Fundamente de informatică

Logică propozițională

Marius Minea
marius@cs.upt.ro

http://www.cs.upt.ro/~marius/curs/fi

21 octombrie 2011

De ce logică

E necesară în programare și dincolo de ea
în raționamente, argumente, etc.

Logica propozițională e unul din cele mai simple *limbaje*
așa cum codificăm numere, etc. în *biți*
putem exprima probleme prin *formule* în logică

Problema de azi: fiind dată o formulă, poate fi adevărată ?
(realizabilitate, engl. satisfiability) $\Rightarrow$ SAT checking

Ce știm despre logică?

Știm deja: operatorii logici ȘI ($\wedge$), SAU ($\vee$), NU ($\neg$)

p	$\neg p$
F	T
T	F

negație $\neg$ NU

	$p \wedge q$	q = F	q = T
p	F	F	F
p	T	F	T

conjuncție $\wedge$ ȘI

	$p \vee q$	q = F	q = T
p	F	F	T
p	T	T	T

disjuncție $\vee$ SAU

Implicația logică $\rightarrow$

$p \rightarrow q$

Semnificație: dacă p e adevărat, atunci q e adevărat (if-then)
dacă p nu e adevărat, nu știm nimic despre q (poate fi oricum)

Deci, $p \rightarrow q$ e fals doar dacă p e adevărat și q e fals
(q ar trebui să fie adevărat)

	$p \rightarrow q$	q = F	q = T
p	F	T	T
	T	F	T

Atenție! *fals* implică orice!
$\Rightarrow$ un raționament cu o verigă falsă poate duce la orice concluzie

Implicația nu înseamnă cauzalitate
un fapt adevărat implică orice fapt adevărat (fără legătură)
fals implică orice

Logica propozițională, mai riguros

Limbajul logicii propoziționale: format din *simboluri* pentru
- *propoziții*: p, q, r, etc.
- *operatori* (conectori logici): $\neg$, $\rightarrow$
- paranteze ( )

Formulele logicii propoziționale:
- orice propoziție atomică este o formulă
- dacă α este o formulă, atunci $(\neg\alpha)$ este o formulă.
- dacă α și β sunt formule, atunci $(\alpha \rightarrow \beta)$ este o formulă.

Operatorii cunoscuți pot fi definiți folosind $\neg$ și $\rightarrow$:
- $\alpha \wedge \beta \stackrel{def}{=} \neg(\alpha \rightarrow \neg\beta)$
- $\alpha \vee \beta \stackrel{def}{=} \neg\alpha \rightarrow \beta$
- $\alpha \leftrightarrow \beta \stackrel{def}{=} (\alpha \rightarrow \beta) \wedge (\beta \rightarrow \alpha)$

(am renunțat la parantezele redundante)

Calculul în logică: funcții de adevăr

O *funcție de adevăr* v: atribuie la orice formulă o *valoare de adevăr* $\{T, F\}$ astfel încât:

$v(p)$ e definită pentru fiecare propoziție atomică p.

$$v(\neg\alpha) = \begin{cases} T & \text{dacă } v(\alpha) = F \\ F & \text{dacă } v(\alpha) = T \end{cases}$$

$$v(\alpha \to \beta) = \begin{cases} F & \text{dacă } v(\alpha) = T \text{ și } v(\beta) = F \\ T & \text{în caz contrar} \end{cases}$$

O *interpretare* a unei formule = o evaluare pentru propozițiile ei

O intrepretare *satisface* o formulă dacă o evaluează la T.
(interpretarea e un *model* pentru formula respectivă).

