu semantic* este o mulţime echipată cu o anumită structură, care permite reprezentarea conceptului de calcul şi aproximare în paşi discreţi [6, 63, 5, 49, 56].
- Deocamdată, vom utiliza conceptul de *domeniu* într-un mod strict intuitiv, dar implementările Haskell ne vor ajuta să îi înţelegem (parţial) semnificaţia.

12.2 Medii semantice şi construcţii de punct fix

- Dorim să definim o semantică denotaţională pentru L_1^{rec} de forma $[\![\cdot]\!] : X \to Env \to \mathbb{D}$.
- Să zicem că introducem semantica sub forma unei funcţii de transformare de stare (vezi 11.3):

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

- Pentru ilustrarea proiectăm o semantică directă, deci punem:

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

- Presupunem date funcţii de evaluare pentru expresii numerice şi booleene (care depind de stare):

  $$\mathcal{N}[\![\cdot]\!] : N \to \Sigma \to \mathbb{Z}$$
  $$\mathcal{B}[\![\cdot]\!] : B \to \Sigma \to \{true, false\}$$

- 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 ataşată.

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

- Majoritatea clauzelor care definesc funcţia denotaţională $[\![\cdot]\!]$ pentru L_1^{rec} sunt cunoscute, sau uşor de înţeles. Modelul dat mai jos este proiectat în semantică directă (vezi secţiunea 11.3.1). Pentru lizibilitate folosim aici o notaţie matematică, dar modelul este implementat în secţiunea 12.4.

$$
\begin{aligned}
&[\![\ \mathsf{skip} \ ]\!]\,\eta = \ \lambda\sigma.\,\sigma \\
&[\![\ v := n \ ]\!]\,\eta = \ \lambda\sigma.\,(\sigma \mid v \mapsto \mathcal{N}[\![\ n \ ]\!]\,\sigma) \\
&[\![\ \mathsf{if}\ b\ \mathsf{then}\ x_1\ \mathsf{else}\ x_2 \ ]\!]\,\eta \\
&\qquad = \ \lambda\sigma.\begin{cases} [\![\ x_1 \ ]\!]\,\eta\sigma & \text{if} \quad \mathcal{B}[\![\ b \ ]\!]\,\sigma = true \\ [\![\ x_2 \ ]\!]\,\eta\sigma & \text{if} \quad \mathcal{B}[\![\ b \ ]\!]\,\sigma = false \end{cases} \\
&[\![\ x_1; x_2 \ ]\!]\,\eta = \ \lambda\sigma.\ [\![\ x_2 \ ]\!]\,\eta([\![\ x_1 \ ]\!]\,\eta\sigma)
\end{aligned}
$$

- Ultima clauză (care defineşte comportamentul compunerii secvenţiale în semantică directă) este echivalentă cu:

$$[\![\ x_1 \ ; \ x_2 \ ]\!]\,\eta = \ [\![\ x_2 \ ]\!]\,\eta \ \circ \ [\![\ x_1 \ ]\!]\,\eta$$

  unde ’$\circ$’ este operatorul de compunere a funcţiilor.

- De asemenea, dorim să avem:

$$[\![\ y \ ]\!]\,\eta = \eta(y)$$

- Urmează că singura dificultatea este legată de definirea unei semantici compoziţionale pentru construcţia **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 construcţie **letrec** y **be** x_1 **in** $\cdots$ în raport cu un mediu semantic η este soluţia următoarei *ecuaţii*:

$$\phi = [\![x_1]\!] (\eta \mid y \mapsto \phi) \qquad (*)$$

- 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 ecuaţiei $(*)$???
- Răspunsul la această întrebare este afirmativ. În această secţiune introducem doar la nivel intuitiv ideea ce stă la baza soluţiei.

- Desigur, se poate înlocui apariţia lui ϕ din partea dreaptă a ecuaţiei $(*)$ cu expresia care îl defineşte şi se obţine:

$$\phi = [\![x_1]\!] (\eta \mid y \mapsto [\![x_1]\!] (\eta \mid y \mapsto \phi))$$

- Repetând de un număr finit de ori o asemenea 'depliere' nu vom ajunge însă la o soluţie a ecuaţiei. Totuşi, intuitiv, soluţia s-ar putea obţine ca limită a unui şir de asemenea 'deplieri', când 'efectul' (necunoscut al) lui ϕ din partea dreaptă a ecuaţiei 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 obţine o soluţie generală pentru ecuaţia $(*)$.
- În matematică, o ecuaţie de forma

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

se numeşte ecuaţie de *punct fix*.

- Formal, soluţia ecuaţiei $(*)$ se obţine ca punct fix al unei funcţii de ordin superior de forma:

$$f : \mathbb{P} \to \mathbb{P}, \qquad f(\phi) = [\![x_1]\!] (\eta \mid y \mapsto \phi)$$

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

$$fix(f) = fix(\lambda\phi. [\![x_1]\!] (\eta \mid y \mapsto \phi))$$

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

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

- Semantică a construcţiei **letrec** este:

$$[\![\textsf{letrec } y \textsf{ be } x_1 \textsf{ in } x_2]\!] \eta = \\ [\![x_2]\!] (\eta \mid y \mapsto fix(\lambda\phi. [\![x_1]\!] (\eta \mid y \mapsto \phi)))$$

12.3 Notaţia μ

- Construcţia letrec este intuitivă dar mai puţin convenabilă din punct de vedere matematic.
- În studiile matematice este preferată construcţia:

$$\mu\, y.\, x$$

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

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

- Expresia $\mu\, y.\, x$ reprezintă o definiţie şi în acelaşi timp o utilizare a (un apel al) procedurii y.

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

$$\begin{array}{rcl} x & ::= & \mathsf{skip} \mid v := n \\ & \mid & x; x \mid \mathsf{if}\ b\ \mathsf{then}\ x\ \mathsf{else}\ x \\ & \mid & y \mid \mu\, y.\, x \end{array}$$

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

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

$$[\![\ \mu\, y.\, x\ ]\!]\,\eta = fix(\ \lambda\phi.\ [\![\, x\,]\!]\,(\eta \mid y \mapsto \phi)\ )$$

  Celelalte definiţii şi ecuaţii semantice se pot prelua fără modificări din secţiunea 12.1 şi se obţine o semantică directă pentru L_1^{μ}.

