Logică și structuri discrete

Logică propozițională

Marius Minea
marius@cs.upt.ro

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

7 noiembrie 2016

Unde folosim formule propoziționale ?
- în *hardware*, construit din *circuite logice*
- în *software*
  - formule mici (simple), în **`if`**, **`while`**, etc.
  - formule mari, în probleme de *căutare* și *planificare*

Verificarea realizabilității și aplicațiile ei

Ce înseamnă o *demonstrație* logică?

Unde aplicăm logica booleană?

Calculatoarele sunt construite din *circuite logice*
⇒ realizează aceleași *funcții* ca în logică (ȘI, SAU, NU)

Numerele sunt reprezentate în calculator *în baza 2*
⇒ valori *boolene* / biți: (0 sau 1, F sau T)

Aritmetica pe numere e implementată prin circuite logice

```
unsigned add(unsigned a, unsigned b) {
  return b ? add(a^b, (a&b) << 1) : a;
}
```

```
let rec add a b =
  if b = 0 then a else add (a lxor b) ((a land b) lsl 1)
```

Mulțimile pot fi reprezentate prin vectori de valori boolene
pentru fiecare element: face sau nu parte din mulțime ?

Orice noțiune din matematică sau realitate e reprezentată pe biți

Aplicație: Planificarea

= găsirea unei secvențe de *acțiuni* care *duc la ținta* dorită

Exemple:

deplasări de roboți inteligenți

comportamentul sistemelor autonome (sonde spațiale)

rezolvarea de probleme (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: ordonarea a 3x3 piese

2		5
1	3	4
8	6	7

Se poate reface ordinea? Din câte mutări?

Numerotăm pozițiile:
```
123
456
789
```

starea jocului e dată de piesele de pe fiecare din cele 9 poziții:
un *vector de stare* $\bar{v} = (p_1, p_2, \ldots, p_9)$, $p_i \in [0..8]$ $(0 = \text{liber})$

fiecare valoare p_i poate fi reprezentată boolean (cu 4 biți)

sau direct cu booleni b_{ij} = piesa j (0..8) e pe poziția i (1..9)
cu constrângeri: un singur b_{ij} adevărat pentru fiecare i
(un singur număr în fiecare loc)

O *stare* e descrisă printr-o *formulă logică*

peste variabilele de stare (elementele vectorului de stare)

Starea inițială din figură: $S_0(\bar{v}) \stackrel{\text{def}}{=} p_1 = 2 \wedge p_2 = 0 \wedge p_3 = 5 \wedge p_4 = 1$
$\wedge\, p_5 = 3 \wedge p_6 = 4 \wedge p_7 = 8 \wedge p_8 = 6 \wedge p_9 = 7$

Reprezentarea unei mutări

Notăm $\bar{v}$ starea curentă, $\bar{v}' = (p'_1, p'_2, \ldots, p'_9)$ starea următoare

Sunt 12 perechi de poziții vecine: $P = \{(1,2), (1,4), \ldots (8,9)\}$ 123
456
789

O *mutare* e posibilă doar între două poziții vecine s și d

$m_{sd}(\bar{v}, \bar{v}') = (p'_s = p_d) \wedge (p'_d = p_s)$ interschimbă valorile

$\wedge (p_s = 0 \vee p_d = 0)$ dacă una din poziții e liberă

$\wedge \bigwedge_{i \neq s,d} (p'_i = p_i)$ celelalte piese rămân la fel

Toate mutările potențiale definesc *relația de tranziție* a sistemului:
o *relație* între starea curentă $\bar{v}$ și posibilele stări următoare $\bar{v}'$

$$R(\bar{v}, \bar{v}') = \bigvee_{(s,d) \in P} m_{sd}(\bar{v}, \bar{v}')$$

avem *disjuncție*: facem schimbul $1 \leftrightarrow 2$ *sau* schimbul $1 \leftrightarrow 4$, *sau* ...