O formulă poate fi:
validă (*tautologie*): adevărată în toate interpretările
realizabilă (satisfiable): adevărată în cel puțin o interpretare
nerealizabilă (*contradicție*): falsă în orice interpretare

Implicația logică (adevărul logic)

O mulțime de formule $H = \{\varphi_1, \ldots, \varphi_n\}$ *implică* o formulă φ

$$H \models \varphi$$

dacă orice funcție de adevăr care satisface H (formulele din H) satisface φ

Pentru a stabili implicația logică trebuie să *interpretăm* formulele (cu valori/funcții de adevăr)

$\Rightarrow$ lucrăm cu *semantica* (înțelesul) formulelor

Deducții logice

O variantă de a stabili adevărul unei formule în mod *sintactic* (folosind doar structura ei)

bazată pe o *regulă de deducție*

$$\frac{\varphi_1 \qquad \varphi_1 \to \varphi_2}{\varphi_2} \qquad \textit{modus ponens}$$

(din φ_1 și $\varphi_1 \to \varphi_2$ deducem φ_2)

și un set de *axiome* (formule care pot fi folosite ca premise/ipoteze)
A1: $\alpha \to (\beta \to \alpha)$
A2: $(\alpha \to (\beta \to \gamma)) \to ((\alpha \to \beta) \to (\alpha \to \gamma))$
A3: $(\neg\beta \to \neg\alpha) \to (\alpha \to \beta)$

Deducție

Fie H o mulțime de formule. Numim *deducție* (demonstrație) din H un șir de formule $A_1, A_2, \cdots, A_n$, astfel ca:

1. A_i este o axiomă, sau
2. A_i este o formulă din H, sau
3. A_i rezultă prin MP din A_j, A_k anterioare ($j, k < i$)

Spunem că A_n rezultă din H (e deductibil, e consecință): $H \vdash A_n$

Exemplu: demonstrăm că $\varphi \to \varphi$

(1) $\varphi \to ((\varphi \to \varphi) \to \varphi))$	A1
(2) $\varphi \to ((\varphi \to \varphi) \to \varphi)) \to ((\varphi \to (\varphi \to \varphi)) \to (\varphi \to \varphi))$	A2
(3) $(\varphi \to (\varphi \to \varphi)) \to (\varphi \to \varphi)$	MP(1,2)
(4) $\varphi \to (\varphi \to \varphi)$	A1
(5) $\varphi \to \varphi$	MP(3,4)

Verificarea unei demonstrații e un proces simplu, mecanic (cele 3 reguli de mai sus), chiar dacă găsirea demonstrației poate fi dificilă.

Consistență și completitudine

$H \vdash \varphi$: *deducție* (pur sintactică, din axiome și reguli de deducție)
$H \models \varphi$: *implicație* (semantică, bazat pe tabele de adevăr) Care e legătura între ele ?

Consistență: Dacă H e o mulțime de formule, și α este o formulă astfel ca $H \vdash \alpha$, atunci $H \models \alpha$.
(Orice teoremă în logica propozițională este o tautologie).

Completitudine: Dacă H e o mulțime de formule, și α este o formulă astfel ca $H \models \alpha$, atunci $H \vdash \alpha$. (Orice tautologie este o teoremă).

Realizabilitatea unei formule propoziționale (satisfiability)

Se dă o formulă în *logică propozițională*.
Există vreo atribuire de valori de adevăr care o face adevărată ?
= e *realizabilă* (engl. *satisfiable*) formula ?

$$\begin{aligned} & (a \vee \neg b \vee \neg d) \\ \wedge\ & (\neg a \vee \neg b) \\ \wedge\ & (\neg a \vee c \vee \neg d) \\ \wedge\ & (\neg a \vee b \vee c) \end{aligned}$$

Găsiți o atribuire care satisface formula?

Formula e în *formă normală conjunctivă* (conjunctive normal form)
= conjuncție de disjuncții de *literale* (pozitive sau neg)

Fiecare conjunct (linie de mai sus) se numește *clauză*

De ce e importantă?

Practic:

În *probleme de decizie* / constrângere:

Putem găsi o soluție la ... cu proprietatea ... ?

$\Rightarrow$ condițiile se pot exprima ca formule în logică

- în verificarea de circuite (ex. optimizăm funcția f în f_{opt})
  $f(v_1, ..., v_n) = f_{opt}(v_1, ..., v_n)$ e echivalent cu
  $f(v_1, ..., v_n) \oplus f_{opt}(v_1, ..., v_n) = 0$
  $\Rightarrow$ e corect dacă $f \oplus f_{opt}$ NU poate fi adevărată
- în verificarea de software (model checking), testare, depanare
- în biologie (determinări genetice), etc.