- Se pot efectua următoarele calcule:

$$\begin{array}{l} [\![\ \mu\, y.\, x\ ]\!]\,\eta \\ = [\![\ \mathsf{letrec}\ y\ \mathsf{be}\ x\ \mathsf{in}\ y\ ]\!]\,\eta \\ = [\![\, y\,]\!]\,(\eta \mid y \mapsto fix(\ \lambda\phi.\ [\![\, x\,]\!]\,(\eta \mid y \mapsto \phi)\ )) \\ = fix(\ \lambda\phi.\ [\![\, x\,]\!]\,(\eta \mid y \mapsto \phi)\ ) \end{array}$$

Exerciţii

- **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 acelaşi comportament:

  1. *în limbajul* L_1^{rec} *utilizând construcţia* letrec *şi*
  2. *în limbajul* L_1^{μ} *utilizând construcţia* 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 declaraţiile tipurilor `V` (variabile) şi `N` (expresii numerice) sunt cele introduse la începutul capitolului 11, iar tipurile `Y` şi `B` sunt definite astfel:

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

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

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

- Pentru stări utilizăm reprezentarea teoretică ca funcţii de la variabile la valori (reprezentare introdusă în 11.1.1). Semantica expresiilor numerice ($\mathcal{N}[\![\cdot]\!]$) este cea definită în secţiunea 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 şi `type D = Cont -> P` pentru semantică de continuare).

- Pentru "perturbarea" mediilor semantice utilizăm operatorul polimorfic `upd` introdus în secţiunea 11.1.1. Repetăm aici definiţia `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 iniţial toate procedurile sunt nedefinite.

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

- Am ales să reprezentăm stările ca funcţii de la variabile la valori.
  - Această opţiune ne permite să utilizăm acelaşi operator polimorfic `upd` pentru a modifică medii semantice sau stări.
  - Această strategie corespunde mai bine definiţiilor matematice.
  - În absenţa unei instrucţiuni `write` însă, această strategie îngreunează testarea interpretoarelor.

- În baza mecanismului de evaluare leneşă limbajul Haskell permite următoare definiţie succintă a operatorului de punct fix.

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

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

- În semantica directă codomeniul semanticii denotaţionale este chiar tipul `P`.

```
type D = P
```

- Suntem acum pregătiţi pentru prezentarea funcţiei semantice. Prezentăm aici o semantică directă pentru limbajul L_1^{rec} (oferim implementarea funcţiei semantice ce a fost introdusă în secţiunea 12.2, acolo utilizând o notaţie 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 funcţii 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 şi 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 obţine cu uşurinţă un interpretor semantic pentru limbajul L_1^{μ} prin simpla înlocuire a ecuaţiei ce defineşte semantica construcţiei **letrec** (din interpretorul limbajului L_1^{rec}) cu următoarea ecuaţie semantică pentru construcţia μ:

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

- Toate celelalte definiţii (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} şi L_1^{μ}

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

- În proiectarea modelului CSC pentru L_1^{rec} se utilizează toate definiţiile şi convenţiile introduse în secţiunea 12.4 (înainte de subsecţiunea 12.4.1).
- Definiţiile tipurilor `Comp`, `SC`, `Kont` şi `Cont` rămân ca în secţiunea 10.3.2.
- De asemenea, definiţiile următoarelor constante şi funcţii rămân ca în 10.3.2: `c0 :: Cont` (continuarea iniţială), operatorul de control `addc`, funcţia de execuţie a continuării `cc`, procedura de normalizare `re`, şi funcţia planificator `kc`.
- Domeniul `D` (din signatura funcţiei denotaţionale) este tot ca în secţiunea 10.3.2, dar repetăm aici definiţia sa pentru a sublinia faptul că proiectăm o semantică de continuare.

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

- Constanta `empty` (utilizată în definiţia funcţiei `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 secţiunea 12.4.1 astfel:

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

- La fel ca şi în secţiunea 12.4.1, plecând de la interpretorul CSC al L_1^{rec} se poate obţine un interpretor CSC pentru L_1^{μ} prin simpla înlocuire a ecuaţiei ce defineşte semantica construcţiei **letrec** cu ecuaţia semantică pentru construcţia 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 secţiunea 12.4.1 astfel:

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

Exerciţii

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

Capitolul 13

Semantică de continuare pentru concurenţă

- Tehnica CSC (engl. *continuation semantics for concurrency*) [11, 25] permite atât modelarea compunerii secvenţiale cât şi modelarea compunerii paralele în semantică de întreţesere, 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 maşină secvenţială. În fiecare moment există o singură denotaţie *activă*. O denotaţie rămâne activă numai până când execută o acţiune elementară. Ulterior, o altă denotaţie, preluată din continuare este planificată spre a deveni activă. În acest mod se poate modela execuţia întreţesută a proceselor paralele.

- Tehnica CSC permite proiectantului de limbaj să stabilească o relaţie simplă între o noţiune generală de continuare structurată şi conceptele de control ale limbajului (concurent) studiat.
- Am văzut în capitolele precedente că pentru modelarea compunerii secvenţiale continuarea poate fi structurată sub forma unei stive de calcule.
- Pentru modelarea compunerii paralele în *semantică de întreţesere*[1] continuările CSC pot fi structurate utilizând conceptul de *multiset*.[2]
- Împreună, conceptele de stivă şi multiset par să furnizeze un fundament pentru semantica de fluenţă a controlului.
- În aplicaţii mai avansate, se utilizează continuări CSC cu structură mai complexă, de exemplu arbori *ps* [73], care au la bază o combinaţie a conceptelor de stivă şi multiset.
- Atât conceptul de stivă cât şi conceptul de multiset pot fi implementate ca liste Haskell.

[1]De exemplu, dacă a_1 şi a_2 sunt acţiuni elementare (atomice / neîntreruptibile) dintr-un limbaj concurent uniform (noţiunea de limbaj *uniform* a fost introdusă în capitolul 10) atunci semantica compunerii paralele $a_1 \parallel a_2$ este dată de următoarea 'mulţime' de două trasări:

$$[\![a_1 \parallel a_2]\!] = \{a_1a_2, a_2a_1\}.$$

în care acţiunile a_1 şi a_2 sunt 'întreţesute'. Tehnic vorbind, $\{a_1a_2, a_2a_1\}$ este un element al unui domeniu putere cu structură lineară [5].

În cele ce urmează vom defini semantica denotaţională a concurenţei utilizând tehnica continuărilor, deci funcţia denotaţională va mai avea un parametru (continuarea). Semantica compunerii paralele poate fi însă definită şi fără continuări [81, 5].

[2]Un multiset este o colecţie ce admite elemente duplicate.