Reprezentarea unui șir de mutări

Un lanț de mutări are o mutare (tranziție) între fiecare stare și cea următoare. Numerotăm vectorii pentru fiecare stare $\bar{v}^0$, $\bar{v}^1$, ... $\bar{v}^k$:

$$R(\bar{v}^0, \bar{v}^1) \wedge R(\bar{v}^1, \bar{v}^2) \wedge \ldots \wedge R(\bar{v}^{k-1}, \bar{v}^k)$$

O *tranziție* și un *șir de tranziții* se pot reprezenta ca formule logice

Prima mutare: $R(\bar{v}^0, \bar{v}^1) =$

$$(p_1^0 = 0 \vee p_2^0 = 0) \wedge p_1^1 = p_2^0 \wedge p_2^1 = p_1^0 \wedge p_3^1 = p_3^0 \wedge \ldots \wedge p_9^1 = p_9^0 \quad 1 \leftrightarrow 2$$

$$\vee (p_1^0 = 0 \vee p_4^0 = 0) \wedge p_1^1 = p_4^0 \wedge p_4^1 = p_1^0 \wedge p_2^1 = p_2^0 \wedge \ldots \wedge p_9^1 = p_9^0 \quad 1 \leftrightarrow 4$$

...

$$\vee (p_8^0 = 0 \vee p_9^0 = 0) \wedge p_9^1 = p_8^0 \wedge p_8^1 = p_9^0 \wedge p_3^1 = p_3^0 \wedge \ldots \wedge p_7^1 = p_7^0 \quad 8 \leftrightarrow 9$$

Starea finală e tot o formulă peste elementele vectorului de stare $\bar{v}$:

$S_f(\bar{v}) = p_1 = 1 \wedge p_2 = 2 \wedge \ldots \wedge p_7 = 7 \wedge p_8 = 8 \wedge p_9 = 0$

Există o *soluție* în k pași dacă și numai dacă

$$S_0(\bar{v}^0) \wedge R(\bar{v}^0, \bar{v}^1) \wedge R(\bar{v}^1, \bar{v}^2) \wedge \ldots \wedge R(\bar{v}^{k-1}, \bar{v}^k) \wedge S^f(\bar{v}^k)$$

Găsirea unui plan

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

$$S_0(\bar{v}^0) \land R(\bar{v}^0, \bar{v}^1) \land S_f(\bar{v}^1)$$

($\bar{v}^0$ e o stare inițială și $\bar{v}^1$ o stare finală și e o tranziție între ele)

A ajunge la S_f din S_0 în *k mutări* $\Leftrightarrow$ e realizabilă formula

$$S_0(\bar{v}^0) \land R(\bar{v}^0, \bar{v}^1) \land ... \land R(\bar{v}^{k-1}, \bar{v}^k) \land S_f(\bar{v}^k)$$

$\Rightarrow$ Găsim un *plan de lungime minimă* căutând succesiv soluții pentru formule tot mai complexe: 1, 2, 3, ... pași

Există și alți algoritmi dedicați planificării.
Aici am redus problema la o exprimare *simplă*, fundamentală:
determinarea realizabilității unei formule boolene (problema SAT)

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 *literali* (pozitive sau negate)

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

Cum stabilim dacă o formulă e realizabilă ?

Reguli de simplificare:

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 T

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

R2a) Dacă un literal e T, *pot fi șterse clauzele* în care apare
(ele sunt adevărate, și nu mai influențează formula)

R2b) Dacă un literal e F, *el poate fi șters* din clauzele în care apare
(nu ajută în a face clauza adevărată)

Exemplele de mai sus se simplifică:

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

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

și de aici $a = T$, 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*

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

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

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

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

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

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

- cu valoarea F
- cu valoarea T

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 (1962)

function solve(truelit: lit set, clauses: clause list)
(truelit, clauses) = simplify(truelit, clauses) (* R1, R2 *)
if clauses = lista vidă **then**
  return truelit; (* R3: realizabila, returneaza atribuirile *)
if clauses conține clauza vidă **then**
  raise Unsat; (* R3: nerealizabila *)
if clauses conține clauză cu unic literal *a* **then**
  solve (truelit $\cup \{a\}$, clauses) (* R1: *a* trebuie să fie T *)
else
  try solve (truelit $\cup \{\neg a\}$, clauses); (* R4: încearcă a=F *)
  with Unsat $\rightarrow$ solve (truelit $\cup \{a\}$, clauses); (* încearcă T *)