De ce e importantă?

Teoretic:

E prima problemă demonstrată a fi *NP-completă*.
(probleme care se crede că nu au soluții polinomiale)

O problemă e *NP-completă* dacă
o soluție poate fi *verificată* în timp polinomial (e în *NP*)
(a *verifica* o soluție e mult mai ușor decât a o găsi!)
dacă se *rezolvă* polinomial, atunci și orice altă problemă din NP.

Cum demonstrăm că o problemă e NP-completă (grea) ?
reducem o problemă cunoscută la problema studiată
$\Rightarrow$ dacă s-ar putea rezolva polinomial problema nouă,
atunci s-ar putea rezolva și problema cunoscută

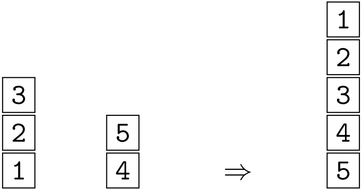
Aplicație: Planificarea

= un termen general pentru probleme de luare de decizie

Exemple:
- deplasările unor roboți inteligenți
- comportamentul sistemelor autonome (sonde spațiale)
- rezolvarea de probleme (de tip puzzle, jocuri, etc.)

În general: într-un sistem descris prin *stări* și *acțiuni* (tranziții), cum găsim o cale de la o *stare inițială* la o *stare țintă* (finală) ?

Exemplu: lumea blocurilor



Acțiuni: mutarea unui bloc liber pe alt bloc.

Ce acțiuni trebuie efectuate ? Care e numărul minim de acțiuni ?

! *Stările* și *tranzițiile* sistemului se pot reprezenta ca *formule logice*

Reprezentarea unei stări

Putem folosi *propoziții* (variabile boolene):

$\boxed{2}$

$\boxed{1}$ $\quad\quad$ $\boxed{3}$ $\quad\quad$ $p_{2on1} \land p_{1on0} \land p_{3on0}$ (2 e pe 1; 1 și 3 pe masa)

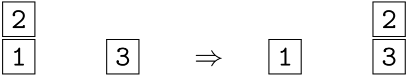
Avem nevoie de: $n \cdot (n-1)$ propoziții pentru perechi de n obiecte, plus n propoziții care exprimă dacă un obiect e pe masă (nr. 0)

scriem și propozițiile neadevărate (din totalul de n^2 propoziții)
$\neg p_{1on2} \land \neg p_{1on3} \land \neg p_{2on0} \land \neg p_{2on3} \land \neg p_{3on1} \land \neg p_{3on2}$

Sau: reprezentăm *pe ce* se află fiecare piesă:
$base_1 = 0 \land base_2 = 1 \land base_3 = 0$
întregi, codificați în binar $\Rightarrow$ total $n \log n$ biți (propoziții)

Codificarea mai compactă nu duce neapărat la rezolvare eficientă.

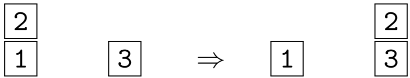
Reprezentarea unei tranziții



Efectul mutării: p'_{2on3} (notăm cu $'$ starea următoare)

Constrângeri de execuție (starea anterioară):

Reprezentarea unei tranziții



Efectul mutării: p'_{2on3} (notăm cu $'$ starea următoare)

Constrângeri de execuție (starea anterioară):

$\neg p_{1on2} \wedge \neg p_{3on2}$ (piesa mutată e liberă)

Reprezentarea unei tranziții

$$\begin{array}{ccccc} \boxed{2} & & & & \boxed{2} \\ \boxed{1} & \boxed{3} & \Rightarrow & \boxed{1} & \boxed{3} \end{array}$$

Efectul mutării: p'_{2on3} (notăm cu $'$ starea următoare)

Constrângeri de execuție (starea anterioară):

$\neg p_{1on2} \land \neg p_{3on2}$ (piesa mutată e liberă)

$\neg p_{1on3} \land \neg p_{2on3}$ (piesa țintă e liberă)