- Comportamentul structurii de stivă poate fi definit direct utilizând operaţiile de bază (*car*, *cdr*, *cons*) ataşate 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:

```
ms [d1,d2,d3] =
    [(d1,[•,d2,d3]),
     (d2,[•,d3,d1]),
     (d3,[•,d2,d1])]
```

* 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.
* Cerculeţul plin '•' (care sugerează lipsa unui element) indică poziţia conceptuală a elementului activabil (care a fost selectat).
* Un multiset este o structură neordonată; poziţia 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 secţiune studiem limbajul L_{par}, ce furnizează o construcţie $x_1 \parallel x_2$ pentru compunerea paralelă a proceselor. Clasa $(x \in)X$ a instrucţiunilor $L_{par}(= X)$ este introdusă prin următoarea gramatică:

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

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

- Reutilizând declaraţiile tipurilor `V`, `N`, `B` şi `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 denotaţională pentru L_{par} proiectată cu tehnica CSC.

- În această construcţie considerăm reprezentarea stărilor (tipul `S`) ca funcţii de la variabile la valori (vezi

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

- Optăm aici pentru o funcţie semantică ce depinde de starea curentă şi produce istorii de observabile
  1. fie asamblate în colecţia tuturor trasărilor posibile,
  2. fie alese în mod (pseudo-)aleator.
- În prima variantă interpretorul semantic implementează o semantică denotaţională clasică ce utilizează un *domeniu putere* [71] 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 netriviale.

13.1.1 Specificare

- Începem prin a prezenta modelul teoretic ce produce toate trasările posibile pentru un program nedeterminist. În acest model semantica denotaţională depinde de starea curentă şi produce colecţii 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 secvenţe de observabile. În principiu ar fi putut fi implementat printr-o declaraţie de forma `type Q = [Obs]`, dar declaraţia pentru care am optat aici va permite o tranziţie mai uşoară spre un model semantic ce incorporează noţiunea de *blocare* (engl. *deadlock*) într-o secţiune ulterioară. În anexa B sunt prezentate o instanţă `Eq` şi o instanţă `Show` pentru această declaraţie a tipului `Q`.

- Pentru tipul `P` implementăm:
  - o operaţie **`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 operaţii pentru modelarea alegerii nedeterministe
    - Operaţia **`ned`** modelează o alegere nedeterministă între două alternative; operaţia **`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
```

- Deşi multe definiţii se repetă din capitolele anterioare, pentru a evita orice ambiguitate oferim o prezentare completă a interpretorului semantic pentru L_{par}.
- Domeniile `D` (de denotaţii) şi `Env` (de medii semantice) sunt cele uzuale pentru semantica de continuare.

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

- Definiţia mediului semantic iniţial `e0` precum şi definiţia operatorului polimorfic `upd` rămân ca în secţiunea 12.4.
- Deocamdată şi definiţia domeniului de calcule `Comp` rămâne ca în 10.3.2, dar această definiţie va trebui să fie modificată în secţiunea 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 iniţială (`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 denotaţia activă.
  - Continuările deschise sunt manipulate de evaluator, iar continuările închise sunt manipulate de planificator.

- Evaluatorului CSC pentru L_{par} cuprinde definiţia funcţiilor semantice `semA` şi `sem` împreună cu operatorii de control `addc` şi `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)
```

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

addp :: Comp -> Cont -> Cont
addp p pc = p : pc
```

Compunerea paralelă a două instrucţiuni `x1` şi `x2` este modelată ca o alegere nedeterministă între două alternative: una începând cu instrucţiunea `x1`, cealaltă începând cu `x2`.

În cazul simplu al limbajului L_{par} operatorii de control `addc` şi `addp` adaugă un calcul la multiset. În alte aplicaţii definiţiile operatorilor de control ai evaluatorului pot fi ceva mai complicate [73].

- Funcţia de execuţie a continuării şi procedura de normalizare rămân ca în capitolele precedente; doar constanta `empty` se modifică (repetăm totuşi definiţiile funcţiilor `cc` şi `re`).

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

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

  Procedura de normalizare este funcţia identitate. Pentru limbaje ce incorporează construcţii de control mai avansate (de exemplu pentru limbaje modelate prin continuări structurate sub forma arborilor *ps*) definiţia procedurii de normalizare este ceva mai complexă.

  – Reamintim că procedura de normalizare transformă o continuare activă (deschisă) într-o continuare închisă corespunzătoare. În esenţă, 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 secţiunea 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
```

  Funcţia `scheda` utilizează `actc` pentru a calcula toate planificările de activare posibile. În cazul simplu al limbajului L_{par} (unde o continuare este un multiset) funcţia `actc` coincide cu `ms`. Pentru structuri de continuări mai complexe definiţia `actc` este mai elaborată.

- Pentru testarea acestui interpretor semantic considerăm următoarea instrucţiune 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 exponenţial în lungimea trasărilor de execuţie.
- Este însă posibil să se modifice un interpretor CSC astfel încât să producă la fiecare execuţie 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 şi o funcţie 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 [82]).

```
type R = Int
type RNG = (R,R -> R)

rng0 :: RNG
rng0 = (17489,
        \r -> (25173 * r + 13849)
              `mod` 65536)
```

- Se poate obţine un interpretor CSC pentru L_{par} ce produce o unică trasare (arbitrar aleasă) la fiecare execuţie modificând exclusiv domeniul `P` şi operatorii ataşaţi 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 definiţia constantei `empty` (care este tot de tip `P`).

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

- Toate celelalte definiţii rămân aşa cum au fost date în secţiunea 13.1.1.

- Pentru testarea acestei noi variante a interpretorului este convenabil să utilizăm următoarea funcţie, care execută de un număr specificat de ori o instrucţiune iniţializând de fiecare dată generatorul de numere aleatoare cu o altă valoare. În consecinţă, la execuţii succesive se pot obţine trasări diferite pentru acelaşi 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 instrucţiunea `x1` introdusă în secţiunea 13.1.1, împreună cu următoarea instrucţiune 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; execuţia proceselor paralele apare întreţesută 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 execuţie se poate obţine o trasare diferită.