Rezolvitoarele (*SAT solvers/checkers*) moderne pot rezolva formule cu milioane de variabile (folosind optimizări)

Implementare: lucrul cu liste și mulțimi

Structuri de date:

- *lista* clauzelor (listă de liste de literali)
- *mulțimea* literalilor cu valoare T

Prelucrări:

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

Cum reprezentăm un literal ?

un șir (numele variabilei) etichetat cu P (pozitiv) / N (negativ)

```
module L = struct
  type t = P of string | N of string (* pozitiv / negat *)
  let compare = compare  (* fct. std. Pervasives.compare *)
  let neg = function     (* negare = schimba eticheta *)
    | P s -> N s
    | N s -> P s
 end
 module S = Set.Make(L) (* pentru multimi de literali *)
```

(cod după Conchon et. al, SAT-MICRO, 2008)

Sau reprezentăm o propoziție p_k prin indicele întreg $k \in \mathbb{N}^*$
și folosi indici negativi pentru negație

```
module L = struct
  type t = int
  let compare = compare
  let neg x = -x
end
```

Simplificarea unei clauze

`tset` = mulțimea literalilor adevărați

Găsirea unui literal adevărat e un caz special (R2a)
⇒ nu mai continuăm prelucrarea clauzei (*excepția* `Exit`)

Altfel, păstrăm un literal dacă nu e sunt sigur fals (R2b)
(nu apare negat în mulțimea celor adevărate, `tset`)

```
let filter_clause tset =
  List.filter (fun lit ->
    if S.mem lit tset then raise Exit (* clauza e adevarata *)
    else not (S.mem (L.neg lit) tset)) (* pastram daca nu e F *)
```

Simplificarea listei de clauze

Acumulăm cu `List.fold_left` o *pereche* de valori:
mulțimea de literali adevărați și lista clauzelor modificate

```
let rec simplify trueset = List.fold_left
  (fun (tset, clst) cl -> (* acumulator + clauza curenta *)
    try (match filter_clause tset cl with
    | [] -> raise Unsat (* clauza vida -> nerealizabila *)
    | [lit] -> simplify (S.add lit tset) clst (* reia cu lit=T *)
    | newcl -> (tset, newcl :: clst))
    with Exit -> (tset, clst) (* nu adauga clauza T *)
  ) (trueset, [])
```

Dacă `filter_clause` dă un unic literal, se adaugă la cele adevărate
și reluăm simplificarea clauzelor deja prelucrate
Dacă returnează lista vidă, toată formula e nerealizabilă
Dacă produce excepția `Exit`, clauza nu are efect (e adevărată)
Altfel, adăugăm clauza simplificată la listă

Verificarea propriu-zisă

Dacă simplificând obținem lista vidă de clauze, returnăm mulțimea literalilor adevărați (restul nu contează)

Altfel, cu primul literal din prima clauză încercăm ambele valori
   dacă prima încercare dă excepția `Unsat`, încercăm și a doua

```
let sat clauses =
  let rec sat1 tset clist =
    match simplify tset clist with
    | (ts1, (lit::cl)::ctail) ->  (* luam primul literal *)
      S.union ts1 (          (* cei deja true + nou aflati *)
        try sat1 (S.singleton (L.neg lit)) (cl::ctail) (* lit=F *
        with Unsat -> sat1 (S.singleton lit) ctail (* lit=T *)
      )
    | (ts1, _) -> ts1 (* _ va fi []; ts1 = literali adevarati *)
  in S.elements (sat1 S.empty clauses) (* init.fara atribuiri *)

sat [[P "a"; P "b"; N "c"]; [N "a"; P "c"]; [P "a"; N "b"]]
- : S.elt list = [N "a"; N "b"; N "c"]
```

$(a \vee b \vee \neg c) \wedge (\neg a \vee c) \vee (\neg a \vee b)$ e realizabilă cu $a = b = c = \mathsf{F}$

Unde se aplică determinarea realizabilității?