Reprezentarea unei tranziții

$$\begin{array}{ccccc} \boxed{2} & & & & \boxed{2} \\ \boxed{1} & \boxed{3} & \Rightarrow & \boxed{1} & \boxed{3} \end{array}$$

Efectul mutării: p'_{2on3} (notăm cu $'$ starea următoare)

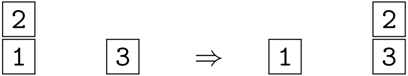
Constrângeri de execuție (starea anterioară):

$\neg p_{1on2} \wedge \neg p_{3on2}$ (piesa mutată e liberă)

$\neg p_{1on3} \wedge \neg p_{2on3}$ (piesa țintă e liberă)

Implicit: $\neg p'_{2on0} \wedge \neg p'_{2on1}$ (2 nu va fi pe altceva)

Reprezentarea unei tranziții



Efectul mutării: p'_{2on3} (notăm cu $'$ starea următoare)

Constrângeri de execuție (starea anterioară):

$\neg p_{1on2} \wedge \neg p_{3on2}$ (piesa mutată e liberă)

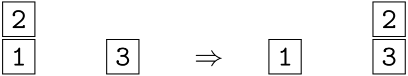
$\neg p_{1on3} \wedge \neg p_{2on3}$ (piesa țintă e liberă)

Implicit: $\neg p'_{2on0} \wedge \neg p'_{2on1}$ (2 nu va fi pe altceva)

$\wedge \neg p'_{1on2} \wedge \neg p'_{3on2}$ (nu va fi altceva pe 2)

$\wedge \neg p'_{1on3}$ (nu va fi altceva pe 3)

Reprezentarea unei tranziții



Efectul mutării: p'_{2on3} (notăm cu $'$ starea următoare)

Constrângeri de execuție (starea anterioară):

$\neg p_{1on2} \wedge \neg p_{3on2}$ (piesa mutată e liberă)

$\neg p_{1on3} \wedge \neg p_{2on3}$ (piesa țintă e liberă)

Implicit: $\neg p'_{2on0} \wedge \neg p'_{2on1}$ (2 nu va fi pe altceva)

$\wedge \neg p'_{1on2} \wedge \neg p'_{3on2}$ (nu va fi altceva pe 2)

$\wedge \neg p'_{1on3}$ (nu va fi altceva pe 3)

Valorile rămân la fel pentru perechile neimplicate:

$p'_{1on0} = p_{1on0} \wedge p'_{3on0} = p_{3on0} \wedge p'_{3on1} = p_{3on1}$

Conjuncția relațiilor descrie mutarea lui 2 pe 3, în toate cazurile

Funcții și relații

O *funcție* $F : A \rightarrow B$ de pe mulțimea A la mulțimea B asociază fiecărui element din A un *unic* element din B.

O *relație* R între mulțimile A si B e o submulțime a produsului cartezian $A \times B$: $R \subseteq A \times B$
adică o mulțime de perechi (a_i, b_j)

Un element $a \in A$ poate fi în relație cu 0, 1, > 1 elemente din B.

O *relație* e mai generală decât o funcție.

Dacă un sistem poate trece dintr-o stare în mai multe stări, folosim o *relație* ca să-i descriem tranzițiile.

Reprezentarea sistemului

Spațiul (mulțimea) *stărilor* S:
dat de propozițiile boolene $p_{i \, \mathrm{on} \, j}$, $1 \le i \le n, 0 \le j \le n$, $i \ne j$

Relația de tranziție:
se poate executa *oricare* (SAU) din tranzițiile posibile într-o stare:
p'_{1on0} (mută 1 pe masă) $\wedge$ *constrângeri mutare 1 pe 0*
$\vee$ p'_{1on2} (mută 1 pe 2) $\wedge$ *constrângeri mutare 1 pe 2*
$\vee$ ... (total 3 x 3 mutări potențiale)
$\vee$ p'_{3on2} (mută 3 pe 2) $\wedge$ *constrângeri mutare 3 pe 2*

Notăm cu $\bar{v} = \langle p_1, p_2, ..., p_N \rangle$ vectorul de stare
Relația de tranziție e o formulă $R(\bar{v}, \bar{v}')$
între starea curentă și starea următoare