13.2 Comunicare sincronă

- Introducem un limbaj L_{syn}, care extinde L_{par} cu comunicare sincronă de tip CSP [8, 9].
- Clasa $(x \in)X$ a instrucţiunilor $L_{syn}(= X)$ este dată de următoarea gramatică:

$$\begin{array}{rcl} a & ::= & v := n \mid \mathsf{write}\,(n) \\ x & ::= & \mathsf{skip} \mid a \cdot x \mid \mathsf{if}\ b\ \mathsf{then}\ x\ \mathsf{else}\ x \\ & & \mid y \mid \mathsf{letrec}\ y\ \mathsf{be}\ x\ \mathsf{in}\ x \mid x \parallel x \\ & & \mid \mathsf{ned}\,[(\,\vartheta \to i\,)^*] \end{array}$$

- Unica noutate este dată de construcţia $\mathsf{ned}\,[(\,\vartheta \to i\,)^*]$. Toate celelalte construcţii şi clase sintactice rămân ca în limbajul L_{par}.
- Se utilizează o clasă $(c \in)Ch$ de *canale de comunicare*. Gărzile alternativelor nedeterminste sunt:

$$\vartheta \ ::= \ c!n \mid c?v$$

- Oferim un exemplu de program L_{syn}:

$$
\begin{array}{l}
\mathsf{letrec}\quad y_1\quad \mathsf{be}\\
\mathsf{if}\ (0 < v)\ \mathsf{then}\ \ \mathsf{ned}\ [\ c!(v-1) \to y_1\\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ |\ c!(v+1) \to y_1\\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ |\ c!(v-1) \to y_1\ ]\\
\qquad\qquad\ \ \mathsf{else}\ \ \mathsf{skip}\\
\mathsf{in}\ \ \mathsf{letrec}\quad y_2\quad \mathsf{be}\\
\quad \mathsf{if}\ (0 < v)\ \mathsf{then}\\
\qquad\qquad\qquad\ \ \mathsf{ned}\ [\ c?v \to\ \ \mathsf{write}\,(v)\ \ \cdot y_2\ ]\\
\qquad\qquad\quad\ \ \mathsf{else}\ \ \mathsf{skip}\\
\quad \mathsf{in}\ \ v := 3 \cdot\ \ \mathsf{write}\,(v)\ \ \cdot (y_2 \parallel y_1 \parallel y_1 \parallel y_2)
\end{array}
$$

Acest program creează o reţea de 4 procese comunicante, care tind (în mod nedeterminist) să micşoreze valoarea variabilei v. Nu este dificil de văzut că programul se poate termina normal, dar se poate termina şi în *deadlock*.

- Construcţiile $c!n$ şi $c?v$ sunt ca în Occam [83].
- Execuţia sincronă a două acţiuni $c!n$, $c?v$ care apar în procese paralele revine la transmiterea valorii expresiei n prin canalul c de la procesul care execută instrucţiunea de transmitere $c!n$ la procesul care execută instrucţiunea de recepţionare $c?v$. Procesul care recepţionează valoarea expresiei n o atribuie variabilei v. Intuitiv, acest mecanism implementează o *atribuire distribuită*.
- O construcţie $\mathsf{ned}\,[(\,\vartheta \rightarrow i\,)^*]$ conţine un set de alternative nedeterministe 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 nereuşite.
- Un program se *termină* normal dacă toate procesele îşi 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 comunicaţie şi `C` este tipul gărzilor alternativelor nedeterministe.

- Clasa instrucţiunilor este modificată în raport cu definiţia corespunzătoare pentru L_{par} numai prin adăugarea alternativei `Ned [(C,X)]`.

- Tipurile `V`, `N`, `B` şi `Y` sunt la fel ca pentru L_{par}.

- În cele ce urmează prezentăm numai extensiile şi modificarile ce trebuie să fie aduse definiţiilor date în secţiunea 13.1 pentru limbajul L_{par} pentru a se obţine 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
```

- Instanţele `Eq` şi `Show` ataşate tipului `Q` sunt date în anexa C (`Obs` este un sinonim pentru tipul `Int`, `type Obs = Int`, la fel ca în secţiunea 11.4.3).

- Pentru a modela sincronizarea proceselor concurente extindem definiţia domeniului de calcule `Comp` astfel:

```
data SemC = SemSnd Ch (S -> Int)
          | SemRcv Ch V
data Comp = Den D | Sync [(SemC, D)]
```

- Definiţiile domeniilor pentru denotaţii (`D`), medii semantice (`Env`) şi continuări (tipurile `Cont`, `Kont` şi `PC`) rămân ca în secţiunea 13.1, dar domeniul continuărilor utilizează noul domeniu de calcule `Comp`. De asemenea mediul semantic iniţial (`e0`) şi continuarea iniţială (`c0`) rămân ca în 13.1.

- Evaluatorul CSC pentru limbajului L_{syn} se poate obţine din evaluatorul CSC al limbajului L_{par} prin adăugarea următoarei ecuaţii ce defineşte semantica construcţiei `(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 funcţiei semantice `sem`, precum şi funcţia semantică `semA` şi operatorii de control `addc` şi `addp` rămân ca în secţiunea 13.1.

- De asemenea, funcţia de execuţie a continuării `cc` (inclusiv constanta `empty`) şi procedura de normalizare `re` rămân ca în secţiunea 13.1.

- Planificatorul CSC pentru limbajul L_{syn} reutilizează numai definiţiile funcţiilor `scheda` şi `actc` date în secţiunea 13.1 pentru L_{par}. Domeniul planificărilor `Sched` şi funcţia 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 =
  [ Scheds 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 funcţia `scheda` (dată în 13.1) calculează lista planificărilor de activare de forma `(Scheda d c)`. Planificatorul `kc` mai utilizează acum o funcţie `scheds` ce calculează lista planificărilor de sincronizare de forma `(Scheds v pe k)`.

- Implementarea constantelor `empty` şi `deadlock` depinde de tipul de semantică considerat: toate trasările posibile sau o singură trasare.

13.2.1 Specificare

- Pentru a obţine un model semantic ce calculează toate trasările posibile pentru orice program L_{syn} se poate reutiliza tipul `P` aşa cum a fost definit împreună cu operatorii `put`, `ned`, `bigned` şi constanta `empty` în secţiunea 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 obţine un model semantic ce calculează o singură trasare aleasă în mod (pseudo-)aleator pentru orice program L_{syn} se poate reutiliza tipul `P` aşa cum a fost definit împreună cu operatorii `put`, `ned`, `bigned` şi constanta `empty` în secţiunea 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 secţiunii 13.2.

- Utilizând funcţia `test` dată în secţiunea 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 şi ultima trasare de execuţie se termină în *deadlock*.

Partea IV

Semantica construcţiilor aplicative. Verificare tipuri.

Capitolul 14

Un limbaj de tip PCF

- În acest capitol studiem un limbaj aplicativ L_{pcf} de tip PCF[1] [84]. Prezentarea porneşte de la [50, 49, 60, 57, 58].
- În esenţă, PCF extinde calculul λ simplu tipizat cu constante şi operaţii peste tipuri primitive.
- Introducem L_{pcf} printr-o definiţie BNF împreună cu reguli de deducţie ce permit verificarea statică a corectitudinii tipurilor. Regulile de deducţie au la bază pe triplete de forma:

  $$u \vdash x \ : \ t$$

  unde u este o *atribuire de tipuri* (noţiune ce va fi explicată imediat), x este un termen L_{pcf} şi 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.

[1] Programming Language for Computable Functions

14.1 Sintaxa L_{pcf}

- Introducem sintaxa construcţiilor ($x(\in X)$) şi tipurilor (t) L_{pcf} prin următoarea gramatică:

$$
\begin{array}{rcl}
t & ::= & \mathbf{bool} \mid \mathbf{nat} \mid t \to t \\
x & ::= & \mathsf{true} \mid 0 \\
& & \mid \mathsf{not}(x) \mid \mathsf{succ}(x) \mid \mathsf{pred}(x) \\
& & \mid x \ \mathsf{and} \ x \mid x == x \mid \cdots \\
& & \mid i \mid \mathsf{if}\ x\ \mathsf{then}\ x\ \mathsf{else}\ x \mid \mathsf{let}\ i\ \mathsf{be}\ x\ \mathsf{in}\ x \\
& & \mid x\ x \mid \lambda\, i : t\,.\, x \mid \mu\, i : t.\, x
\end{array}
$$

- Explicaţii
  - Constantele şi operaţiile primitive au semantica intuitivă. Se pot introduce şi alte constante şi alţi 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 construcţie $x_1\ x_2$ termenul x_1 denotă o funcţie ce poate fi aplicată asupra valorii termenului x_2.
  - Construcţia $\lambda\, i : t\,.\, x$ reprezintă o abstracţie lambda în care t este tipul identificatorului i.
  - Construcţia $\mu\, i : t.\, x$ este folosită pentru definiţii 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 şi construcţia `(Mul x1 x2)` pentru operaţia de înmulţire $x_1 * x_2$ (care nu apare în sintaxa abstractă). Desigur, se pot introduce şi alte asemenea operaţii primitive.
  - Nu este însă dificil de văzut că operaţii precum adunarea sau înmulţirea se pot implementa prin definiţii recursive pe baza operaţiilor primitive succ şi pred. De aceea, nu este uzual ca operaţii precum înmulţirea să fie predefinite în studiile semantice asupra PCF.
- Anexa D conţine instanţele `Eq` şi `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 $i_1 : t_1, ..., i_n : t_n$ de perechi identificator-tip, în care identificatorii i_i sunt distincţi. Conceptual, o atribuire de tipuri este o *funcţie* ce mapează un domeniu (finit de) identificatori la tipuri. În Haskell punem:

```
type U = [(I,T)]
```

- Pentru atribuirile de tipuri definim o operaţie de "perturbare" $(u \mid i \mapsto t)$, implementată aici de funcţia `updu`, şi o operaţie de evaluare $u(i)$, implementată de funcţia `iu`.[2]

[2] `updu` este o simplă redenumire a funcţiei `upd` dată în anexa A; `iu` este o variantă a funcţiei `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 [49, 50, 56, 57]:

$$\frac{u_1 \vdash x_1 \ : \ t_1 \qquad \cdots \qquad u_n \vdash x_n \ : \ t_n}{u \vdash \cdots x_1 \cdots x_n \cdots \ : \ t}$$

- De exemplu, regula pentru conjuncţia logică este (aici se utilizează aceeaşi atribuire de tipuri în premise şi în concluzie, dar nu toate regulile respectă o asemenea disciplină):

$$\frac{u \vdash x_1 \ : \ \mathbf{bool} \qquad u \vdash x_2 \ : \ \mathbf{bool}}{u \vdash \ x_1 \ \mathsf{and} \ x_2 \ : \ \mathbf{bool}}$$

- Regulile sunt proiectate astfel încât să ataşeze un tip unic oricărei expresii L_{pcf}. În Haskell, specificăm aceste reguli de deducţie printr-o funcţie:

```
t :: X -> U -> T
```

De exemplu, specificăm regula pentru conjuncţie astfel:

```
t (And x1 x2) u        =
  case (t x1 u,t x2 u) of
       (BOOL,BOOL) -> BOOL
       _           -> err "And"
```

- Funcţia auxiliară `err` este:

```
err :: String -> a
err s = error ("Eroare de tip: " ++ s)
```

- Oferim acum definiţia completă a funcţiei `t` ce ataşează 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"
```

Exerciţii

- **E 14.2.1** *Să se descrie regulile implementate de funcţia* `t` *utilizând notaţia abstractă:*

$$\frac{u_1 \vdash x_1 \ : \ t_1 \qquad \cdots \qquad u_n \vdash x_n \ : \ t_n}{u \vdash \cdots x_1 \cdots x_n \cdots \ : \ t}$$

14.3 Semantică denotaţională

- Datorită recursivităţii, semantica L_{pcf} poate fi definită în mod riguros doar în cadrul matematic al teoriei domeniilor (vezi, de exemplu, [49, 50, 56]).
- Un *domeniu* este o structură matematică care descrie noţiunea de calcul şi aproximare [63].
  - Dana Scott a introdus un element, $\perp$, pentru reprezentarea noţiunii 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 funcţii continue).
  - În această abordare se utilizează adesea noţiunea de mulţime parţial ordonată completă (engl. *complete partial order* (cpo)[3]) ca sinonim pentru conceptul de *domeniu semantic*.

[3]Ideile de bază sunt următoarele (pentru mai multe informaţii cititorul poate consulta [49, 50, 56]):

1. O relaţie de ordine parţială '$\sqsubseteq$' peste o mulţime $(x, y, z) \in X$ este o relaţie:
   reflexivă: $x \sqsubseteq x$
   tranzitivă: $x \sqsubseteq y$ şi $y \sqsubseteq z$ implică $x \sqsubseteq z$, şi
   antisimetrică: $x \sqsubseteq y$ şi $y \sqsubseteq x$ implică $x = y$.
2. Foarte pe scurt, o mulţime parţial ordonată $(x \in)X$ este (ω-)completă dacă odată cu toate elementele unui (ω-)şir crescător $x_1 \sqsubseteq x_2 \cdots \sqsubseteq x_n \sqsubseteq \cdots$ conţine şi elementul *supremum* (engl. *least upper bound*) al şirului.