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})
  dacă $f(v_1, ..., v_n) = f_{opt}(v_1, ..., v_n)$ (echivalente)
  atunci $\neg(f(v_1, ..., v_n) = f_{opt}(v_1, ..., v_n))$ e nerealizabilă
  putem verifica dacă transformarea (optimizarea) e corectă
- în verificarea de software (model checking), testare, depanare
  găsirea de teste care duc programul pe o anume cale
  găsirea de vulnerabilități de securitate în software
- în biologie (determinări genetice), etc.

Complexitatea realizabilității

n propoziții: 2^n atribuiri $\Rightarrow$ timp *exponențial* încercând toate
O atribuire dată se verifică în timp *liniar* (în dimensiunea formulei)

P = clasa problemelor care pot fi rezolvate în timp polinomial
(relativ la dimensiunea problemei)

NP = clasa problemelor pentru care o soluție poate fi *verificată* în timp polinomial (a verifica e mai ușor decât a găsi)

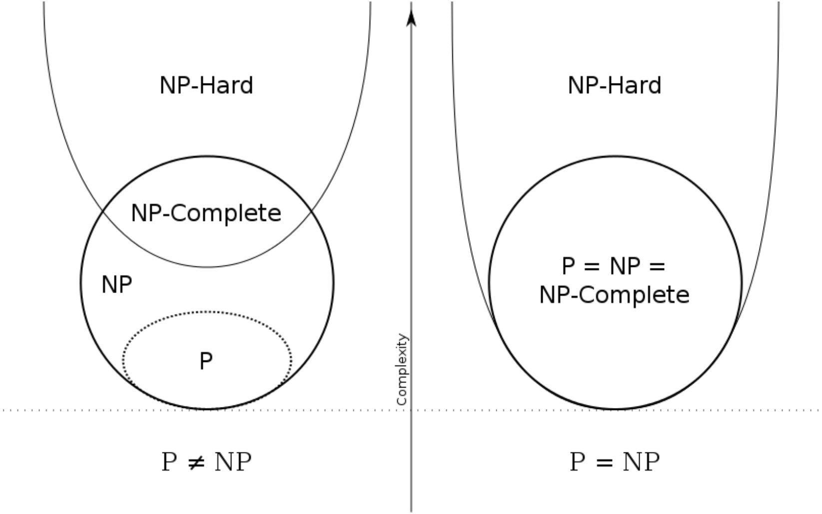
Probleme *NP-complete*: cele mai dificile probleme din clasa *NP*
dacă s-ar rezolva în timp polinomial, orice altă problemă din NP s-ar rezolva în timp polinomial $\Rightarrow$ ar fi P = NP (se crede P $\neq$ NP)

Realizabilitatea (SAT) e prima problemă demonstrată a fi *NP-completă* (Cook, 1971). Sunt multe altele (21 probleme clasice: Karp 1972).

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

P = NP?

Una din cele mai fundamentale probleme în informatică



Se crede că $P \neq NP$, dar nu s-a putut (încă) demonstra

Imagine: `http://en.wikipedia.org/wiki/File:P_np_np-complete_np-hard.svg`

Sintaxă și semantică

Pentru logica propozițională, am discutat:

Sintaxa: o formulă are *forma*:
propoziție sau ($\neg$ *formulă*) sau (*formulă* $\rightarrow$ *formulă*)

Semantica: calculăm *valoarea de adevăr* (înțelesul), pornind de la cea a propozițiilor

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

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

Deducții logice

Deducția ne permite să demonstrăm o formulă în mod *sintactic* (folosind doar structura ei)

E bazată pe o *regulă de inferență* (de deducție)