! *Stările* și *tranzițiile* sistemului se reprezintă ca *formule logice*

Găsirea unui plan

Fie $S_0(\bar{v})$ și $S_f(\bar{v})$ formulele ce exprimă stările inițiale și finale
Atingerea lui S_f din S_0 în *1 mutare* = e realizabilă formula

$$S_0(\bar{v}_0) \wedge R(\bar{v}_0, \bar{v}_1) \wedge S_f(\bar{v}_1)$$

($\bar{v}_0$ e o stare inițială și $\bar{v}_1$ o stare finală și e o tranziție între ele)

Atingerea lui S_f din S_0 în *k mutări* = e realizabilă formula

$$S_0(\bar{v}_0) \wedge R(\bar{v}_0, \bar{v}_1) \wedge ... \wedge R(\bar{v}_{k-1}, \bar{v}_k) \wedge S_f(\bar{v}_k)$$

$\Rightarrow$ Găsim un plan de lungime minimă căutând succesiv soluții pentru formule tot mai complexe: $2 \cdot N$, ..., $(k+1) \cdot N$ propoziții

Există și alți algoritmi dedicați planificării.
Aici am redus problema la o exprimare *simplă*, fundamentală:
rezolvarea unei formule boolene

Cum stabilim dacă o formulă e realizabilă ?

Observații și reguli simple:

R1) Un literal *singur într-o clauză* are o singură valoare fezabilă:

în	$a \wedge (\neg a \vee b \vee c) \wedge (\neg a \vee \neg b \vee \neg c)$	a trebuie să fie 1
în	$(a \vee b) \wedge \neg b \wedge (\neg a \vee \neg b \vee c)$	b trebuie să fie 0

Cum stabilim dacă o formulă e realizabilă ?

Observații și reguli simple:

R1) Un literal *singur într-o clauză* are o singură valoare fezabilă:

în $a \wedge (\neg a \vee b \vee c) \wedge (\neg a \vee \neg b \vee \neg c)$ $\quad$ a trebuie să fie 1

în $(a \vee b) \wedge \neg b \wedge (\neg a \vee \neg b \vee c)$ $\quad$ b trebuie să fie 0

R2a) Dacă un literal e 1, *pot fi șterse clauzele* în care apare

R2b) Dacă un literal e 0, *el poate fi șters* din clauzele în care apare

Exemplele de mai sus se simplifică:

$a \wedge (\neg a \vee b \vee c) \wedge (\neg a \vee \neg b \vee \neg c) \quad \stackrel{a=1}{\rightarrow} \quad (b \vee c) \wedge (\neg b \vee \neg c)$

$(a \vee b) \wedge \neg b \wedge (\neg a \vee \neg b \vee c) \quad \stackrel{b=0}{\rightarrow} \quad a$

(și de aici $a = 1$, deci formula e realizabilă)

Cum stabilim dacă o formulă e realizabilă ?

R3) Dacă *nu mai sunt clauze*, am terminat (și avem o atribuire)
Dacă se ajunge la o *clauză vidă*, formula *nu e realizabilă*

$a \wedge (\neg a \vee b) \wedge (\neg b \vee c) \wedge (\neg a \vee \neg b \vee \neg c)$

$\stackrel{a=1}{\rightarrow} b \wedge (\neg b \vee c) \wedge (\neg b \vee \neg c)$

$\stackrel{b=1}{\rightarrow} c \wedge \neg c \quad \stackrel{c=1}{\rightarrow} \quad \emptyset$ ($\neg c$ devine clauza vidă $\Rightarrow$ nerealizabilă)

Dacă *nu mai putem face reduceri* după aceste reguli ?

$a \wedge (\neg a \vee b \vee c) \wedge (\neg b \vee \neg c) \quad \stackrel{a=1}{\rightarrow} \quad (b \vee c) \wedge (\neg b \vee \neg c)$??

R4) Alegem o variabilă și încercăm (*despărțim pe cazuri*)

- cu valoarea 1
- cu valoarea 0

O soluție pentru oricare caz e bună (nu căutăm o soluție anume).
Dacă nici un caz nu are soluție, formula nu e realizabilă.