- De exemplu, dacă $\mathsf{op_{syn}}$ este un operator sintactic unar (cum este **succ**, **pred** sau **not**) atunci semantica expresiei $\mathsf{op_{syn}}(x)$ este:

$$[\![\, \mathsf{op_{syn}}(x) \,]\!]\,\eta = \begin{cases} \mathsf{op_{sem}}([\![\, x \,]\!]\,\eta) & \text{dacă } [\![\, x \,]\!]\,\eta \neq \bot \\ \bot & \text{dacă } [\![\, x \,]\!]\,\eta = \bot \end{cases}$$

  - Se evaluează expresia x (în mediul semantic η) şi dacă rezultatul este definit atunci asupra rezultatului se aplica operatorul semantic $\mathsf{op_{sem}}$.
  - Dacă rezultatul evaluării expresiei x este nedefinit ($\bot$) atunci şi rezultatul evaluării expresiei $\mathsf{op_{syn}}(x)$ este nedefinit.

- În Haskell nu vom implementa explicit constanta $\bot$. Aceasta este o abstracţie ce exprimă un calcul care nu se termină (un calcul divergent).

- De exemplu, vom implementa semantica unei operaţii $\mathsf{op_{syn}}(x)$ (în Haskell `(OpSyn x)`) în forma:

```
sem (OpSyn x) e = opSem (sem x e)
```

- Calculele care nu se termină pot fi modelate în Haskell pe baza mecanismului de evaluare leneşă. Dacă însă evaluarea semanticii `(sem x e)` nu se termină atunci nici pentru expresia `(OpSyn x)` nu se va putea calcula semantica.

- În absenţa recursivităţii, semantica tipurilor L_{pcf} ar putea fi definită (inductiv) astfel:

$$\begin{aligned}
&[\![\ \mathbf{bool}\ ]\!] = \{true, false\}\\
&[\![\ \mathbf{nat}\ ]\!] = \mathbb{N}\\
&[\![\, t_1 \to t_2 \,]\!] = [\![\, t_1 \,]\!] \to [\![\, t_2 \,]\!]
\end{aligned}$$

  unde aici $[\![\, \cdot \,]\!]$ este mulţimea valorilor (interpretărilor) semantice de tipul '$\cdot$'.

  – 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** $\to$ **nat** s-ar evalua la o funcţie $\in \mathbb{N} \to \mathbb{N}$, etc.

- În prezenţa recursivităţi, este necesar să se utilizeze *domenii* (iar nu mulţimi ordinare) pentru interpretarea semantică. Schiţăm câteva idei:
  - Orice mulţime $(m \in)M$ devine un cpo $(M, \sqsubseteq)$ dacă este echipată cu relaţia de ordine discretă:
    $$m_1 \sqsubseteq m_2 \quad \Leftrightarrow \quad m_1 = m_2.$$
  - Pentru orice mulţime M se poate construi un cpo $M_\perp = (M \cup \{\perp\}, \sqsubseteq)$ în care
    $$m_1 \sqsubseteq m_2 \quad \Leftrightarrow \quad (m_1 = m_2 \quad \text{sau} \quad m_1 = \perp)$$
  - Este posibil să se definească în mod riguros operaţii ce construiesc spaţii de funcţii $(\to)$, produse carteziene $(\times)$ şi sume $(+)$ de domenii, rezultatele păstrând structura de domeniu (cpo).
- Domeniul pentru semantica denotaţională a limbajului L_{pcf} poate fi definit (prin inducţie după structura tipurilor t) astfel:

$$[\![\mathbf{bool}]\!] = \{true, false\}_\perp$$
$$[\![\mathbf{nat}]\!] = \mathbb{N}_\perp$$
$$[\![t_1 \to t_2]\!] = [\![t_1]\!] \to [\![t_2]\!]$$

- După cum am explicat înainte, în implementarea Haskell nu există o reprezentare explicită pentru elementul cel mai mic al unui domeniu ($\perp$), divergenţa fiind modelată pe baza mecanismului de evaluare leneşă. Domeniul semanticii denotaţionale a L_{pcf} poate fi implementat prin următoarea declaraţie 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 conţine. În general, semantica unui termen x se defineşte în raport cu o atribuire de tipuri u faţă de care x are tipul t: $u \vdash x \ : \ t$.
- Există o legătură naturală între mediile semantice (care mapează identificatori la valori) şi atribuirile de tipuri (care mapează identificatori la tipuri).

- Spunem despre un mediu semantic η că este *compatibil* [49] cu o atribuire de tipuri u dacă pentru orice identificator i din domeniul lui u $(u = ..., i : t, ...)$, $\eta(i) \in [\![u(i)]\!] (= [\![t]\!])$.
  - Notăm cu $(\eta \in) Env_u$ domeniul mediilor semantice compatibile cu atribuirea de tipuri u.
- Semantica denotaţională poate fi definită pentru fiecare atribuire de tipuri u şi tip t prin funcţii $[\![\cdot]\!]_{ut}$ ce mapează termeni x pentru care $u \vdash x : t$ la funcţii de la Env_u la $[\![t]\!]$:

$$[\![\cdot]\!]_{ut} : \{x \mid u \vdash x : t\} \to Env_u \to [\![t]\!]$$

- De exemplu, ecuaţia ce defineşte semantica abstracţiei μ este:

$$[\![\mu\, i : t.\, x]\!]_{ut}\, \eta = fix(\lambda d\,.\, [\![x]\!]_{(u|i \mapsto t)t}(\eta \mid i \mapsto d))$$

- Indicii u şi t sunt însă adesea omişi din ecuaţiile semantice.

Exerciţii

- **E 14.3.1** *Să se descrie în forma* $[\![x]\!]_{ut}\,\eta = \cdots$ *ecuaţiile ce definesc semantica următoarelor construcţii* L_{pcf}*:*

$\mathsf{0}$	$-$*constanta* $\mathsf{0}$
$\mathsf{succ}(x)$	$-$*operaţia succesor*
i	$-$*identificator*
$\lambda\, i : t\,.\, x$	$-$*abstracţie* λ
$x_1\ x_2$	$-$*aplicare*
$\mu\, i : t.\, x$	$-$*abstracţie* μ