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

(din φ_1 și $\varphi_1 \rightarrow \varphi_2$ deducem/inferăm φ_2)

și un set de *axiome* (formule care pot fi folosite ca premise/ipoteze)
A1: $\alpha \rightarrow (\beta \rightarrow \alpha)$
A2: $(\alpha \rightarrow (\beta \rightarrow \gamma)) \rightarrow ((\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma))$
A3: $(\neg\beta \rightarrow \neg\alpha) \rightarrow (\alpha \rightarrow \beta)$
în care α, β etc. pot fi înlocuite cu *orice* formule

Deducție (demonstrație)

Fie H o mulțime de formule. O *deducție* (demonstrație) din H e un șir de formule $A_1, A_2, \cdots, A_n$, astfel ca $\forall i \in \overline{1, n}$

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

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

Exemplu: demonstrăm că $\varphi \to \varphi$ pentru orice formulă φ

(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.

Alte reguli de deducție

Modus ponens e suficient pentru a formaliza logica propozițională dar sunt și alte reguli de deducție care simplifică demonstrațiile

$$\frac{p \to q \qquad \neg q}{\neg p}$$ *modus tollens (reducere la absurd)*

$$\frac{p}{p \vee q}$$ *generalizare (introducerea disjuncției)*

$$\frac{p \wedge q}{p}$$ *specializare (simplificare)*

$$\frac{p \vee q \qquad \neg p}{q}$$ *eliminare (silogism disjunctiv)*

$$\frac{p \to q \qquad q \to r}{p \to r}$$ *tranzitivitate (silogism ipotetic)*

Deducția (exemplu)

Fie $H = \{a,\ \neg b \vee d,\ a \to (b \wedge c),\ (c \wedge d) \to (\neg a \vee e)\}$.
Arătați că $H \vdash e$.

(1) a	ipoteză, H_1
(2) $a \to (b \wedge c)$	ipoteză, H_3
(3) $b \wedge c$	modus ponens (1, 2)
(4) b	specializare (3)
(5) d	eliminare (4, H_2)
(6) c	specializare (3)
(7) $c \wedge d$	(5) și (6)
(8) $\neg a \vee e$	modus ponens (7, H_4)
(9) e	eliminare (1, 8)

Consecința logică (semantică)

Reamintim: o *interpretare* e o atribuire de adevăr pentru propozițiile unei formule.
O formulă poate fi adevărată sau falsă într-o interpretare.

O mulțime de formule $H = \{\varphi_1, \ldots, \varphi_n\}$ *implică* o formulă φ
(sau φ e o *consecință logică* / consecință semantică a ipotezelor H)

$$H \models \varphi$$

dacă orice interpretare care satisface (formulele din) H satisface φ

Pentru a stabili consecința semantică trebuie să *interpretăm* formulele (cu valori/funcții de adevăr)
$\Rightarrow$ lucrăm cu *semantica* (înțelesul) formulelor

Exemplu: $\{\alpha \vee \beta, \gamma \vee \neg\beta\} \models \alpha \vee \gamma$ Fie o interpretare v.
Cazul 1: $v(\beta) = \mathrm{T}$. Atunci $v(\alpha \vee \beta) = \mathrm{T}$ și $v(\gamma \vee \neg\beta) = v(\gamma)$.
Dacă $v(\gamma) = T$, atunci $v(\alpha \vee \gamma) = T$, deci afirmația e adevărată.
Cazul 2: $v(\beta) = \mathrm{F}$. La fel, reducem la $\{\alpha\} \models \alpha \vee \gamma$ (adevărat).

Consistență și completitudine

$H \vdash \varphi$: *deducție* (pur sintactică, din axiome și reguli de inferență)
$H \models \varphi$: *implicație, consecință semantică* (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).

Deci, logica propozițională e *consistentă și completă*.

Ca să demonstrăm o formulă, putem arăta că e *validă*.
Pentru aceasta, verificăm că *negația ei nu e realizabilă*.