Un algoritm de rezolvare

Problema are ca date:

- lista clauzelor (formula)
- mulțimea variabilelor deja atribuite (inițial vidă)

Regulile 1 și 2 ne *reduc problema la una mai simplă*
(mai puține necunoscute sau clauze mai puține și/sau mai simple)

Regula 3 spune când ne oprim (avem răspunsul).

Regula 4 reduce problema la rezolvarea a *două probleme mai simple*
(cu o necunoscută mai puțin)

Reducerea problemei la *aceeași problemă cu date mai simple*
(una sau mai multe instanțe) înseamnă că problema e *recursivă*.

Obligatoriu: trebuie să avem și o *condiție de oprire*

Algoritmul Davis-Putnam-Logemann-Loveland

```
function solve(env: lit set, org-clauses: lit list list)
clauses = simplify-ones(env, org-clauses)
if clauses is empty list then
   return true;
if clauses has empty clause then
   return false;
if clauses contains single literal a then
   solve (env with a=true, clauses)
else if clauses contains literal with one polarity then
   {optional}
   solve (env with lit=assigned, clauses)
else
   return solve (env with a=false, clauses)
          or solve (env with a=true, clauses);
```

Cu optimizări poate rezolva formule cu $10^4 - 10^5$ variabile

Implementare: lucrul cu liste și mulțimi

Structuri de date:

- *lista* clauzelor (listă de liste de literale)
- *mulțimea* literalelor atribuite cu 1

Prelucrări:

Implementare: lucrul cu liste și mulțimi

Structuri de date:

- *lista* clauzelor (listă de liste de literale)
- *mulțimea* literalelor atribuite cu 1

Prelucrări:

- *căutarea* unui literal în mulțimea celor atribuite

Implementare: lucrul cu liste și mulțimi

Structuri de date:

- *lista* clauzelor (listă de liste de literale)
- *mulțimea* literalelor atribuite cu 1

Prelucrări:

- *căutarea* unui literal în mulțimea celor atribuite
- *adăugarea* unui literal la mulțimea celor atribuite

Implementare: lucrul cu liste și mulțimi

Structuri de date:

- *lista* clauzelor (listă de liste de literale)
- *mulțimea* literalelor atribuite cu 1

Prelucrări:

- *căutarea* unui literal în mulțimea celor atribuite
- *adăugarea* unui literal la mulțimea celor atribuite
- *parcurgerea* literalelor dintr-o listă (clauză)

Implementare: lucrul cu liste și mulțimi

Structuri de date:

- *lista* clauzelor (listă de liste de literale)
- *mulțimea* literalelor atribuite cu 1

Prelucrări:

- *căutarea* unui literal în mulțimea celor atribuite
- *adăugarea* unui literal la mulțimea celor atribuite
- *parcurgerea* literalelor dintr-o listă (clauză)
- *eliminarea* unui literal dintr-o listă (clauză)
- *eliminarea* unei clauze dintr-o listă (formula)

$\Rightarrow$ avem nevoie de tipuri de date de nivel înalt și operații cu ele

Iterarea prin liste

`env` : mulțime de literale adevărate
`lst` : listă de literale (propoziții sau negații)
Dorim: crearea unei noi clauze din care ștergem literalele false
Simplificăm : selectăm elementele `>= 0` dintr-o listă de întregi

```
let rec clrneg = function
  | [] -> []
  | h :: t -> if h >= 0 then h :: clrneg t else clrneg t

val clrneg : int list -> int list = <fun>

# clrneg [1; -4; 5; 6; -7; -4; 2];;
- : int list = [1; 5; 6; 2]
```

Important: s-a creat o *listă nouă*, fără a modifica cea veche
(important în recursivitate: fiecare apel lucrează cu datele proprii)

Iterarea prin liste

Funcția scrisă *filtrează* elemente după un anumit criteriu.
Pentru alt criteriu s-ar schimba doar testul (`h >= 0`)
⇒ putem scrie prelucrarea parametrizând criteriul de filtrare:

```
let rec filter f = function
  | [] -> []
  | h :: t -> if f h then h :: filter f t else filter f t
(* sau, factorizand prelucrarea comuna *)
let rec filter f = function
  | [] -> []
  | h :: t -> let nt = filter f t in
      if f h then h :: nt else nt
```