- Interpretorul semantic pentru limbajul L_{pcf} este proiectat în baza presupunerii că *verificarea tipurilor este realizată static* (înainte de execuţie) printr-un apel al funcţiei `t` (vezi definiţia funcţiei `exe`).
  - În consecinţă, indicii u şi t nu apar în ecuaţiile ce definesc funcţia semantică `sem`.
- Semantica denotaţională a L_{pcf} utilizează operatorul `fix` definit în secţiunea 12.4.
- Operatorul `upde` (utilizat aici doar pentru "perturbarea" mediilor semantice) coincide cu operatorul `upd` definit în secţiunea 11.1.1, dar este redenumit pentru a nu fi confundat cu operatorul `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 denotaţională a L_{pcf} astfel (pentru definiţiile matematice se poate consulta, de exemplu, [49, 50, 56]):

```
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

applys :: D -> D -> D
applys (F f) d = f d
```

- Pentru testarea interpretorului L_{pcf} mai utilizăm următoarele definiţii:

```
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] A. Church, A formulation of the simple theory of types, *Journal of Symbolic Logic*, 5:56-68, 1940.

[2] H.B. Curry, R.Feys, W. Craig, *Combinatory logic*, vol. I, North Holland, 1958 (second edition, 1968).

[3] G. Plotkin. A structural approach to operational sematics. Report DAIMI FN-19, Aaurhus University, 1981.

[4] 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.

[5] J.W. De Bakker, E.P. De Vink. *Control flow semantics.* MIT Press, 1996.

[6] 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.

[7] C.A. Gunter, D.S. Scott. Semantic domains. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 633-674, 1990.

[8] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21:667-677, 1978.

[9] C.A.R. Hoare. *Communicating sequential processes.* Prentice Hall, 1985.

[10] I. Stark. A fully abstract domain model for the π-calculus. *Proceedings of LICS'96*, pages 36–42, 1996

[11] 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.

[12] G. Ciobanu, E.N. Todoran. Continuation semantics for asynchronous concurrency. *Fundamenta Informaticae*, vol. 131(3-4), pages 373–388, IOS Press, 2014.

[13] 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*, vol. 83, pages 152–205, 1989.

[14] J.J.M.M. Rutten. Semantic correctness for a parallel object-oriented language. *SIAM Journal of Computing*, vol. 19, pages 341–383, 1989.

[15] J.W. de Bakker, E.P. de Vink. Rendez-vous with metric semantics. *New Generation Computing*, vol. 12, pages 53–90, 1993.

[16] A. De Bruin, W. Bohm, The denotational semantics of dynamic networks of processes, *ACM Transactions on Programming Languages and Systems* 7(4), 656–679, 1985.

[17] E.N. Todoran, N. Gherman. Semantic interpreter for modern communication abstractions in concurrent object oriented programming. *Proceedings of 2008 IEEE 10th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2008)*, pages 289–294, IEEE Computer Press, 2008.

[18] E.N. Todoran, D. Simina, M. Balc, D. Zaharia. Distributed objects and join methods: design issues and operational semantics. *Proceedings of 2010 IEEE 6th International Conference on Intelligent Computer Communication and Processing (ICCP 2010)*, pages 321–328, 2010.

[19] E.N. Todoran, C. Adam, M. Balc, R. Pop, R. Radu, D. Simina, E. Varga, D. Zaharia. Mobile objects and modern communication abstractions: design issues and denotational semantics. *Proceedings of 2011 IEEE 10th International Symposium on Parallel and Distributed Computing (ISPDC 2011)*, pages 191–198, IEEE Computer Press, 2011.

[20] J.W. de Bakker, J.N. Kok. Comparative metric semantics for Concurrent Prolog. *Theoretical Computer Science*, vol. 75, pages 15–43, 1990.

[21] J.W. de Bakker. Comparative semantics for flow of control in logic programming without logic. *Information and Computation*, vol. 94, pages 123–179, 1991.

[22] F.S. de Boer, J.N. Kok, C. Palamidessi, J.J.M.M. Rutten. From failure to succes: comparing a denotational and a declarative semantics for Horn clause logic. *Theoretical Computer Science*, vol. 101, pages 239–263, 1992.

[23] F.S. de Boer, J.N. Kok, C. Palamidessi, J.J.M.M. Rutten. A paradigm for asynchronous communication and its application to concurrent constraint programming. K.R. Apt, J.W. de Bakker, J.J.M.M. Rutten, editors, *Logic programming languages - constraints, functions and objects*, pages 82–114, MIT Press, 1993.

[24] E.N. Todoran, J. den Hartog, E.P. de Vink. Comparative metric semantics for Commit in Or-parallel logic programming. *Proceedings of International Logic Programming Symposium 97 (ILPS 1997)*, pages 101–115, MIT Press, 1997.

[25] 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.*

[26] E.N. Todoran, P. Mitrea, N. Papaspyrou. Comparative semantics for the basic Andorra model. *Automation, Computers, Applied Mathematics*, 14(1):27–41, 2005.

[27] G. Paun. *Membrane computing. An introduction.* Springer, 2002.

[28] O. Andrei, G. Ciobanu, D. Lucanu. A rewriting logic framework for operational semantics of membrane systems. *Theoretical Computer Science*, vol. 373 (3), pages 163–181, 2007.

[29] G. Ciobanu. Semantics of P systems. *Handbook of membrane computing*, pages 413-436, 2010.

[30] L. Cardelli. Strand algebras for DNA computing. *Natural Computing* , vol. 10(1), pages 407–428, 2011.

[31] G. Ciobanu, E.N. Todoran. Metric denotational semantics for parallel rewriting of multisets. *Proceedings of 2011 IEEE 13th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2011)*, pages 276–283, IEEE Computer Press, 2011.

[32] G. Ciobanu, E.N. Todoran. Relating two metric semantics for parallel rewriting of multisets. *Proceedings of 2012 IEEE 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2012)*, pages 273–280, IEEE Computer Press, 2012.

[33] G. Ciobanu, E.N. Todoran. Continuation semantics for dynamic hierarchical systems. *Proceedings of 2015 IEEE 17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2015)*, pages 281-288, IEEE Computer Press, 2015.

[34] E.N. Todoran, N. Papaspyrou. Experiments with continuation semantics for DNA computing. *Proceedings of 2013 IEEE 9th International Conference on Intelligent Computer Communication and Processing (ICCP 2013)*, pages 251–258, 2013.

[35] G. Ciobanu, E.N. Todoran. Correct metric semantics for a language inspired by DNA computing. *Concurrency and Computation: Practice and Experience*, in press, Wiley, 2015.

[36] P.Y. Wong, J. Gibbons. A process semantics for BPMN. *Lecture Notes In Computer Science*, vol. 5256, pages 355–374, 2008.