Funcția `filter` există, predefinită în modulul `List`
⇒ nu e nevoie să rescriem prelucrarea recursivă a listei:
`List.filter (fun x -> x >= 0) [1; -2; 3]`

Alte prelucrări iterative

`iter`: aplică o funcție la toate elementele listei

```
List.iter: ('a -> unit) -> 'a list -> unit = <fun>
  List.iter print_int [1; -2; 3]
```

`map`: creează o listă nouă aplicând o funcție la elemente.

```
List.map: ('a -> 'b) -> 'a list -> 'b list = <fun>
  List.map (fun x -> -x) [1; -2; 3]
```

Iteratorii sunt *polimorfici* (aplicabili la orice tip de liste).

Scriem direct funcțiile aplicate, nu e necesară definirea separată
(funcțiile se folosesc ca și orice alte valori)

! Cu *iteratorii* scriem cod simplu, clar, corect, modular.

E timpul să structurăm

Vrem cod independent de reprezentarea literalelor (șiruri, întregi...)

E esențial: să putem nega un literal, și să putem crea mulțimi.
Defini *semnătura* (interfața) unui tip și un modul de implementare

```
module type LITERAL = sig                    (* interfaţa *)
  type t
  val compare: t -> t -> int  (* necesar pt. mulţimi *)
  val neg: t -> t
end
module StrLit = struct  (* instantiem tipul propriu-zis *)
  type t = Pos of string | Neg of string
  let compare = Pervasives.compare     (* fct. standard *)
  let neg = function
    | Pos s -> Neg s
    | Neg s -> Pos s
end
```

(cod după Conchon et. al, SAT-MICRO)

Un modul parametrizat

Creem un modul care poate lucra cu orice literal care satisface interfața (semnătura) `LITERAL` definită.
⇒ putem schimba oricând reprezentarea, păstrând codul

```
module Sat(L: LITERAL) = struct

  module S = Set.Make(L)  (* tipul multime de literali *)

  exception Sat of S.t    (* transmite literalii = T *)
  exception Unsat

  (* aici definim functiile modulului *)
end
```

Revenim: filtrăm literalele adevărate

Păstrăm în clauza `cl` doar lit. care *nu* apar neg în `env` (R2b)

```
List.filter (fun lit ->
               not (S.mem (L.neg lit) env)) cl
(* S.mem e functia membru pentru tipul multime S *)
```

Funcția transformă fiecare element din `clauses` ⇒ o nouă listă

```
List.map (fun cl -> List.filter
           (fun lit ->
             not (S.mem (L.neg lit) env)) cl) clauses
```

Găsirea unui literal adevărat e un caz special (R2a)
⇒ nu mai continuăm prelucrarea clauzei (*excepție*)
⇒ clauza e ștearsă ⇒ nu putem folosi `map`

O mapare selectivă a listelor

Înlocuim `map` cu o funcție care poate elimina elemente din listă:

```
(* am denumit filter_clause filtrarea definita anterior *)
let rec filtermap = function
  | [] -> []
  | cl :: t -> let newcl = filter_clause cl in
        if newcl = [] then filtermap t
        else newcl :: filtermap t
```

Dacă `newcl` nu e vidă, e adăugată la lista rezultat.
Funcția face prelucrări (`::`) *după revenirea* din apelul recursiv
⇒ folosește stivă proporțională cu lungimea listei

Soluție: *acumularea* rezultatului ca parametru suplimentar
⇒ *recursivitate la dreapta* (tail recursion)
⇒ *transformabilă automat în ciclu*, nu consumă stiva

De la map la o funcție mai generală: fold

```
(* am denumit filter_clause prelucrarea unei clauze/liste *)
let rec filtermap result = function
  | [] -> result    (* rezultatul acumulat pana acum *)
  | cl :: t -> let newcl = filter_clause cl in
       if newcl = [] then filtermap result t
       else filtermap (newcl :: result) t
```

Funcția se apelează recursiv pe coada listei `t` cu un *rezultat parțial* care e o funcție de cel anterior `result` și capul listei `cl`

```
let rec fold_left f res = function
  | [] -> res
  | h :: t -> fold_left f (f res h) t
```

$fold\_left\ f\ \ r\ [x_1; ...; x_n] = f(...f(f(f\ \ r\ x_1)\ x_2)\ x_3...)\ x_n$
Funcția e predefinită: `List.fold_left`

Exemple cu `fold_left`

Suma elementelor unei liste:

```
List.fold_left (+) 0 [1; 4; 6]
```

Produsul elementelor unei liste:

```
List.fold_left (*) 1 [2; 4; 5; 7]
```

Inversarea unei liste:

```
List.fold_left (fun t h -> h :: t) [] [1; 2; 3; 4]
```

Funcțiile `filter` și `map` sunt cazuri particulare:

```
let filter f lst = List.rev (List.fold_left
      (fun r h -> if f h then h :: r else r) [] lst)
let map f lst = List.rev
      (List.fold_left (fun r h -> f h :: r) [] lst)
```

Simplificarea clauzelor

Construim lista de clauze noi aplicând `fold_left` pe `clauses` pornind de la lista vidă:

```
let simplify env =
  List.fold_left (
    fun ncls cl ->      (* ncls = lista clauze noi *)
      match List.filter
        (fun lit -> not (S.mem (L.neg lit) env)) cl with
          | [] -> raise Unsat     (* clauza vidă *)
          | [lit] -> ncls   (* şterge clauza unitate *)
          | newcl -> newcl :: ncls  (* adaugă clauza modif. *)
  ) []
```

Completăm codul:

dacă `filter` găsește lit. în `env`, ștergem clauza (excepție, R2a)

un literal unitate `[lit]` e adăugat la `env` ca 1 (R1+R2a)

Lucrul cu excepții

`raise` *exceptie*

generează excepția numită

`try` *expresie* `with` *exceptie* -> *expresie-rezultat*

captează excepția numită și calculează un alt rezultat

Excepțiile întrerup prelucrări prin oricâte apeluri de funcție

```
try
  List.fold_left (fun v el ->
    if el = 0 then raise Exit else v * el) 1 [1;2;0;3;4;5]
with Exit -> 0
```

Excepții predefinite: `Exit`, `Failure of string`

`raise (Failure "text")` se mai scrie `failwith "text"`

Excepțiile pot returna valori: definim `exception` *Nume-exc* `of` *tip*

Simplificarea clauzelor (cont.)

```
let simplify env =
  List.fold_left (
    fun (nenv, ncls) cl -> try
      let ncl = List.filter
        (fun lit ->
          if S.mem lit nenv then raise Exit;
          not (S.mem (L.neg lit) nenv)) cl
      in match ncl with
          [] -> raise Unsat
        | [lit] -> simplify (S.add lit nenv) ncls
        | _ -> (nenv, necl :: ncls)
    with Exit -> (nenv, ncls)
  ) (env, [])
```

Noua variantă returnează o *pereche* (env, clauses)
când găsește un literal simplu `[lit]` simplifică din nou clauzele
deja parcurse, adaugând literalul la mulțimea celor adevărați

Verificarea propriu-zisă

Implementăm R4 care încearcă ambele valori pentru un literal:

```
let rec sat ones clist =
  let (ones, clist) = simplify ones clist in
  if clist = [] then raise (Sat ones) else
    let lit = List.hd (List.hd clist) and rst = List.tl clist in
    try sat (S.add lit ones) rst
    with Unsat -> sat (S.add (L.neg lit) ones) rst
```

Soluția finală:

```
let solve clist = try sat S.empty clist
                  with Sat ones -> S.elements ones

module SatS = Sat(StrLit)
open StrLit;;
SatS.solve [[Pos "a"; Pos "b"; Neg "c"]; [Neg "a"; Pos "c"];
          [Pos "a"; Neg "b"]];;
- : SatS.S.elt list = [Pos "a"; Pos "c"]
```