[37] P.Y. Wong, J. Gibbons. A relative timed semantics for BPMN. *Electr. Notes in Theoretical Computer Science*, vol. 229(2), pages 59–75, 2009.

[38] P.Y. Wong, J. Gibbons. Formalisations and applications of BPMN. *Science of Computer Programming*, vol. 76(8), pages 633650, 2011.

[39] E.N. Todoran, P. Mitrea. Semantic investigation of a control-flow subset of BPMN 2.0. *Proceedings of 2015 IEEE 11th International Conference on Intelligent Computer Communication and Processing (ICCP 2015)*, pages 483–490, 2015.

[40] 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.

[41] C. Strachey. Fundamental concepts of programming languages. Lecture Notes for a NATO Summer School, Copenhagen, 1967.

[42] 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.

[43] D.S. Scott and C. Stratchey. 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.

[44] D.S. Scott. Data types as lattices. *SIAM Journal on Computing*, 5:522–587, 1976.

[45] M. Nivat. Infinite words, infinite trees, infinite computations. J.W. de Bakker and J van Leeuwen, editors, *Foundations of Computer Science III, part 2: Languages, Logic, Semantics*, pages 3–52. Mathematical Centre Tracts 109, Mathematical Centre, Amsterdam, 1979

[46] R. Milner. Processes: a mathematical model of computing agents. In *Proc. of Logic Colloquium*, pages 157-173, North Holland, Amsterdam, 1975.

[47] R. Milner. Fully abstract models of typed lambda-calculi, *Theoretical Computer Science*, 4:1–22, 1977.

[48] J.W. de Bakker, J.I. Zucker. Processes and the denotational semantics of concurrency. *Information and Control*, vol. 54, pages 70–120, 1982.

[49] R.D. Tennent. *Semantics of programming languages.* Prentice Hall, 1991.

[50] C. Gunter. *Semantics of programming languages: structures and techniques.* MIT Press, 1992.

[51] C. Strachey and C. Wadsworth. Continuations: a mathematical semantics for handling full jumps. *Higher-Order and Symbolic Computation*, vol. 13(1/2), pages 135–152, 2000.

[52] G. Ciobanu, E. N. Todoran. Continuation semantics for concurrency with multiple channels communication. *Proceedings of 17th International Conference on Formal Engineering Methods 2015 (ICFEM 2015)*, *Lecture Notes in Computer Science*, vol. 9407, pages 400–416, Springer, 2015.

[53] G. Ciobanu, E.N. Todoran. Continuation semantics for concurrency. Technical Report FML-09-02, Romanian Academy, 2009. `http://iit.iit.tuiasi.ro/TR/reports/fml0902.pdf`

[54] G. Ciobanu, E.N. Todoran. Abstract continuation semantics for asynchronous concurrency. Technical Report FML-12-02, Romanian Academy, 2012. `http://iit.iit.tuiasi.ro/TR/reports/fml1202.pdf`

[55] D.G. Bobrow, B. Wegbreit, A Model and Stack Implementation of Multiple Environments, *Comm. ACM* 16(10), 591–603, 1973.

[56] J.C. Mitchell. *Foundations for programming languages*. MIT Press, 1996.

[57] B. Pierce. *Types and programming languages*. MIT Press, 2002.

[58] B. Pierce, editor, *Advanced Topics in Types and Programming Languages,* MIT Press, 2005.

[59] B. Pierce, et al, *Software Foundations*, Electronic textbook, 2013. (available at `http://www.cis.upenn.edu/ bcpierce/sf`)

[60] M. Felleisen and M. Flat. *Programming languages and Lambda calculi*. Available from `http://www.cs.utah.edu/plt/publications/pllc.pdf`, 2006.

[61] F. Turbak, D. Gifford. *Design concepts in programming languages*. MIT Press, 2008.

[62] 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.

[63] V. Stoltenberg-Hansen, I. Lindstrom and E.R. Griffor. *Mathematical theory of domains*. Cambridge University Press, 1994.

[64] S. Banach, Sur les oprations dans les ensembles abstraits et leur application aux equations integrales, *Fund. Math.* 3:133–181, 1922.

[65] S. Mac Lane. *Categories for the working mathematician*. Springer, 1971.

[66] M.B. Smyth and G.D. Plotkin. The category-theoretic solution of recursive domain ecuations. *SIAM Journal on Computing*, 11:761–783, 1982.

[67] 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.

[68] H.P. Barendregt, *The Lambda Calculus,* North Holland, 1984.

[69] E.N. Todoran. Continuation Semantics for Maximal Parallelism and Imperative Programming. *Automation, Computers, Applied Mathematics*, 23(1):29–35, 2014.

[70] 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.

[71] G. Plotkin. A powerdomain construction. *SIAM Journal on Computing*, 5:522–587, 1976.

[72] R. Hieb, R.K. Dybvig and C.W. Anderson. Subcontinuations. *Lisp and Symbolic Computation*, 7(1):83–110, 1994.

[73] E.N. Todoran, N. 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. `http://www.softlab.ntua.gr/research/techrep/CSD-SW-TR-1-06.pdf`

[74] G. Ciobanu, E.N. Todoran. Continuation semantics for concurrency applied to parallel rewriting of multisets. *Proceedings of 2010 IEEE 12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2010)*, pages 387–391, IEEE Computer Press, 2010.

[75] 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.

[76] V.R. Pratt. The pomset model of parallel processes. *Seminar on Concurrency 1984*, pages 180–196, LNCS 197, 1985.

[77] R. Milner. *Communication and concurrency.* Prentice Hall, 1989.

[78] R. Milner. *Communicating and mobile systems: the π calculus.* Cambridge Univ. Press, 1999.

[79] G. Ciobanu, E.N. Todoran. A methodology for concurrent languages development based on denotational semantics. *Proceedings of 2009 IEEE 11th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2009)*, pages 290–298, IEEE Computer Press, 2009.

[80] E.N. Todoran. A study on the relationship between direct semantics and continuation semantics for concurrency. *Automation, Computers, Applied Mathematics*, 21(1):3-17, 2012. ISSN 1221-437X

[81] G. Plotkin. *Pisa notes,* 1983.

[82] S. Thompson. *Haskell: the craft of functional programming.* Addison-Wesley, 1999.

[83] INMOS Ltd. *Occam programming manual.* Prentice Hall, 1984.

[84] G. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223-255, 1977.

Anexa A

Definiţii pentru starea ca listă de asociaţie

- Această anexă oferă implementarea operatorilor `val` şi `upd` pentru reprezentarea conceptului de stare ca listă de asociaţie dată în secţiunea 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

Definiţii auxiliare pentru secţiunea 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

Definiţii auxiliare pentru secţiunea 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

Definiţii 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}"
```